Empowering Data Analysis with Program Synthesis

Chenglong Wang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Rastislav Bodík, Co-Chair

Alvin Cheung, Co-Chair

Amy J. Ko

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Empowering Data Analysis with Program Synthesis

Chenglong Wang

Co-Chairs of the Supervisory Committee:
Prof. Dr. Rastislav Bodík
Paul G. Allen School of Computer Science and Engineering, University of Washington
Prof. Dr. Alvin Cheung
Electrical Engineering and Computer Sciences, University of California at Berkeley

Data manipulation and visualization support data scientists' efforts to explore and understand data throughout the exploratory analysis process. Nowadays, experienced data scientists can use programming languages like SQL and R to achieve efficient and flexible analysis, and inexperienced users can easily learn and use interactive tools to accomplish simple analysis tasks. However, the lack of tools in between interactive tools and programming systems leads to a *programmability gap* that prevents inexperienced users from conducting expressive analysis that only users with programming experience can achieve.

To help end users traverse this gap, we apply program synthesis to build tools that can synthesize programs from examples. We first introduce Falx, a visualization by example tool that lets the user create expressive visualizations using demonstrations of how a few data points are mapped to the canvas. Falx's compositional algorithm design let it synthesize both data transformation and visualization programs directly from end-to-end demonstration. We next introduce Scythe, a SQL query synthesizer that lets the user author advanced SQL queries using input-output examples. Using a language of abstract queries, Scythe can prune families of infeasible queries to achieve synthesis efficiency. To let inexperienced users distinguish synthesized complex queries, we developed a symbolic engine to compute a distinguishing input that the two queries would return different outputs. Finally, we summarize our synthesizer building experience into a framework, Kopis, that illustrates how to build an efficient relational query synthesizer using value-preserving abstractions. Together, these three contributions demonstrate the value of using program synthesizers to empower future data science, and offer guidance on how to build such synthesis-powered tools efficiently for new domains.

Table of Contents

Page

Introduction 1

Part I: Synthesis-Powered Data Visualization 10

Chapter 1: Falx: Synthesis-Powered Visualization Authoring 11
- 1.1 Introduction 11
- 1.2 Usage Scenario 14
- 1.3 System Architecture 21
- 1.4 User Study 26
- 1.5 Related Work 36
- 1.6 Discussion & Future Work 39

Chapter 2: The Visualization by Example Algorithm 41
- 2.1 Introduction 41
- 2.2 Overview 44
- 2.3 Problem Definition 50
- 2.4 Synthesis Algorithm 56
- 2.5 Implementation 63
- 2.6 Evaluation 64
- 2.7 Related Work 71
- 2.8 Summary 73

Part II: Synthesis-Powered Database Querying 76

Chapter 3: Scythe: Synthesizing Highly Expressive SQL Queries 77
- 3.1 Introduction 77
- 3.2 Overview 79
- 3.3 The SQL Language 85
- 3.4 The Language of Abstract Queries 86
- 3.5 Synthesis Algorithm 88
- 3.6 Evaluation 96

3.7 Related Work . . . 102
3.8 Summary . . . 103

Chapter 4: Speeding up Symbolic Reasoning for Relational Queries . . . 104
4.1 Introduction . . . 104
4.2 Problem Definition . . . 110
4.3 Overview . . . 112
4.4 Definitions . . . 116
4.5 Algorithm . . . 117
4.6 Evaluation . . . 124
4.7 Related Work . . . 129
4.8 Summary . . . 132

Part III: Program Synthesis with Value Preserving Abstraction 134

Chapter 5: The Kopis Framework: Synthesizing Relational Programs with Value Preserving Abstraction . . . 135
5.1 The Synthesis Task . . . 136
5.2 The Synthesis Algorithm . . . 138
5.3 Value-preserving Abstraction for Program Reasoning . . . 141
5.4 Summary . . . 145

Conclusion 147

Acknowledgments

I'd like to thank my advisors, Rastislav Bodik and Alvin Cheung.

They give me courage, knowledge, space and time.

They fill me with the determination to open a new path to the unknowns research land.

I'd like to thank my thesis readers Jeffrey Heer, Amy Ko, and Cecelia Aragon.

They give me the wings to view the research land from the sky.

I'd like to thank my mentors, Isil Dillig, Rishabh Singh, Pushmeet Kohli, Michael Ernst and Edward Grefenstette.

They take me to new joyful research lands that have been full of fog.

I'd like to thank my collaborators Yu Feng, Alex Polozov, Dominik Moritz, Shumo Chu, Amanda Swearngin, Po-Sen Huang, Rudy Bunel, Hanjun Dai, Krishnamurthy Dvijotham, Halden Lin, and Xi Victoria Lin.

They light the bonefire in dark nights of the research that is filled with with projects, discussions, milestones and friendship.

I'd like to thank all of the frields in the PLSE lab, especially Jared Roesch, Eunice Jun, Chandrakana Nandi, Sam Kaufman, Julie Newcomb, Jacob Van Geffen, Calvin Loncaric, Stuart Pernsteiner, Pavel Panchekha, Sarah Chasins, Mangpo Phothilimthana, James Bornholt, Rene Just, Zach Tatlok, and Enima Torlak.

They are the stars in the nightsky and give me a lifetime of friendship to look forward to.

I'd like to thank all of the Allen School staff and advisors.

I'd like to thank my friends, and my family.

I'd like to thank my wife, Tianzi Zhang.

A light fills the sky. It seems this journey is finally over. I am filled with determination.

It's time for a new journey ahead.

Dedication

to my wife, Tianzi

Introduction

Data manipulation and data visualization support data scientists' efforts to explore and understand data throughout the analysis process. Data scientists manipulate data to select desired data from relational databases, to improve its quality, and to fit it into a shape required by different analysis tools; data scientists create visualizations to explore patterns in the data [89, 91]. The importance of these tasks lead to the development of many tools, with the goal of making data visualization and data manipulation more accessible and increasing the user's data analysis productivity:

- *Programming tools:* Programming tools include both libraries in general purpose languages and domain-specific languages designed for data analysis. Examples of these tools include visualization libraries ggplot2 [203] (R), matplotlib [83] (Python), data transformation libraries tidyverse [204] and pandas [120], and DSLs like SQL, Vega-Lite [190]. Specialized in data analysis, these tools offer expressive high-level abstractions that allow data scientists to concisely specify complex analysis, but they are often only accessible to users with programming experience.

- *GUI (graphical user interface)-based interactive tools:* GUI-based interactive tools provide interactive panels that data scientists can directly manipulate data to solve their tasks. Examples of these tools include interactive visualizations tools Tableau [175], PowerBI [60], data wrangling tool Trifacta [90], OpenRefine [191] and domain specific analysis tools like SPSS [131] (for social science) and more. These tools are often easy to learn and use if the analysis task is within reach of the basic functionality of the tool. However, because these tools trade off expressiveness for ease-of-use, they can be challenging to use for more advanced tasks. In these cases, data scientists need to manually configure tool parameters (which requires considerable experience with the tool) or resort to low-level analysis tools like Excel or Adobe Illustrator to process the data (which requires more manual effort).

For data scientists with programming background, they can use these expressive powerful programming tools to solve complex tasks, and for end users like business people and data workers who only need simple analysis, they can easily use interactive tools to accomplish their goals. However, between these two set of tools there is a *programmability gap*: an inexperienced user who want to transition from interactive tools to programming tools to

solve complex data analysis tasks must need to learn to program, an skill challenging for non-programmers to acquire.

In practice, these is a large set of users who fall into this gap, including social scientists [67], journalists [92], genome scientists, advanced business analysts [89]. On the one hand, these people are often not professionally trained as programmers and some of them lack the experience to work without programming tools [67]; on the other hand, many tasks are beyond basic functionality of existing GUI-interactive tools (e.g., create heatmaps to visualize large scale genomic data; create geographic maps for geospatial analysis).

This *programmability gap* brings the need to design a new generation of tools that retain the expressiveness and flexibility of the programming tools to support advanced tasks these data scientists need, but act like an interactive tool so that it is easy to learn and use (even for advanced tasks). To bridge this gap, we build program synthesis-powered tools — tools that synthesize programs from examples and demonstrations. In comparison to existing interactive tools that require the user the precisely specify the analysis task, synthesis-powered tools allow the user to provide incomplete tasks specifications to demonstrate the task, and they have the power to generalize user specification into programs that can solve the full analysis task. Unlike full task specifications that will become more complex as the complexity of the analysis tasks increase, partial task specifications can still be simple even for complex tasks because the user can omit some analysis details in the demonstration. In this way, synthesis-powered tools let users solve complex analysis tasks with simple, incomplete task interaction.

In this thesis, we introduce two synthesis-powered data analysis tools: the first tool, Falx, is a synthesis powered data visualization tool that lets users create expressive visualizations from demonstrations, and the second tool, Scythe, is a query-by-example tool, that lets users author SQL queries using input-output examples.

- Falx [194, 195] is designed to address the challenge that creating expressive visualizations requires data scientists to be experienced with not only plotting tools but also data transformation tools to prepare the data to match the format desired by the visualization design. Falx is a visualization-by-example tool that can synthesize programs to prepare and visualize data from demonstrations (Figure 1 top): the data scientist demonstrates how a few data point in the dataset should be mapped to the canvas, and Falx finds a program that generalizes the mapping to visualize the full dataset. Our user study with 33 data scientists shows that users can effectively and confidently adopt Falx to create visualizations that they otherwise could not implement due to their lack of programming expertise.

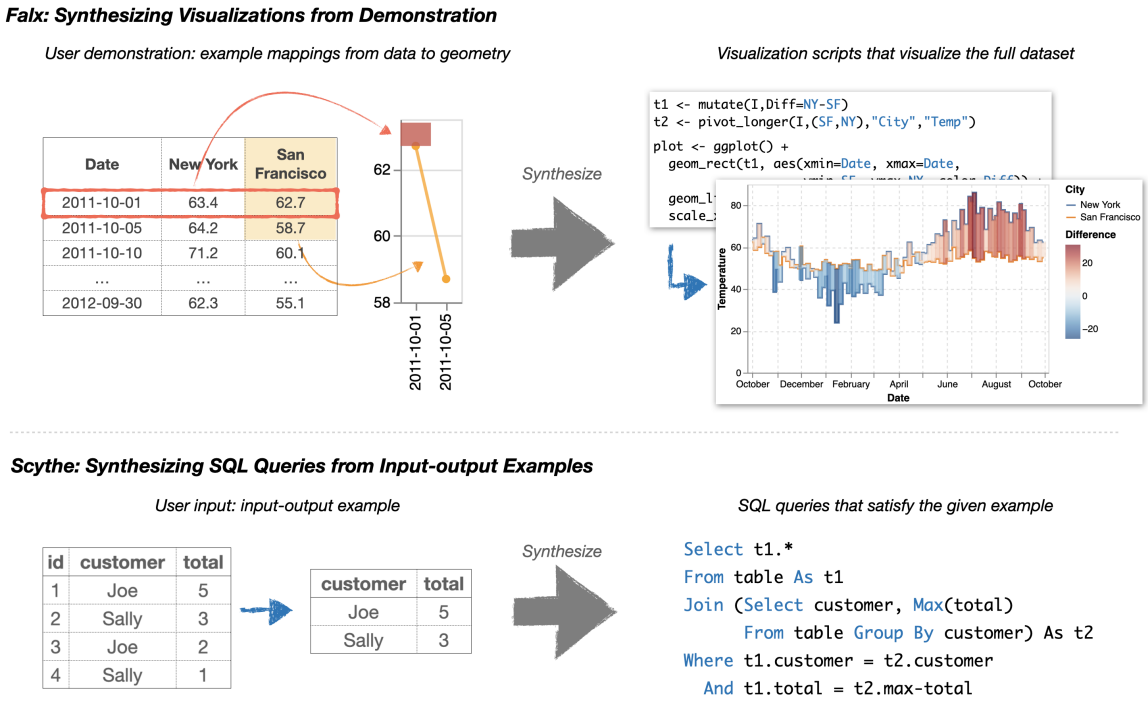


Figure 1: (1) Falx: synthesizing visualizations from demonstrations. (2) Scythe: SQL query synthesizer from input-output examples,

- Scythe [192, 193] is designed to address the challenge that many practical data querying tasks require using advanced SQL features, like aggregation, subqueries and outer-joins, that many data scientists struggle with. We developed Scythe, a query-by-example tool that can generalize input-output examples, a medium novice users often use in online forums like Stack Overflow to demonstrate their tasks to experts, into highly expressive SQL queries that can be used to retrieve data from full databases (Figure 1 bottom). When tested on 193 real-world tasks from Stack Overflow, Scythe solved 70% of the tasks, and 80% of the solved tasks can be solved within 10 seconds.

In the following, we review challenges in data visualization and database querying and walk through the designs of synthesis-powered tools to solve these challenges.

The Programmability Gap

Data Visualization Challenges

Many modern data visualization tools [154, 203, 175, 60, 187, 108, 152] are built on top of the "tidy data" [205] assumption where each column in the data table represents an observation

and each row corresponds to one observation. This assumption allows a simplification of the visualization process [206]: the user only needs to specify mappings from data columns to geometric properties of the visualization marks (e.g., x, y-positions or colors of a point) and the tool will attach each row in the data table to one mark in the visualization.

However, these systems' reliance on the "tidy data" assumption raises the data transformation problem for the users: authoring a visualization may require the backing dataset to be structured or formatted in a particular way that may not be clear to authors priori [153]. Even for users that understands required data layouts for their visualizations, using data transformation tools [204, 120, 90] to perform the transformation remains highly challenging [207, 153, 89]. First, identifying observations/variables from an untidy data requires understanding semantics of the dataset, which cannot be fully automated, and users of mixed-initiative data transformation tools [90] still face the challenge to understand and guide the data transformation process [94]. Second, the data transformation task often requires more than one data transformation step that involves a collection of transformation operators, and breaking down a complex data transformation task into small steps requires experience [58]. Third, different data transformation languages, libraries and tools provide different sets of operators that make transitions between data analysis platforms difficult [67]. Finally, not all visualization designs take the default tidy data format as the input because different designs may need different sets of variables for geometric properties [153], and this raises the overhead for design exploration in an exploratory analysis setting.

As a result, visualization tool designers made the following reflection [153]:

> *"It is not sufficient to simply extend the underlying visualization models to provide data transformation capabilities. Such an approach presents a non-trivial gulf of execution by expecting authors to manually define data transformations. Instead, these systems must develop higher-level scaffolding that automatically infers or suggests appropriate transformations when necessary."*

Our solution to this challenge is the design of a synthesis-powered visualization tool, Falx, that allows the user to specify visualization in a data layout independent way using demonstrations. From these demonstrations, Falx automatically synthesizes data transformation and visualization scripts to accomplish the visualization task. As we will show more in Chapter 1 and Chapter 2, Falx lets users directly specify visualization on messy data and let users create visualizations they otherwise cannot create.

Database Querying Challenges

SQL is one of the most commonly used query languages to query relational data. SQL can be used for many basic tasks, such as selecting columns from a table; its rich features also

make it useful for solving complex data manipulation tasks, such as computing argmax and joining multiple tables together with aggregates.

However, the various operators available in SQL and the many ways that they can be combined to form advanced idioms (e.g., correlated subqueries, unions, nested queries, groupings, various types of joins, etc) make the language difficult to master. While experienced programmers can sometimes bypass this problem by scripting imperative programs on top of simple SQL queries to achieve the same goal, inexperienced users have little choice other than asking for help from others, which is evidenced by over 10,000 Stack Overflow SQL related posts. In fact, many problems are so common among end users that they are grouped with popular tags, such as "greatest-n-per-group," "argmax," and "moving-average."

Despite prior attempts to build query-by-example tools [220, 159] and query-by-natural language tools [104, 73, 214] to make databases more accessible, these tools are not expressive enough for many practical tasks. Can we build an expressive tool that can act like an expert to help inexperienced users to answer their database querying questions? As we will show in detail in Chapter 3 and Chapter 4, our answer is to build a new SQL query synthesizer, Scythe, that can synthesize *highly expressive* SQL queries from input-output examples. Unlike prior tools that supports only basic relational algebra, Scythe supports SQL with advanced features like outer-join, aggregation and subqueries, making it expressive enough to tackle $\sim 70\%$ of practical SQL problems users post on online forums.

The Design of Synthesis-powered Tools

Synthesis tools take as input the input data and the user demonstration and outputs a set of programs that are consistent with the demonstrations. There are three main components inside a synthesis tool: (1) *an input specification* the let users concisely demonstrate a potentially complex data analysis task, (2) *a scalable synthesis algorithm* that lets the synthesizer search for programs that generalize the user demonstration, and (3) *an interactive model* that let the user explore synthesized programs to find the desired solution. Table 1 shows these components in Falx and Scythe, and we elaborate them below.

Input Specification

The input specification decides what information the user can provide to illustrate the task and how the user can provide such information to the synthesis tool. Here, the goal is to design a specification that is (1) natural and easy for the user to provide and (2) reasonably unambiguous to capture the user's intent (to allow better generalization).

In Falx, the user specifies the visualization task by demonstrating how a few data points (from the untidy input data) should be mapped to the visualization canvas (*visualization*

	Falx	Scythe
Domain	Data Visualization	Database Querying
User Specification	Demonstrations of how a few data points in the dataset are mapped to the canvas	Input-output example tables
Output	Pairs of data transformation and visualization programs whose output visualizes contain all demonstrated points	SQL queries consistent with the input-output examples
Synthesis Algorithm	Compositional synthesis of data transformation and visualization programs; enumerative search that leverages bidirectional value propagation to reason and prune infeasible subspace	Enumerative search process that uses the language of abstract query to reason and prune families of infeasible queries
Interaction Model	An visualization exploration interface that supports coarse-to-fine exploration of synthesized visualization; a visualization editor that supports fine tuning visualization parameters	A symbolic reasoning engine that computes distinguishing inputs that can differentiate behaviors of similar queries

Table 1: Main components of Falx and Scythe

by example). This design is motivated by observation from a formative user study that data scientists often create partial visualization sketches in the form of example mappings from input data to visual channels (i.e., graphical mark attributes like x, y-positions and color) to illustrate their tasks. In fact, this *example mapping* design is a generalization of the grammar of graphics specification [206], a specification widely used in modern visualization tools. In the grammar of graphics, users precisely specify visualizations by mapping data columns in a tidy dataset to geometric properties. The visualization by example specification is a partial specification consists of mappings from data points to geometric properties, which inherits the simplicity of the grammar of graphics while bypassing the need of data transformation.

Scythe lets the user demonstrate a database querying task using small input-output example tables. This design is motivated by the observation that many Stack Overflow users could often concisely describe their tasks using small input-output examples to seek help from forum experts. Unlike Falx that the users can directly use the raw untidy data as the input and not all input data points needs to be mapped to the demonstration (because the input data size is often large), Scythe users need to craft a small input table and then provide an output example which is a full output with respect to its small input table. In comparison,

Falx's specification has lower specification effort as the user does not need to manually craft a small input (the challenge is to make the example both small and representative for task); despite requires more specification effort, Scythe's specification contains more information about the task and supports negative reasoning: tuples not appeared in the output table should be excluded by the query (which allows the synthesizer to exclude queries whose outputs contain more tuples). Whereas in Falx, a point not appeared in the demonstration could simply be something not demonstrated by the user, and Falx should consider all visualizations that contain user demonstrated points as candidate solutions for the user's task.

Synthesis Algorithm

The synthesis algorithm behind a synthesis tool needs to solve a challenging combinatorial search problem: to find programs from the program space defined by a language that are consistent with the user specification.

In Scythe, the key synthesis challenge comes from the expressiveness of SQL. Comparing to other problem domains like string transformation [71], web extraction [102], assembly code super optimization [138], synthesizing SQL faces the following challenges: (1) SQL is highly parametric with a huge parameter search space to explore, (2) table is the first class value in SQL, which makes query evaluation cost and table memoization cost considerable higher, and (3) SQL is not a language that can be decomposed in polynomial complexity to be solved efficiently in existing frameworks. To address these challenges, we develop the language of abstract SQL to decompose the complex search problem into two stages: (1) searching and pruning abstract queries (queries with all parameters remain unfilled), and (2) instantiating parameters for potential candidate abstract queries. Given an abstract query, this language abstraction allows Scythe to propagate input data through it to derive an over-approximation output to summarize all of its possible outputs, which can then be compared against the user specification to decide whether this family of queries are feasible. This design lets Scythe dramatically prune the search space (with an average reduction of 2145$\times$ in our evaluation) with little overhead.

In addition to similar challenges in Scythe, Falx faces another two new challenges: (1) Falx needs to synthesize programs from two languages (data transformation and data visualization) from end-to-end specification, and (2) because Falx supports partial output in the demonstration, the language abstraction in Scythe is no longer sufficient to prune dramatically. To solve the first challenge, Falx adopts a compositional synthesis algorithm. Falx first decompiles the user demonstration to obtain the visualization program and an intermediate table; it then invokes a data transformation synthesizer to search for the desired data transformation program. For the second challenge, Falx uses a bidirectional analysis

algorithm to analyze behaviors of abstract programs encountered in the search process: besides a forward analysis process similar to Scythe that summarizes output behavior of the abstract program, it employs a backward analysis to backwardly propagate values from the user demonstration to derive a precondition the inputs need to satisfy to conform the user specification. These innovations allow Falx to better prune infeasible search space. In evaluation, Falx solved 70 of 84 real-world visualization tasks within 20 seconds, while prior state of the art can only solve 49 out of 84 tasks.

In summary, the key innovation of the synthesis algorithm design brought up by this thesis is the use of *value-preserving abstractions* to reason about abstract programs to enable dramatic early pruning of the search space. This broadens the design space of synthesis algorithms: when we cannot design the language to be both expressive and decomposable (like in the Prose framework [140]), we can design powerful value-preserving abstractions to scale up the synthesis algorithm. We summarize this approach as the Kopis framework that we will introduce in Chapter 5.

Interaction Model

A data scientist's interaction with a synthesis tool does not end when candidate programs are returned from the synthesis algorithm. Because the user specification is an incomplete specification of the data analysis task, ambiguity exists and the synthesis tool can generalize the user specification in many different ways. This brings up the need to design an interaction model to allow the user to understand and select the desired program from a pool of candidate programs that are all consistent with the user specification. Because users of the synthesis tool are often inexperienced programmers or non-programmers, it is not idea to let the user directly read the programs to disambiguate them [99].

Scythe leverages a symbolic reasoning engine to reason about semantic differences between a pair of SQL queries. Given a pair of queries, the symbolic engine can efficiently compute a small distinguishing input that lets the two queries return different results. In this way, the user can distinguish the queries based on their outputs.

Falx, instead, transforms this challenging program disambiguation problem into a visualization exploration problem with the design of an exploration interface that lets users navigate solutions in the visualization space. Using the exploration interface, users can coarsely scan all designs to quickly rule out visualizations with high-level errors (e.g., wrong axes or labels) and then side-by-side compare similar ones in detail to choose the desired solution. Once they identified the desired visualization, the user can further proceed to fine-tune the visualization with an editing panel to interactively improve the visualization design. In our user study with 33 participants, Falx users showed statistically significant efficiency improvements on two challenging visualization tasks compared to users of R.

Participants found that Falx was "easy to learn," "fast" and "can generate something that you cannot easily do otherwise," and they were "confident about solutions" (Chapter 1).

In general, the key of interaction model design in Falx and Scythe is based on transforming the disambiguation problem in the *program space* to a disambiguation problem in the *value space* (tables in Scythe and visualizations Falx). In this way, an inexperienced data scientist can effectively and confidently select the desired program that can solve the task.

Contributions and Outline

The remainder of this dissertation is arranged in three parts.

Part I presents Falx, a synthesis-powered data visualization tool, to demonstrate how program synthesis can empower non-experienced data scientists to create expressive data visualizations that involve non-trivial data transform effort. In this part:

- Chapter 1 presents the specification and interactive model designs of user-synthesizer Falx, and a study of user experience with Falx.
- Chapter 2 presents the synthesis algorithm behind Falx, featuring the design of the compositional synthesis algorithm and the use of bidirectional value propagation to reason about abstract programs.

Part II presents Scythe, a query-by-example tool, that empowers end users to author complex SQL queries from input-output examples. In this part:

- Chapter 3 presents Scythe's synthesis algorithm, featuring the design of the language of abstract queries to reason and prune families of infeasible SQL queries.
- Chapter 4 presents the interaction model for Scythe. Scythe uses a symbolic reasoning engine to compute a distinguishing input that lets the two queries return different results for user disambiguation.

Part III summarizes the value-preserving abstraction synthesis framework we distilled from our synthesis algorithm design in Falx and Scythe. Kopis supports fast prototyping of efficient program synthesizers for relational queries.

Together, these chapters support the thesis at the core of this dissertation: (1) *synthesis-powered data analysis tools can bridge the programmability gap for non-experienced data scientists by allowing users to specify tasks using partial specifications*, and (2) *value-preserving abstraction can scale up program synthesis algorithm to solve practical complex data analysis problems.*

Part I

SYNTHESIS-POWERED DATA VISUALIZATION

Chapter 1

Falx: Synthesis-Powered Visualization Authoring

Modern visualization tools aim to allow data analysts to easily create exploratory visualizations. When the input data layout conforms to the visualization design, users can easily specify visualizations by mapping data columns to visual channels of the design. However, when there is a mismatch between data layout and the design, users need to spend significant effort on data transformation.

In this chapter, we introduce Falx, a synthesis-powered visualization tool, that allows users to specify visualizations in a similarly simple way but without needing to worry about data layout. In Falx, users specify visualizations using examples of how concrete values in the input are mapped to visual channels, and Falx automatically infers the visualization specification and transforms the data to match the design. In a study with 33 data analysts on four visualization tasks involving data transformation, we found that users can effectively adopt Falx to create visualizations they otherwise cannot implement.

1.1 Introduction

Modern visualization authoring tools, such as declarative visualization grammars like ggplot2 [203], Vega-Lite [154] and interactive visualization tools like Tableau [174] and Voyager [209], are built to reduce data analysts' efforts in authoring visualizations in exploratory data analysis. At the heart of these tools, visualizations are specified using grammars of graphics [206], where every visualization can be succinctly specified using the following three components:

- A graphical mark that defines the geometric objects used to visualize the data (e.g., line, scatter plots, bars),
- A set of visual encodings that map data variables to visual channels (e.g., x, y-positions of points),
- A set of parameters that decide visualization details: coordinate system, scales of axes, legends and titles.

In practice, users only need to specify the mark and the visual encodings in order to create the visualization because many tools use a rule-based engine to automatically fill in parameters

for visualization details (often referred to as "smart defaults") unless the user wants further customization. The abstraction of graphical marks, visual encoding channels, and adoption of smart default parameters open an expressive design space for data analysts that allow them to rapidly construct visualizations for exploratory analysis through simple specifications. For example, to visualize the dataset in Figure 1.1 with three columns Date, Temp (for temperature) and Type as a scatter plot, the user can choose the graphical mark "point" with encodings $\{x \mapsto \text{Date}, y \mapsto \text{Temp}, \textit{color} \mapsto \text{Type}\}$. The visualization tool then creates one point for each row in the input data, by mapping its values in columns Date and Temp to x,y-positions and assigning a color to each point based on its value in column Type. Here, the tool uses the default linear scale for x,y-axis and categorical scale for color, which are default parameters that the user does not need to specify explicitly. The final visualization is rendered in Figure 1.1 (right).

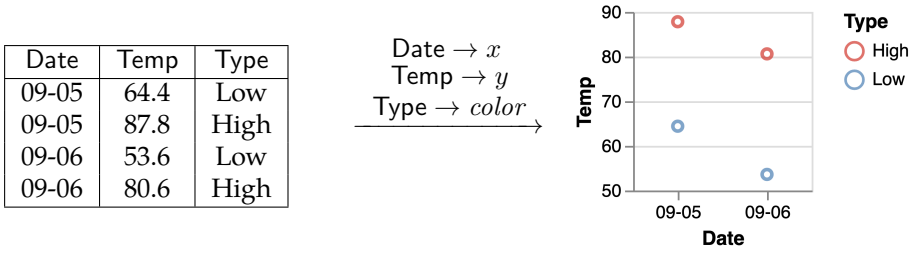


Figure 1.1: An example dataset and its scatter plot visualization that maps Date to x, Temp to y and Type to $\textit{color}$.

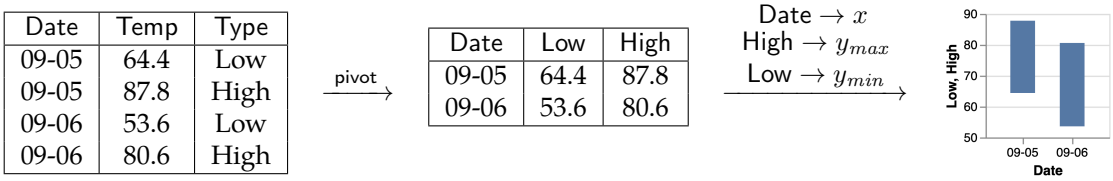


Figure 1.2: A different visualization design requires transforming the original input data.

In fact, the simplicity of these high-level visualization grammars is grounded in their abstract data model. These grammars expect that the input table is organized in a layout that matches the visualization design [205]: (1) each relation forms a row in the input data and corresponds to exactly one geometric object in the visualization, and (2) each data variable forms a column that can be mapped to a visual channel. In practice, however, the mismatch between the data layout and the visualization design is common due to the following reasons [67, 205]:

- Tables exported from different sources (e.g., database, analysis tool, team members) may have different layouts and they may not directly match the visualization design.

- Different analysis tasks require different visualization designs, and changes in the design can lead to different expected data layout.

- The data may need aggregation (e.g., average, count, culminative sum) or additional computation to derive new values prior to visualization.

In all of these cases, data analysts cannot directly visualize the data with a simple specification. They have to conceptualize the expected data layout and utilize data transformation tools (e.g., tidyverse [205], Trifacta [90]) to transform the data to match the visualization design. These additional tasks create a barrier for data visualizations and greatly increase the effort required for exploratory analysis [67, 58, 207, 91]. For example, if the data analyst decides to change the visualization in Figure 1.1 to a bar chart with floating bars that show the temperature range during each day (Figure 1.2 right), the original data layout will no longer match the new design since the new design expects three data columns (date, lowest temperature, highest temperature) that map to x, y_{max} and y_{min}. As a result, the data analyst needs to transpose the table in Figure 1.1 using a pivot operation (to collect key-values pairs in the Type and Temp columns into new columns) before mapping data columns to visual channels (Figure 1.2 right).

We propose Falx, a synthesis-based visualization authoring tool to address the challenges outlined above. Falx builds on recent advances in program synthesis: many program synthesis tools (e.g., FlashFill [71], Wrex [52]) have been developed with the promises of automating challenging or repetitive programming tasks for end users by synthesizing programs from user demonstrations. In our design, instead of asking analysts to transform data and specify visualization manually, Falx asks analysts to *demonstrate* the visualization task using examples of mappings from concrete values in the input data (as opposed to table columns) to visual channels. Using these examples, Falx automatically synthesizes the programs to transform and visualize the full data, such that resulting visualizations are consistent with the examples (i.e., all example mappings are contained within the visualization). For example, for the data in Figure 1.2, the user can create an example bar 

in Falx to demonstrate the task and let Falx create the desired visualization for the full dataset (Figure 1.2 right). Sometimes, the examples can be ambiguous to Falx, and Falx may generate multiple visualizations that match the example but not necessarily the user intent. In such cases, analysts can interact with a built-in exploration panel to inspect the synthesized visualizations and select the desired one. Once the de-

sired visualizations are found, analysts can further fine-tune visualization details through a post-processing panel.

Falx's design has many potential advantages. First, users of Falx specify visualizations by mapping values to visual channels: this approach inherits the simplicity from grammars of graphics but provides more expressiveness since users can use the same examples to specify visualization ideas for inputs with different layouts. Second, Falx offloads the data transformation task to the program synthesizer so that users no longer need to conceptualize the expected data layout or transform the data. Finally, while program synthesizers by design can generate multiple results, users can effectively select and validate the desired visualization from synthesized candidates using the exploration panel in Falx. In general, rather than having to construct a visualization, data analysts demonstrate the task using examples and then select the desired visualization from a candidate pool, which shifts from the challenges of expression to the ease of recognition. With these designs, Falx aims to eliminate users' prerequisites in data transformation and enable data analysts to rapidly author visualizations.

We conducted a user study with 33 participants to test these design hypotheses, studying how users adapt to the new visualization process. Our results show that users of Falx, regardless of previous experience in visualization, can efficiently learn and solve challenging visualizations tasks that cannot be easily solved using the baseline tool ggplot2. However, we also discovered challenges that users face when using the tool and strategies they adopt to solve the problems. We believe these discoveries lead to future opportunities in adopting synthesized-based visualization tools in practice and unveil other potential designs that can further improve the usability of such tools.

1.2 Usage Scenario

We first go through an example to illustrate the anticipated user experience in Falx (Section 1.2.2) compared to R (Section 1.2.1). In this example, a data analyst has the following dataset with New York and San Francisco temperature records from 2011-10-01 to 2012-09-30.

Date	New York	San Francisco
2011-10-01	63.4	62.7
2011-10-05	64.2	58.7
...	...	...
2012-09-25	63.2	53.3
2012-09-30	62.3	55.1

The analyst wants to create a visualization to compare the temperature in the two cities. First, the visualization should contain two lines to show temperature trends in the two

cities; these two lines should be distinguished by color. Second, on top of the line chart, a bar chart should be layered on top to show the temperature difference between the two cities for each date. Each bar should start from the New York temperature and end at the corresponding San Francisco temperature, and the color gradient of the bar should indicate the temperature difference between the two cities on that day. The desired visualization is shown in Figure 1.3.

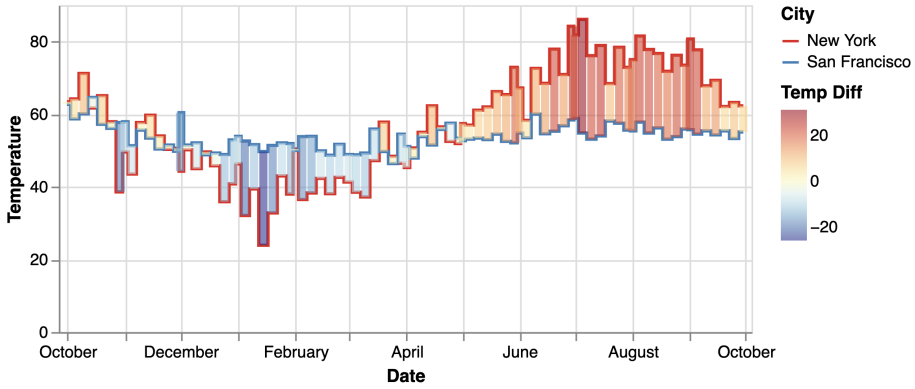


Figure 1.3: A visualization that compares New York and San Francisco temperatures between 2011-10-01 and 2012-09-30.

1.2.1 User experience in R

We first illustrate how a data analyst, Eunice, would create this visualization in R using tidyverse [205] and ggplot2 [203], two widely-used libraries for data transformation and data visualization.

After loading the data into a data frame in R, Eunice decides to first create the line chart that shows temperature trends of the two cities. To do so, Eunice chooses the function `geom_line` from the ggplot2 library. In order to create lines with different colors for different categories, Eunice needs to supply four data variables to the `geom_line` function – two variables for specifying x and y positions, one for colors of the line, and the last one for groups of lines (i.e., which points belong to the same line). Since the input data does not have these variables, Eunice needs to use the tidyverse library to transform the input data. To do so, Eunice first conceptualizes the desired data layout: the data should have 3 fields—date (for x-axis), temperature (for y-axis), and city name (for color and group). Eunice recalls a function `pivot_longer` in tidyverse, which supports pivoting the table from a "wide" to a "long" format by collecting column names and values in the column as key-value pairs

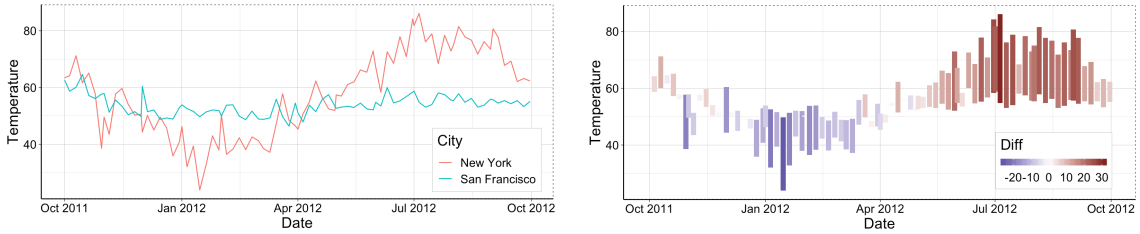
in the body content. Specifically, Eunice writes the following code to transform the data, which yields the data on the right that matches Eunice's expectation.

```
df1 <- pivot_longer(data = df,
         cols = ("New York", "San Francisco"),
         names_to = "City", values_to = "Temperature")
```

Date	City	Temperature
2011-10-01	New York	63.4
2011-10-01	San Francisco	62.7
...	...	...
2012-09-30	San Francisco	55.1

After data transformation, Eunice specifies the visualization using the following script. The script maps Date to x-axis , Temperature to y-axis, and City to both color and group. and it generates the visualization in Figure 1.4a.

```
plot1 <- ggplot(data = df1) +
         geom_line(aes(x = `Date`, y = `City`,
            color= `Temperature`, group = `Temperature`))
```



(a) A line chart that shows temperature trends. (b) A bar chart showing temperature difference.

Figure 1.4: Two visualizations created in R that compare New York and San Francisco temperatures.

Eunice then proceeds to create bars on top of the first layer to visualize the temperature difference. Eunice first finds the function `geom_rect` from the library that supports floating bars. To visualize temperature difference, Eunice needs to specify positions of bars by mapping Date to x_{min} and x_{max} properties and mapping temperatures of the two cities to y_{min} and y_{max}; she also needs to map the temperature difference between the two cities to $color$ to specify bar colors. Since the original data does not contain a column for temperature

difference, Eunice uses the `mutate` function from tidyverse to transform the data. Using the following script, Eunice successfully creates the visualization in Figure 1.4b.

```
df2 <- mutate(df, Diff = `New York` - `San Francisco`)
plot2 <- ggplot(df2) +
         geom_rect(aes(xmin = `Date`, xmax = `Date`,
            ymin = `New York`, ymax = `San Francisco`,
            fill = `Diff`))
```

Finally, Eunice restructures the code to combine the two layers together using a concatenation operator. She also fine-tunes some parameters in ggplot2 to improve visualization aesthetics (e.g., modify titles of the axes and change line chart to a step chart), which generates the visualization that matches her design in Figure 1.3.

Since Eunice is an experienced data analyst, she manages to go through these data transformation and visualization step and eventually generates the desired visualization. However, a less experienced data analyst, Amelia, finds the visualization task challenging.

- First, Amelia is not familiar with the ggplot2 library, so she struggles in identifying the right functions to use. For example, it is difficult for her to distinguish between `geom_path` and `geom_line`, and `geom_bar` or `geom_rect`. She is also unfamiliar with how to compose multi-layered visualizations.
- Second, due to her lack of experience with ggplot2, she finds it difficult to conceptualize the expected input data layout because different functions and tasks require different input layouts.
- Finally, due to her lack of experience with the tidyverse library, she needs to spend significantly more time in finding the right operators and implementing the desired transformation.

1.2.2 Falx User Experience

Now we show how Amelia, a less experienced data analyst, uses Falx (Figure 1.5) to create the same visualization.

First, Amelia uploads the input data to Falx's input panel (Figure 1.5-①) and examines the input data displayed in a tabular view. Amelia decides to first visualize temperature trends of the two cities using a line chart. Amelia goes to the demonstration panel to demonstrate how the first two data points of New York temperatures will be visualized. To do so, Amelia first clicks the "+" icon in the interface and select a line element (Figure 1.6-①), and Falx pops out an editor panel for Amelia to specify properties of this line element. Amelia clicks

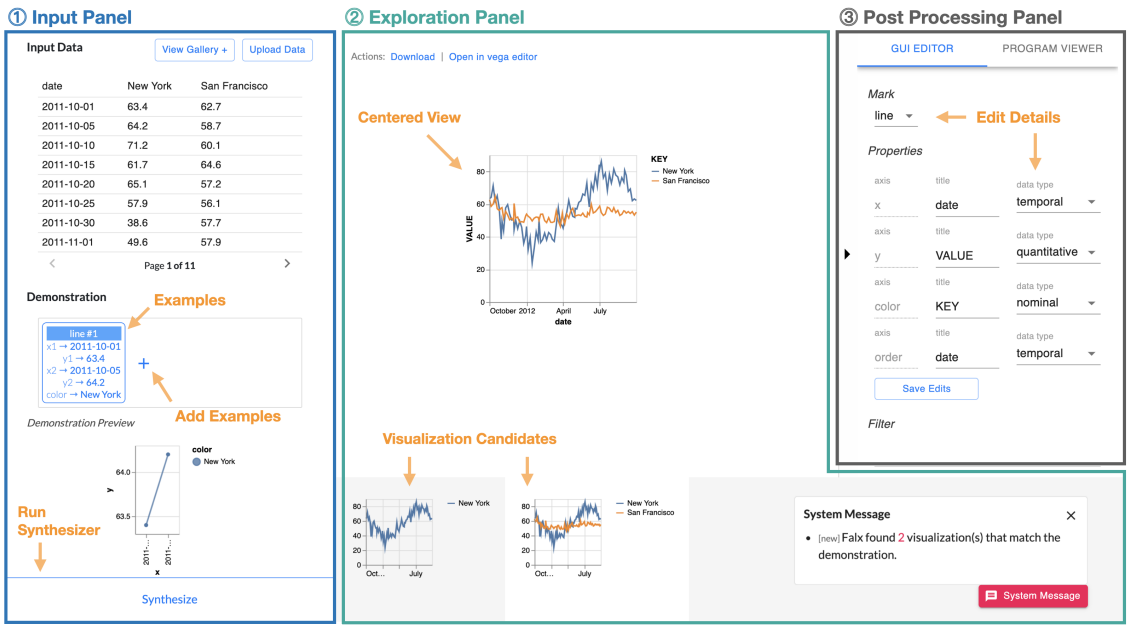


Figure 1.5: Falx interface has three panels: (1) Data analysts import data and create examples in the input panel. (2) Analysts explore and examine synthesized visualizations in the exploration panel. (3) Analysts edit visualization details in the post processing panel.

on values in the input table and copies the values to specify properties of the line element as follows (Figure 1.6-②):

- The line segment starts at the point with x_1 = 2011-10-01, $y_1 = 63.4$ (New York temperature on 2011-10-01)
- The line ends at x_2 = 2011-10-05, $y_2 = 64.2$ (New York temperature on 2011-10-05)
- The color of the line is labeled as "New York"

After saving the edits, Falx registers the example and provides a preview that visualizes the example line segment (Figure 1.6-③) for Amelia to examine. Using this example, Amelia conveys the following visualization idea to Falx: "I want a line chart over the input data that contains the demonstrated line segment". Amelia then presses the "Synthesize" button (in Figure 1.5-①) to ask Falx to find the desired line chart. Internally, Falx first infers the visualization specification and then runs a data transformation synthesizer to transform the input data to match the visualization specification. After approximately four seconds, Falx finds two visualizations that match the example and displays them in the bottom of the exploration panel (Figure 1.5-②). Both visualizations contain the example line segment

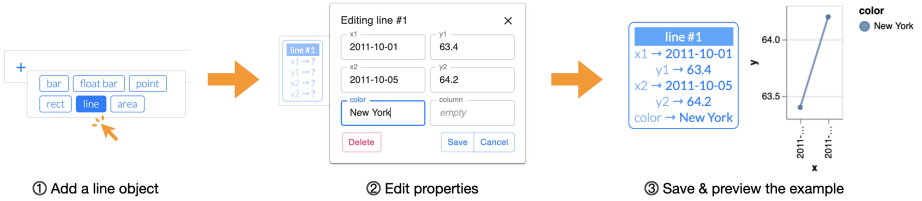


Figure 1.6: Amelia creates a line segment to demonstrate the visualization task.

specified by Amelia but they generalize the example differently: the first visualization only visualizes New York temperatures as demonstrated in the example, while the second generalizes the color dimension to other columns in the input data as well, resulting in a visualization that also contains San Francisco temperatures.

After briefly navigating both candidates in the carousel, Amelia finds the second visualization closer to the design in her mind, so she clicks the second visualization to enlarge it in the center view for a detailed check (Figure 1.5-② top). In the center view, Amelia hovers on the visualization to check details like values of different points in each line. After confirming the visualization matches her design, Amelia moves on to the second layer visualization, which should display temperature differences between the two cities using a series of bars.

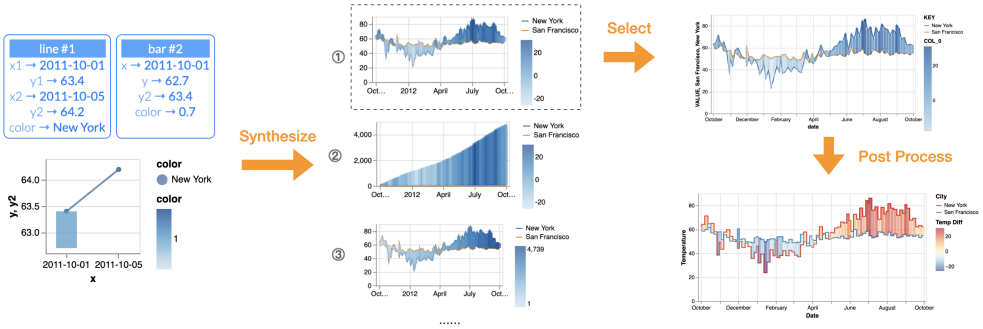


Figure 1.7: Amelia's interaction with Falx to create the second layer visualization.

Next, Amelia creates an example bar to demonstrate how the temperature difference between the two cities on 2011-01-01 should be visualized (Figure 1.7 left): the bar is positioned at date 2011-10-01, it starts at 62.7 (San Francisco temperature), ends at 63.4 (New York temperature), and its color shows the temperature difference of 0.7 for that day. Amelia runs the synthesizer to find visualizations that contain both the example line and the example bar. This time, after 9 seconds, Falx finds 8 candidate visualizations that match the examples (Figure 1.7 middle). To decide which visualization to pick, Amelia can either (1) add a

second example bar to demonstrate the temperature difference of the two cities on another date to help Falx resolve the ambiguity, or (2) navigate candidates in the exploration panel to examine them. Amelia decides to use the second approach again. She first rules out some obviously incorrect visualizations (e.g., visualization 2 in Figure 1.7 middle), then compares similar visualizations, and finally selects the first visualization to check it in detail. After some examination, she decides it matches her design and proceeds to post-process the visualization.

The post processing panel (Figure 1.5-③) contains a GUI editor that allows Amelia to fine-tune visualization details and a program viewer for viewing and editing the synthesized program. Any changes made during the editing process are directly reflected on the center view panel (Figure 1.5-②) to provide immediate feedback. Using the post-processing panel, Amelia changes the line mark to step mark and modifies axis titles, which produces the visualization in Figure 1.7 right. Amelia is happy with this visualization and concludes the task. If Amelia wants to further customize the visualization (e.g., change color scheme, adjust bar spacing), she can directly edit the underlying Vega-Lite program.

In sum, Amelia creates the visualization by iterating through creating examples, exploring synthesized visualizations, and post processing. In this process, she benefits from the following design decisions behind Falx:

- First, while two visualization layers require different data transformations, Amelia does not need to worry about this, as the transformation task is delegated to the underlying synthesizer. In fact, even if the input data comes with a different layout, Amelia can still solve the problem with the same examples.

- Second, Amelia specifies examples by choosing from a small set of visualization marks and specifying mappings from concrete data values to properties. This allows her to create visualizations without programming in the visualization grammar.

- Third, instead of asking Amelia to read synthesized programs to disambiguate synthesis results, Falx provides an exploration interface that allows Amelia to explore and examine results in the visualization space.

- Finally, Falx adopts a scalable synthesis algorithm to explore the exponential number of possible ways to transform and visualize the input data. Each synthesis run takes between 3 and 20 seconds, which makes Amelia conformable at iterating between giving examples and exploring the generated visualizations.

1.3 System Architecture

In this section, we first provide a brief background on program synthesis and then discuss the design and implementation of Falx, our end-to-end synthesis tool for automating data visualization tasks.

1.3.1 Background: Program Synthesis

In recent years, many program synthesis algorithms have been developed to automate challenging or repetitive tasks for end users by automatically generating programs from high-level specifications (e.g., demonstrations, input-output examples, natural language descriptions). For instance, programming-by-example (PBE) is a branch of program synthesis that aims to synthesize programs that satisfy input-output examples provided by the user, such tools been used for string processing [71, 166], tabular data transformation [58, 211, 192], and program completion [171, 172, 144, 75, 114].

While there are different approaches to synthesize programs, one common method is to perform enumerative search over the space of programs by gradually expanding programs from a context-free grammar of some language [61, 2, 186, 211]. In general, these search techniques traverse the program space according to some cost metric and return the candidate programs that satisfy the user-provided specification. Here, the cost metric can be a model that measures simplicity of programs (e.g., based on number of expressions in the program) [61] or a statistical models that estimate likelihood of the program being correct [6, 144]. To speed up the synthesis process, several recent methods use deduction rules to prune incorrect *partial programs* early in the search process [61, 58]. For instance, Morpheus [58] uses predefined axioms of table operators to detect conflicts before the entire program is generated.

1.3.2 Falx Synthesizer

The architecture of Falx is shown in Figure 1.8. To use Falx, a data analyst first provides an input table and creates examples to demonstrate the visualization idea. Once the analyst hits the "synthesize" button, the Falx interface sends the input and examples to the Falx server. Given an input data and an example visualization (in the form of a set of geometric objects), Falx synthesizes pairs of candidate data transformation and visualization programs such that the resulting visualization contains all geometric objects in the visualization example.

To synthesize visualizations consistent with examples from the user, Falx spawns multiple solver threads to solve the synthesis problem in parallel. In each solver thread, Falx first runs a *visualization decompiler* (step 1) to decompile the example visualization into a visualization program and an example table, such that applying the program on the example table yields the example visualization provided by the user. Then, Falx calls the *data transformation*

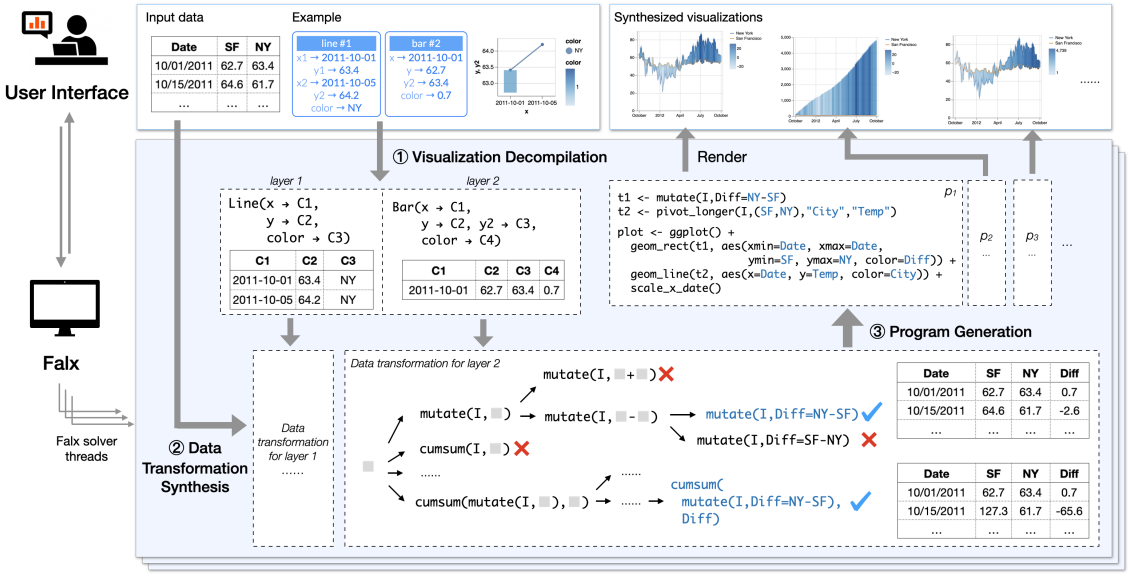


Figure 1.8: The architecture of the Falx system. Each solver thread synthesizes visualizations that match user examples in three steps: (1) visualization decompilation, (2) data transformation synthesis, and (3) program generation.

synthesizer (step 2) to infer programs that can transform the input data to a table that contains the example table generated in step 1. Finally, for each candidate data transformation result, Falx generates a *candidate visualization* (step 3) by combining the transformed data with the visualization program synthesized in step 1 and compiling them to Vega-Lite or R scripts for rendering. Synthesized visualizations from all threads are collected and displayed in Falx's exploration panel for the analyst to inspect. In what follows, we elaborate on the details of each step using the same running example in Section 1.2.

Step1: Visualization Decompilation

Internally, Falx represents visualizations as a simplified visualization grammar similar to ggplot2 and Vega-Lite. In this grammar, a visualization is defined by (1) graphical marks (line, bar, rectangle, point, area), (2) encodings that map data fields to visual channels (x, y, size, color, shape, column, row), and (3) layers, which specify how basic charts are combined into compositional charts. Since Falx only uses this grammar as an intermediate language to capture visualization semantics, visualization details (e.g., scale types) are intentionally omitted. Falx goes through the following three steps to decompile a visualization.

- Falx first infers visualization layers from the user example. In particular, Falx partitions examples provided by the user into groups based on their geometric types and

properties, and creates one visualization layer for each group. Each layer corresponds to a simple chart of a particular type (e.g., scatter plot, line chart).

- Then, for each layer, Falx creates one basic visualization and an example table. The example table contains the same number of columns as the number of visual channels in this layer (derived from properties of geometric objects), and the visualization is specified as encodings that map columns in the example table to visual channels.

- Finally, for each example table, Falx fills the table with values from the example geometric objects.

Example 1. As shown in Figure 1.8-①, given the two visual elements provided by the user, Falx infers that the desired visualization should be a multi-layer chart that is composed by a line chart in layer 1 and a bar chart in layer 2 and decompiles the two layers independently. For example, for the second layer, Falx generates a bar chart program $\mathsf{Bar}\{x \mapsto \mathsf{C1}, y \mapsto \mathsf{C2}, y_2 \mapsto \mathsf{C3}, \mathit{color} \mapsto \mathsf{C4}\}$ with an example table $T = [(\mathit{2011\text{-}10\text{-}01}, 62.7, 63.4, 0.7)]$ where T represents the desired *output table* that should be the result of the data transformation process. Column names $\mathsf{C1}, ..., \mathsf{C4}$ in the bar chart program correspond to names of the four columns in Table T.

Step 2: Data Transformation Synthesis

After decompiling the examples into the visualization program and example tables T, together with the original input table T_{in} providied by the user, Falx reduces the visualization synthesis task into a data transformation synthesis task [58, 192, 194]. For each example table T, the data transformation synthesizer aims to synthesize a transformation program P_t that can transform the input table into a table that contains the example table, i.e., $T \subseteq P_t(T_{\mathsf{in}})$. Falx supports various types of transformation operators commonly used in the `tidyverse` library to handle different layouts of the input from the user (Figure 1.9).

The data transformation synthesizer uses an efficient algorithm to search for programs that are compositions of operators in Figure 1.9 satisfying the requirement $T \subseteq P_t(T_{\mathsf{in}})$. Falx starts the search process by constructing sketches of transformation programs (i.e., programs whose arguments are not filled) and then iteratively expands the search tree and fills arguments in these partial programs. To maintain efficiency in this combinatorial search process, Falx uses deduction to prune infeasible partial programs as early as possible (as used in prior work [58, 192, 194]). The deduction engine analyzes properties of partial programs using abstract interpretation [41] and prunes programs whose analysis results are inconsistent with the example output. Since each partial program corresponds to several dozens of concrete programs, the deduction engine can dramatically prune the search space.

Type	Operator	Description
Reshaping	`pivot_longer`	Pivot data from wide to long format
	`pivot_wider`	Pivot data from long to wide format
Filtering	`select`	Project the table on selected columns
	`filter`	Filter table rows with a predicate
Aggregation	`group`	Partition the table into groups based on values in selected columns
	`summarise`	For every group, aggregate values in a column with an aggregator
	`cumsum`	Calculate cumulative sum on a column for each group
Computation	`mutate`	Arithmetic computation on selected columns
	`separate`	String split on a column
	`unite`	Combine two string columns into one with string concatenation

Figure 1.9: Data transformation operators supported in Falx. For clarity, we omit the parameters of each operator.

When the search algorithm encounters a concrete program (i.e., with all arguments are filled) that is consistent with the example output, Falx adds the program to the candidate pool. The search procedure terminates either when the designated search space is exhaustively visited or when the given search time budget is reached. All synthesized program candidates are sent to the post-processor to generate visualizations.

Example 2. Figure 1.8-② shows the data transformation synthesis process for the second visualization layer (the bar chart) generated in step ①. Given the original input table I (with three columns Date, SF, and NY) the output table T (with four columns C1, C2, C3, and C4) generated in the last step, Falx aims to transform I into a table that contains the example table T. Starting from an empty program, Falx iteratively expands the unfilled arguments (represented as holes "□") in the partial programs to traverse the search space. When Falx encounters a partial program $\mathsf{cumsum}(I, \square)$, Falx abstractly analyzes it and concludes that it is infeasible because cumsum cannot transform an input table with three columns into an output table with four columns. Falx the expands the feasible partial programs (e.g., $\mathsf{mutate}(I, \square)$) and collects concrete programs that are consistent with the objective (e.g., $\mathsf{mutate}(I, \mathsf{Diff} = \mathsf{NY} - \mathsf{SF})$).

Optimization. We made several optimizations on top of existing synthesis algorithms [58, 194] to reduce Falx's response time. First, the major overhead in synthesis is the cost of analyzing partial programs using abstract interpretation, as it often requires running expensive operators like aggregation and pivoting on big tables. To reduce this overhead, Falx memoizes abstract interpretation results for partial programs to allow reusing then whenever possible.

Second, instead of aiming to find only on or a few candidate programs that match user inputs like prior algorithms, Falx expects to find as many different programs as possible that satisfy the examples to ensure the correct visualization is included. To ensure diverse outputs, different Falx solver threads start with different initial program sketches to search for different portions of the search space in parallel. To improve responsiveness, Falx sets different timeouts for different threads to allow faster threads to respond to the user while other threads are searching for more complex transformations. In our implementation, we run 2 solver threads in parallel, we set one thread with 5 seconds timeout and another with 20 seconds timeout based on our perception of how long an analyst would be willing to wait as well as the typical time Falx takes to finish traversing different parts of the search space.

Step 3: Processing Synthesized Visualizations

As the final step in visualization synthesis, Falx generates visualizations by combining the visualization program generated in step 1 with table transformation programs generated in step 2.

Concretely, for each data transformation program, Falx applies the table transformation program on the input data to obtain a transformed output and unifies the output table schema with the schema in the visualization program, since the visualization program was filled with placeholder column names C1, C2, ..., etc. Falx then instantiates other visualization details (e.g., scale type, axis domain, etc.) omitted in the visualization grammar and compiles the visualization program into a Vega-Lite (or R) script through syntax-directed translation. For example, in Figure 1.8-③, Falx generates an R script that both transforms the input and specifies the visualization. Furthermore, Falx notices that the values on the x-axis are dates instead of strings, so it changes the x-axis scale to a temporal scale using the function "`scale_x_date()`".

After compilation, the post-processor removes semantically duplicate visualizations (i.e., visualizations with different specifications but with the same content and detail). Finally, Falx groups and ranks the visualizations based on the complexity of the programs (numbers of expressions). In this way, similar visualizations are grouped together to make comparison easier in the exploration process, and the complexity ranking allows users to explore visualizations constructed from easier transformation programs first before jumping into complex ones. These visualizations are sent to the user interface for rendering to allow user exploration.

1.4 User Study

To understand Falx's benefits and limitations and to examine how analysts might adopt synthesis-based visualization tools, we conduct a between-subjects evaluation centered on the following questions:

- Does Falx improve user efficiency in creating visualizations compared to a baseline tool?
- How does Falx change the visualization authoring process for different data analysts?
- What strategies are used by data analysts to visualize data with Falx?

1.4.1 Participants

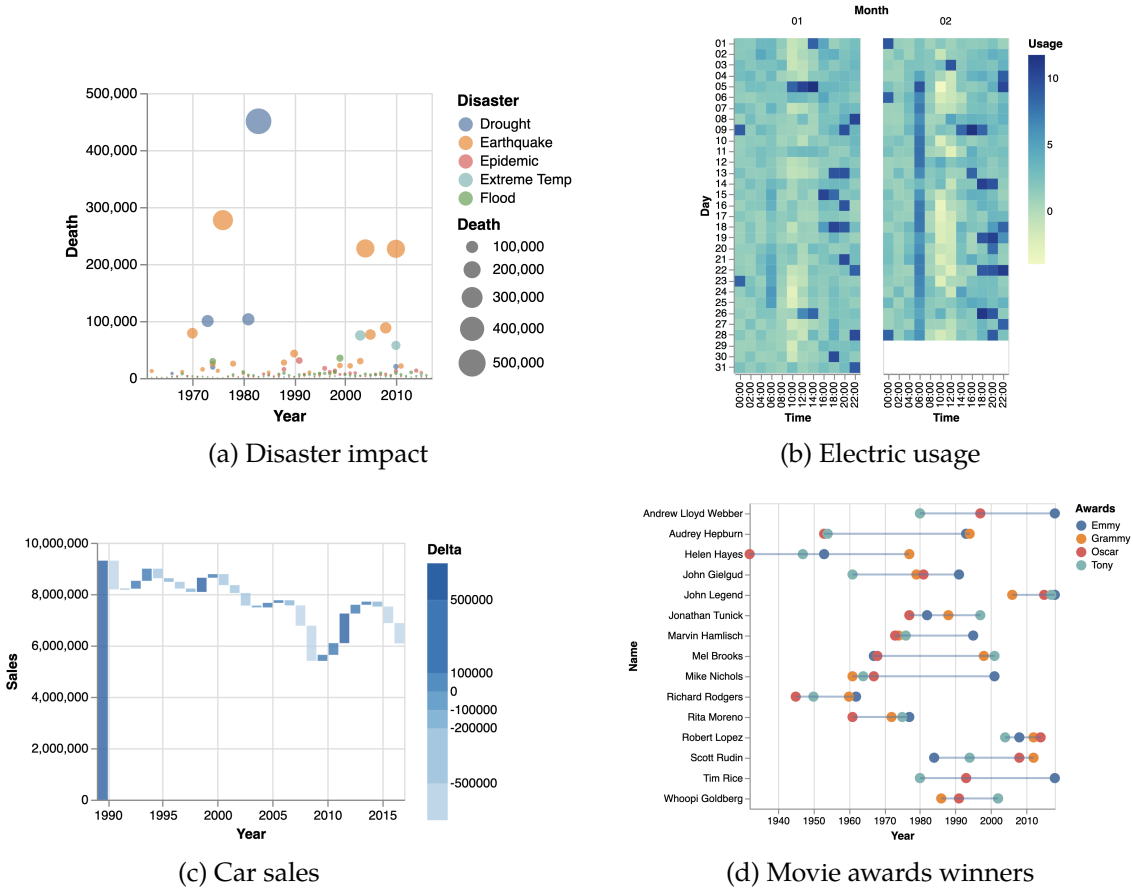
We recruited two groups participants for the study: 16 participants (10 M, 5 F, 1 Unknown, Ages 23-51) for the Falx study, and another 17 participants (12 M, 4 F, Ages 19-60) for the baseline tool study (the R programming language). In the recruiting process, we screened participants by their ability to read a sample visualization. For the baseline group, we additionally required that all participants have experience with R (specifically ggplot2 and tidyverse libraries) for data visualization.

Participants reported their experience in data visualization authoring based on the number of visualizations they created in the past 6 months using any tools. For the Falx study group, there were 6 participants experienced with some visualization tools (created >10 visualizations), 8 with moderate experience with visualization tools (created 1-10 visualizations), and 2 participants with zero experience in creating visualizations in the past. For the baseline group, there were 8 experienced participants (create >10 visualizations) and 9 participants with moderate experience (created 1-10 visualizations).

1.4.2 Procedure

Each participant was asked to complete four visualization tasks, where the Falx study group completed the task using Falx and the baseline group used R to complete the task. We chose R as the baseline tool due to its popularity among data analysts and its ability to support both data transformations and visualizations in the same context (many other visualization tools requires users to process data and specify visualizations in different contexts).

To better examine the use of Falx, participants in the Falx group first completed a 20-minute tutorial together with a warm-up task with a sample solution (creating a grouped line chart to visualize sea ice level change in the past 20 years). After the tutorial, participants were asked to solve four visualization tasks. For R participants, we also provided the same warm-up task with a sample solution to allow users to get familiar with the environment and



(a) Disaster impact

(b) Electric usage

(c) Car sales

(d) Movie awards winners

Figure 1.10: Study Tasks

the data loading process, so that participants could focus on solving the visualization tasks. During the user study, participants were allowed to refer to any resource on the Internet including documentations and QA forums. We collected screen and audio recordings while participants completed tasks. We then interviewed them after all tasks were completed to reflect on their visualization process and strategies.

In the user study, we developed four visualization scenarios (Figure 1.10):

(a) *Disaster Impact*: A scatter plot that visualizes the number of people died from five disasters in the last century.

(b) *Electric Usage*: A faceted heat map for hourly electric usage in each day during the first two months of 2019.

(c) *Car Sales*: A waterfall chart for the number of cars sold in a year. Each bar starts at the sales value in the previous month and ends at the sales values in the month, and its color gradient reflects the increase/decrease compared to the last month.

(d) *Movie Awards*: A layered line/scatter plot for visualizing winners of all four prestigious movie awards. For each celebrity, there are four points showing years these awards were earned and a line showing the time span for the celebrity to win all four awards.

For each visualization task, we provided as input a table that can be directly imported into the tool. We also explicitly described visualization designs to the participants in text so that participants could focus on implementation. Finally, we asked participants that they do not need to optimize the design — a task was considered correctly solved as long as the semantics of the visualization created by the participant matched the example solution regardless of the process and details. In this study, we did not restrict the time participants could spend on each task, but we provided users the option of quitting a task after spending more than 20 minutes without success. Thus, participants could complete each task with one of three outcomes: (1) submit a correct solution, (2) submit a wrong solution, or (3) give up after trying for at least 20 minutes.

We interviewed each participant after they finished all four tasks. For both Falx and baseline groups, we interviewed participants about (1) challenges they encountered while solving the tasks and their solutions, (2) common errors they made and how they fixed them, (3) their confidence about the solutions they submitted and what checks they performed to ensure correctness, and (4) what additional resources they used during the study and how they helped. We additionally asked participants in the Falx group to reflect on their visualization authoring process and interviewed them about (1) strategies adopted when creating examples to demonstrate the visualization task, (2) strategies adopted to explore the synthesized visualizations, and (3) their prior visualization experience and how Falx could potentially fit in their routine work.

The total session was less than 2 hours for all participants. To address learning effects or other carryover effects, we counterbalanced the tasks using a Latin square. We performed our analysis using mixed effect models, treating participants as a random effect and modeling tool, tasks, and experience level as fixed effects.

1.4.3 Quantitative Results

Figure 1.11 shows the percentage of participants that correctly finished each task. Falx participants generally had higher completion rates in all tasks. We observed a statistically significant difference in the completion rate in the car sales visualization ($p < 0.05$); others

were not significant. Among nine failed tasks by Falx users, seven were due to incorrect solutions and, in two cases, participants quit the task after 20 minutes. Among 20 failed cases in the R study group, there were 9 incorrect solutions ans 11 cases where participants quit after 20 minutes.

Task	R ($N = 17$)		Falx ($N = 16$)	
	n	%	n	%
Disaster Impact	16	94.1%	14	87.5%
Electric Usage	13	75.6%	14	87.5%
Car Sales	5	29.4%	11	68.8%
Movie Awards	14	82.4%	16	100%

Figure 1.11: The number and percentage of participants correctly finished each study task.

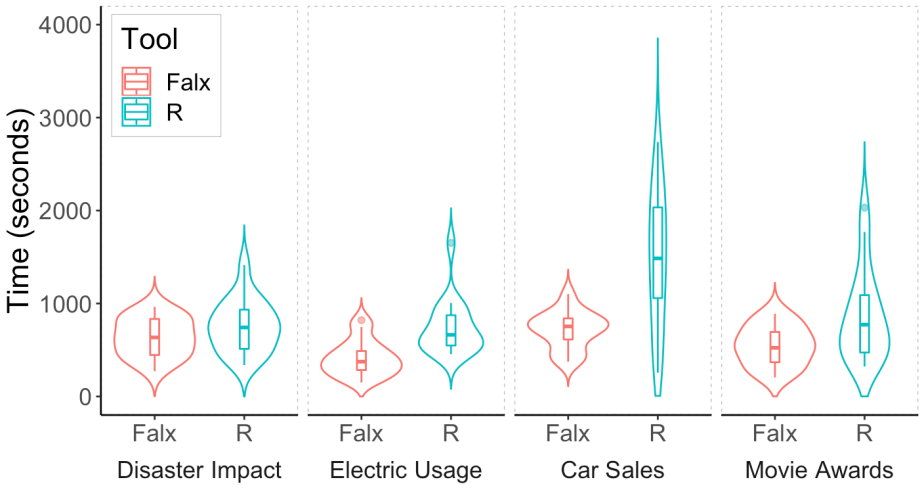


Figure 1.12: Violin plot showing the amount of time participants spent on each task for both Falx and R study groups.

Figure 1.12 shows task completion time in Falx. Using Wilcoxon rank sum test with Holm's sequential Bonferroni procedure for p value correction, we observed a significant improvement in user efficiency for car sales visualization ($t_{\text{Falx}} = 715 \pm 202s, t_{\text{R}} = 1473 \pm 743s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 758s$, $p < 0.01$) [1] and electric usage visualization ($t_{\text{Falx}} = 411 \pm 192s$, $t_{\text{R}} = 740 \pm 297s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 329s, p < 0.001$). While Falx participants were also generally faster

[1] We use t_{Falx} and t_{R} to show the mean and standard deviation of time participants in Falx and R groups spent on each task. We use $\mu_{\text{R}} - \mu_{\text{Falx}}$ to represent the difference of the mean time between the two groups.

in the other two tasks, there was no significant difference for the movie industry celebrity visualization ($t_{\text{Falx}} = 544 \pm 215s$, $t_{\text{R}} = 861 \pm 490s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 323s, p = 0.07$) or the disaster impact visualization ($t_{\text{Falx}} = 638 \pm 209s$, $M_{\text{R}} = 754 \pm 279s$, $\mu_{\text{R}} - \mu_{\text{Falx}} = 116s, p = 0.23$). Participants from the R study group noted that the key reasons for failing on the car sales visualization task was the difficulty of finding the correct API (for waterfall chart) together with the complex transformation behind it (which required calculating a cumulative sum). Falx users also noted they found the car sales visualization difficult due to unfamiliarity with the visualization type. On the other hand, R users reported that the movie awards visualization and the disasters impact visualization were relatively easier since they expected the same pivot operator to transform the input, which is commonly encountered by R users, and the visualization types were relatively standard (line chart and scatter plot).

We found no significant interaction between user experience level (defined in Section 1.4.1) and task completion time ($p = 1$ for all tasks in both study groups using Wilcoxon rank sum test with Holm's sequential Bonferroni correction).

1.4.4 Qualitative Feedback

In this section, we describe qualitative feedback from participants in both groups about general (non-Falx related) visualization challenges both during the study and in their daily work, and how Falx can help with solving some of these challenges. We leave discussions of Falx-specific visualization challenges to Section 1.4.5.

As described in Section 1.4.2, we conducted a semi-structured interview for participants of both groups about visualization challenges they encountered both in the study and in their daily work, and how some of these challenges are typically overcome. To analyze this data, two of the researchers collaboratively conducted a qualitative inductive content analysis on the interviewer's notes, with a sensitizing concept of *visualization challenges and solutions*. In this process, two researchers independently labeled interview notes and then collaboratively discussed and compared high level labels to resolve disagreements in the initial codes.

Finding the right visualization function

The first challenge frequently mentioned by participants was discovering or recalling the correct visualization function. In the R study group, 14 out of 17 participants described this challenge, especially for the car sales task that most participants failed on. Some participants noted that the difficulty came from both finding the right term to search and distinguishing similar candidate functions. For example, participant R14 [2] noted that *"I wasn't aware that*

[2]We use R1-R17 to denote participants from the R study group and F1-F16 to denote participants from the Falx group.

geom_rect() would be more helpful than geom_bar(). One thing that made it more challenging was the fact that this kind of bar chart has no proper name. I tried searching 'non-contiguous bar charts in R', but I didn't get many useful results.". These challenges are also common in compositional charts: R10 noted *"creating the line with the dots is something I never did before so didn't know how to achieve it"*. To address these challenges, participants noted that online example galleries and forums are *"essential to their work"* (R1). Besides, two participants had *"an internal file – R code dictionary"* (R7) and *"a collection of some own code snippets"* (R1) to reduce search effort.

Falx group participants also described that they faced similar challenges of finding right functions in their daily work and Falx could help address them. For example, F1 mentioned: *"Falx can generate something that you cannot easily do. For example, the multi-layered visualization for the movie dataset would be very difficult to do in Excel or Google doc, you may need to specify some formula to specify relationship between two layers."* Participant F11 mentioned that Falx helped with complex tasks because *"It allows you to start by creating a relatively simple visualization in the beginning, which is good, then it allows you to build more complex stuff on top of it which is also helpful."*

Data transformation

Data transformation was another frequently mentioned challenge, including both conceptualizing the expected data layout and implementing the transformation. For example, R17 mentioned *"it [the car sales task] also seems to require some extra aggregation to get the starting and ending value for each rectangle to be drawn, which makes it even more difficult."*. About implementation, R9 said that *"the vocabulary of the tidyverse is critical for trying to do what you want to do, otherwise it is all impossible to achieve."*, and R14 mentioned that *"I had an idea of what I needed to do, but I wasn't able to search the right things on Google to arrive at a useful code snippet for it."*.

Participants from the Falx group mentioned similar issues in their work routine. For example, *"Tableau won't do data preparation and you need to manually put them together"* (F7), *"pivoting table is already something at an intermediate level in Tableau and many people cannot use it"* (F2). Due to lack of skill of preparing data programmatically, some participants would do it manually. For example, *"if I need to pivot data, I do it manually – e.g., just copy the data to a blank area [in Excel] and pivot it"* (F8). Participants appreciated that Falx automatically handled data transformations. Participant F5 mentioned *"I like the fact that it [Falx] solves the data transformation and visual encoding. I'm pretty familiar with visual encoding so it is fine when the data is in the right shape. But I find transforming data annoying."*. Participant F15 mentioned *"I didn't think about data format at all in the process"*. F7 mentioned *"Tableau won't do data preparation because you need to manually put them together and drag drop them for you. Falx is pretty automated on this."*.

Learning to create expressive visualizations

Due to the inherent challenge in visualization and data transformation in these tools, participants mentioned many of existing tools had a learning barrier for new users. For example, F4 mentioned that *"the learning curve is pretty steep (Tableau), and we spent a lot of time learning these tools"*.

On the other hand, while Falx was a new visualization tool, most users found it easy to learn, despite some users requiring some time in the beginning to get used to *"the paradigm shift from my normal understanding"* (F6). For example, participant F4 mentioned that *"the ramp up time [for Falx] is pretty short and it's pretty easy to use."*, and F6 mentioned that *"anyone with basic Excel knowledge should be able to use Falx"*.

1.4.5 The visualization process in Falx

Since Falx is a new tool for data visualization, besides understanding its ability to address existing visualization challenges, we also investigated how participants used Falx to solve visualization tasks. We conducted an inductive content analysis on the interviewer's notes about Falx experience similar to that in Section 1.4.4. In this section, we discuss observations about participants' visualization process in Falx and their indications for future synthesizer-based visualization tool design.

Strategies for creating examples

Data analysts initiate interactions with Falx by creating examples. As a synthesis-powered visualization tool, poorly constructed examples can be highly ambiguous and lead to long running time and a large number of visualization candidates. Also, while users can carefully create multiple examples to increase Falx's performance, it requires more efforts. Falx users identified the following strategies to create examples effectively:

- *Sketching visualizations before demonstration:* Three participants mentioned that sketching the visualization design on paper helped them understand geometry of the visualization, and it helped them creating better examples. For example, participant F13 mentioned *"I sketch out first to get a general understanding of what the visualization would look like, and then use that to drop points."*.

- *Selecting representative data points to demonstrate:* Seven participants mentioned that they considered using *"representative points"* (F7) when creating demonstrations in order to reduce ambiguity to Falx. For example, participant F1 mentioned that *"[In the disaster impact task], I chose a cause that contains non-zero value in that year, because it's a unique value that can avoid confusion of the tool"*.

- *Start from a few examples, add more later if necessary:* Eight participants mentioned that they *"tried to shoot for minimum input"* (F6) for simplicity. In this way, they can *"run the tool to see what it returns"* (F1) before spending more effort on examples, and they would "add more to help narrow it down if there are many visualizations pop up" (F9). Additionally, participant F11 noted that *"It's easy to add multiple elements to mess up with the demonstration. A small number of elements make it easier to go back and fix"*.

- *Start with multiple examples to minimize interaction iterations:* Instead of starting from minimal inputs, 6 participants preferred to create more examples in the beginning to *"avoid ambiguity"* (F2). They remarked that *"it doesn't take that much time to add data points"* (P8) and multiple examples can *"avoid having to wait and choosing from multiple solutions"* (F8).

During the process of creating and revising examples, seven participants found the demo preview panel useful since it allowed them to *" understand more about how a certain layout would look like"* (F11) and it *"helps put me on the right track of solving the task."* (F13). However, nine participants said they did not find it helpful because they *"don't know if it tells enough to help understand anything [about synthesis results]"* (F7); they preferred to *"just click synthesis to get the result since synthesis is pretty fast"* (F14).

Some challenges participants encountered in creating examples included (1) unfamiliarity with terms in Falx (e.g., F4 mentioned *"'size' is a term that I'm not familiar with."*) and (2) not getting used to demonstrate visualization ideas using values (e.g., F6 mentioned *''I was struggling with the paradigm shift about when to use values and when to use table headers"*). In general, the fast response time of Falx enabled participants to get over these challenges through trial and error (e.g., F1 mentioned *"If there is anything wrong, I'll go back and do edits on the points."*), and they *"get faster in later tasks once understand the difference"* (F6). In future, Falx could adopt a mixed-initiative interface [90] to improve experience for new users. In addition, we observed that many participants felt like they were interacting with an intelligent tool (e.g., F13 mentioned *"the tool is quite good at learning from what I demonstrated"*) and they were willing to provide more informative inputs (e.g., F16 *"tried to write the expression because I don't know how Falx would do computation"*). In future, Falx could take advantage of this to support more complex visualization tasks by synthesizing programs from users more informative inputs besides examples (e.g., formulas that describe how certain values in the examples are derived from the input).

Strategies for exploring synthesis results

After creating examples to demonstrate the visualization task, users interact with Falx to explore the synthesized visualizations and identify the desired solution. Prior work [99, 119]

has shown that a main barrier for adoption of synthesis-based programming tools is that users have difficulty understanding and trusting synthesized solutions, especially when there are many solutions consistent with the user demonstration.

We discovered from the interview that many participants shared the following similar 4-step process to select the desired visualization from synthesized visualizations by investigating visualization from coarse to fine:

- *Step 1: Check against the high-level picture.* First, participants noted that it was easy to quickly exclude many visualizations that are obviously far from the desired visualization. For example, *"having too many options is a bit overwhelming, but just keeping in mind what the result you look like can help narrow down the solution"* (F11).

- *Step 2: Check axes and invariants.* After excluding the obviously wrong solutions, participants often investigate domains and ranges of each axis to further refine synthesis results. For example, *"I first looked at color labels, I noticed they tend to be wrong in wrong visualizations – e.g., some charts only contain 2 labels instead of 4"* (F16).

- *Step 3: Compare similar visualizations.* Then, participants investigated similar visualizations to find their difference. For example, *"In the electric case, there is one mistake [in a candidate visualization] with 2019 showing up on y axis, it's small and not obvious. But then, I was able to tell the difference by comparing the two visualizations directly, and notice that year showed up in the 'hour' field"* (F2).

- *Step 4: Inspect visualization detail.* Finally, participants *"check carefully about the values to make sure they are correct"* (F5). An example of such detailed checking is to check values in the chart against known values in the input data: *"if there is a specific value that I know is correct – for example, in the last example (disasters), I knew the total death for 1961 was, then I hover over the output to check if the value is correct "* (F6).

After these steps, participants were confident about the result. In fact, while participants mentioned that their confidence about solutions could be negatively affected by unfamiliarity of visualization types (e.g., F9 mentioned *" I don't do much heatmap so I'm less confident"*), they mentioned that the checking process can raise their confidence about the chosen solution. For example, participants got more confident after *"comparing them [candidates] with my sketch"* (F6), *"looking at solutions and finding their difference"* (F14), or *"checking details"* (F2). They further noted that in many cases, *"it's almost impossible for Falx to get it wrong because these values are all pretty unique"* (F14). In general, participants found the exploration panel *"quite useful"* because it *"allows to choose the best visualization out of that"* (F7).

In sum, Falx's exploration panel allowed users to directly inspect solutions in the visualization space following a coarse-to-fine process, which helped them to disambiguate solutions and trust the chosen results. In future, Falx's interface could be improved to augment users' exploring strategies. For example, Falx could directly summarize the differences among the synthesized visualizations to allow users to make comparisons easier. Also, Falx's center view panel could support displaying traces that show how properties of each geometric object are derived from the input, which could make the synthesis process more transparent and make checking details easier.

1.4.6 Fitting into Data Analysts' Workflows

Finally, participants reflected on how Falx might fit into their workflow. For example, F13 mentioned *"I'll absolutely use this if this is a product. Even as it is now I'll use it"*. Participants found several scenarios that Falx can be helpful.

- Create visualizations for discussions and presentations. For example, F1 noted that *"visualizations generated by Falx can meet standards of presentation slides"* and *"Falx can generate something that you cannot easily do in Excel"*.
- Prototyping complex analysis. For example, F16 mentioned *"Falx is very useful in the prototyping stage because it's very fast to use."* F7 further noted that they can *"take a sample to visualize and then extend to the full visualization"* using Falx for analyzing big datasets.
- Benefit non-experienced users. Six participants mentioned that Falx can be *"more beneficial to new users that cannot create charts"* (F2). Also, Falx can be *"a good teaching tool to help people understand data"* (F7).
- Reduce team collaboration effort. Participant F11 described that visualization readers were often different from visualization creators in their team, and modifying visualization required team efforts. F11 mentioned that Falx could help with it: *"a person presents me with a visualization, but I want to view something differently. Instead of getting back to the person to re-do it, I can probably just use Falx to do it, which would be more efficient"*.

However, several participants also mentioned Falx may not fit well to their current workflow when they need *"very high standard visualizations"* (F1) that requires extensive customization. Another limitation of the current version of Falx is the lack of *"deep integration with other tools"* (F1), e.g., database for handling big datasets and data cleaning tools for *"handling null / dirty data"* (F4). But in general, participants thought that Falx would be helpful when used in the right scenarios and *"would be pretty interesting to try Falx in some of these tasks"* (F5).

1.5 Related Work

Falx builds on top of prior research on grammar based visualization tools, data transformation tools, program synthesis algorithms and automated visualization design systems.

Grammar-based Visualization. Following the initial publication of the Grammar of Graphics [206], high level grammars [154, 203, 174] for data visualizations have grown increasingly popular as a way of succinctly specifying visualization designs. In contrast to low level visualization languages like Protovis [18], D3 [78], and Vega [155] that are designed for creating highly-customizable explanatory visualizations, these high level grammars aim to enable analysts to rapidly construct expressive graphics in exploratory analysis. For example, ggplot2 [202, 203] and Vega-Lite [154] are two visualization grammars that allow users to specify visualizations using visual encodings. In both tools, low level visualization details are handled by default parameters unless users want customization. Tableau [174] adopts a graphical interface approach to enable users to rapidly create views to explore multidimensional database. In Tableau, users drag-and-drop data variables onto visual encoding "shelves", which are later translated into a high-level grammar similar to ggplot2. These tools expect the input data layout to match the design such that (1) each row corresponds to a graphical object, and (2) each column can be mapped to a visual channel. In practice, the mismatch between the design and the input data layout is common, which raises a barrier for creating visualizations [67, 207].

Falx formalizes visualizations in the same way, and synthesized programs are compiled to ggplot2 or Vega-Lite for rendering. Falx's user interface also inherits the expressiveness and simplicity of Grammar of Graphics design, by allowing users to create examples of visual encodings to demonstrate visualization ideas. The main difference is that Falx relaxes the constraints on input data layout and allows users to use layout-independent examples to demonstrate visualization ideas. Falx then automatically infers the visualization spec and synthesizes data transformations to match the data with the design from the examples, which saves users' construction efforts.

Data Transformation Tools. The need to prepare data for statistical analysis and visualization has led to the development of many tools for data transformation [141, 205, 90, 52]. Since different analysis objective requires different layout, users need to frequently transform data throughout the analysis process [205, 89, 207]. Potter's Wheel [141] is a graphical interface that allows users to interactively choose transformation operators and inspect transformation outputs. Wrangler [90] is a mixed initiative data transformation tool which can suggest transformations based on the input data. Tidyverse [205] is a data transformation library in R, which allows users to interleave data transformation code, analysis code and visualization code in the same environment to reduce the effort of context switch. Several

synthesis-powered data transformation tools [192, 58, 52, 140, 8] have been proposed to help automate data transformation. For example, Prose [140] includes several programming-by-example tools that automatically synthesize programs for data cleaning and transformation from input-output examples. Morpheus [58] and Scythe [192] are two specialized data transformation synthesizer with better scalability and expressiveness.

Falx inherits the transformation language design in tidyverse [205], and Falx is a realization of prior program synthesis algorithms [58, 194] as an interactive system for visualization authoring. Falx's main difference from automated data transformation tools is the unification of the visualization task and transformation tasks. In this way, Falx users do not need to conceptualize expected data layout or frequently switch between visualization and data transformation tools. The unification also enables Falx users to easily explore synthesis results in the visualization space as opposed to program space, which is considered challenging [119]. Besides data layout transformation, many data preparation tools also support data cleaning (e.g., handling missing data or invalid data) [196], data normalization (collecting non-relational data into relation format) [8], and string formatting [52, 71, 219]. Falx currently does not support directly visualizing dirty or non-relational data. In future, Falx could work with these tools to further automate visualization process.

Visualization Automation. Automated visualization design tools [125, 82, 151] have been proposed to help data analysts to explore the visualization design space. Draco [125] and Dziban [107] use constraint logic approaches to model design knowledge, and they can recommend visualization designs from partial specifications. VizNet [82] uses a deep neural network trained from visualization corpus to suggest designs. Voyager [209] combines recommendation and exploration for mixed-initiative design exploration. VisExemplar [151] allows users to demonstrate changes in the visualization layout to explore alternative visualizations designs. Falx is complementary to these design automation tools. Falx allows users to implement visualization designs they have in mind without data layout constraints, while design automation tools helps users to explore visualization designs from a fixed data layout. A combination of the two approaches could potentially help users to explore a larger visualizations design space without data layout constraints.

User Interaction with Program Synthesizers. In general, program synthesizers can be categorized into exploration tools and implementation tools. Synthesis-based exploration tools aim to generate a large number of solutions from users' weak constraints to aid users to explore the search space [177, 125]. For example, Scout [177] is a synthesis-based exploration tool to discover mobile layout ideas. In these tools, users interact with an exploration interface to navigate and save interesting solutions. Implementation tools [192, 58, 71, 140, 219, 52], instead, aim to synthesize programs to help solve a concrete task (e.g., implement a design

user already have in mind). In these tools, the main interaction objective is to help users to disambiguate spurious programs that happen to be consistent with the user specification but are incorrect for the full task [119]. To solve this challenge, Wrex [52] generates readable programs for users to inspect and edit; Regae [219] and FlashProg [119] interactively ask users disambiguating questions to refine synthesis results; PUMICE [106] lets users collaborate with the agent to recursively resolve any ambiguities or vagueness through conversations and demonstrations.

Falx is an implementation tool for data visualization. Falx's contribution to the user interaction model is that Falx brings the exploration design (from exploration tools) to address the disambiguation and trust challenges in implementation tools. Allowing users to explore and examine synthesized programs in the visualization space reduces the barrier for user interaction (e.g., users do not need to be familiar with underlying programs to disambiguate [119]) and increases users' confidence about solutions.

Expressive Visualization Design Tools. Besides tools for standard visualization authoring, many visualization tools have been proposed to enable designers to create more expressive visualizations. Examples of these tools are Data illustrator [108], Lyra [152], Charticular [146], Data-driven Guides [96], and StructGraphics [183]. Besides high-level design layout (e.g., x,y ,column) and standard mark properties (e.g., color, shape), these tools allow users to customize marks to create more expressive glyphs (e.g., compound marks, parametric marks). These tools expect users to prepare data into a tidy format to start with, but they support rich visualization designs. Falx, in comparison, supports standard visualization designs but automates data transformation.

Several design reconstruction tools (e.g., VbD [151], Liger [150], iVolVER [122]) are proposed to let designers create expressive visualization by destructing and reconstructing existing visualization designs. Using these tools, users can transform existing visualizations to new ones by demonstrating desired design changes. Functionally, these tools are design exploration tools that take as input a visualization design and produce a new visualization design. They differ from Falx because Falx takes data as input and maps it to a visualization design for initial design authoring.

There are opportunities to combine Falx with these tools for better visualization authoring. Falx can work with expressive designs tools to support new designs from non-tidy data: users can first design customized marks using example data values, and the tool would automatically synthesize binding between data and these fine-grained mark properties from these examples. Falx can also work with design reconstruction tools to allow users to first use Falx to create initial design from data, and then subsequently interactively explore new designs by transforming the initial design.

1.6 Discussion & Future Work

We have presented Falx, a novel synthesis-powered visualization authoring tool that allows users to demonstrate a visualization design using examples of visual encodings and then receive suggestions for visualization designs. Our goal was to create a system that does not require users to manually specify the visualization or worry about data transformations, thereby improving user efficiency and reducing the learning burden on novice analysts. Our study found that Falx often achieved these goals: Falx users were able to effectively adopt Falx to solve visualization tasks that they could otherwise cannot solve, and in some cases, they do so more quickly. We next discuss some implications of this work in guiding future research.

Data Layout Flexible Visualization Exploration. Besides visualization authoring, combining Falx with data exploration tools like GraphScape [97] Voyager [209], GraphScape [97], VbD [151] and Dziban [107] can potentially bring out new design exploration tools that allow users to discover both new relations from the dataset and new designs to visualize the them. In current design exploration tools, given an input data and an initial design, the tool will automatically suggest diverse visualization designs for input data, but new designs are specific to the fixed data layout. Falx can work together with these design to make design exploration no longer constrained by data layout. For example, in an anchored design exploration scenario [107, 97, 151], users can demonstrate data layout changes alongside design changes using the new tool to incrementally discover data insights from a larger design space. Similarly, Falx might also work with visualization recommendation engines [125, 82] to find better designs for the dataset based on users initial examples.

Visualization Learning. As we discovered from our study, users often describe existing programming tools as "flexible, powerful" but "having a steep learning curve". Falx can fill in this gap by helping data analysts to learn to create visualizations. Since Falx does not require its users to have programming expertise, new users can use Falx to create visualizations and learn and learn visualization and data transformation concepts by inspecting programs generated by Falx. For example, Falx could generate readable code like Wrex [52] for users to learn to program.

Bootstrapping Complex Data Analysis. While Falx currently focuses on non-experienced data analysts, it could also benefit experienced data analysts by bootstrapping complex data analysis tools, where analysts could first create visualizations in Falx and then build complex analysis on top of synthesized programs. To achieve this goal, Falx need more transparency and better integration with programming environments. For example, Falx can expose synthesized programs during the synthesis process and allow users to steer the synthesis

process to better disambiguate results. Falx can also be integrated into programming environments like mage [93], Wrex [52] or Sketch-n-Sketch [80] to make programming editing easy.

User-Synthesizer Interaction by Exploration. Allowing users to effectively disambiguate synthesis results and trust the selected result has always been a challenge for the interaction between users and program synthesizers [119, 99]. Falx takes advantages of the visualization domain and transforms this problem into a visualization exploration problem. Similar interaction models could be applied to improve user-synthesizer interaction experience in other domains where exploration can be effectively achieved. For example, Android wire-frame synthesizers [14] use the same model to compare different implementations of the same design; 3D model synthesizers [53] could also allow users to directly inspect 3D programs to check whether they correctly generates user demonstrations.

All of these possibilities suggest a promising future for augmenting design work with synthesis-based techniques. We hope future work can build on Falx and its overarching design, identifying ever more robust interface paradigms and architectures to empower human work.

Chapter 2

The Visualization by Example Algorithm

As mentioned in the last chapter, creating visualizations from a complex dataset is far from an easy task despite they play a crucial role in data analysis. In particular, besides understanding the functionality provided by existing visualization libraries, generating the desired visualization also requires reshaping and aggregating the underlying data as well as composing different visual elements to achieve the intended visual narrative.

This chapter introduces the synthesis algorithm behind the visualization-by-example system we described in Chapter 1: our approach aims to simplify visualization tasks by automatically synthesizing the required program from simple *visual sketches* provided by the user. Specifically, given an input data set and a visual sketch that demonstrates how to visualize a very small subset of this data, our technique automatically generates a program that can be used to visualize the entire data set.

From a program synthesis perspective, automating visualization tasks poses several challenges that are not addressed by prior techniques. First, because many visualization tasks require data wrangling in addition to generating plots from a given table, we need to decompose the end-to-end synthesis task into two separate sub-problems. Second, because the intermediate specification that results from the decomposition is necessarily imprecise, this makes the data wrangling task particularly challenging in our context. In this chapter, we address these problems by developing a new *compositional* visualization-by-example technique that (a) decomposes the end-to-end task into two different synthesis problems over different DSLs and (b) leverages bi-directional program analysis to deal with the complexity that arises from having an imprecise intermediate specification.

We implemented our synthesis algorithm in a tool called Viser and evaluate it on 83 visualization tasks collected from on-line forums and tutorials. Viser can solve 84% of these benchmarks within a 600 second time limit, and, for those tasks that can be solved by Viser, the desired visualization is among the top-5 in 70% of the cases.

2.1 Introduction

Visualizations play an important role in today's data-centric world for discovering, validating, and communicating insights from data. Due to the prevalence of non-trivial visualization

tasks across different application domains, recent years have seen a growing number of libraries that aim to facilitate complex visualization tasks. For instance, there are at least a dozen different visualization libraries for Python and R, and more than ten different visualization libraries for JavaScript have emerged in the past year alone [1]. In addition, there has also been a flurry of research activity around building programming systems like D3 [19] and Vega-Lite [154] to further facilitate real-world visualization tasks.

Despite all these recent efforts, data visualization still remains a challenging task that requires considerable expertise — in fact, so much so that some companies even have job titles like "data visualization expert" or "data visualization specialist." Generally speaking, there are three key reasons that make data visualization a challenging task. First, beyond having a good insight about how the data can be best visualized, one needs to have sufficient knowledge about how to use the relevant visualization libraries. Second, different visualization primitives typically require the data to be in different formats; so, in order to experiment with different types of visualizations, one needs to constantly reshape the data into different formats. Finally, generating the intended visualization typically requires modifications to the original dataset, including aggregating and mutating values and adding new columns to the input tables, and doing so often require deep knowledge in data manipulation.

We propose a new technique, coined *visualization-by-example*, for automating data visualization tasks using program synthesis. In our proposed approach, the user starts by providing a so-called *visual sketch*, which is a partial visualization of the input data for just a few input points. Given the original data set T_{in} and a visual sketch S provided by the user, our technique can synthesize one or more visualization scripts whose output is consistent with S for the input data set. These visualization scripts can then be applied to T_{in} to generate several visualizations of the *entire data set*, and the user can choose the desired visualization among the ones that are generated.

Despite these appealing aspects of our approach to end-users, the data visualization problem presents unique challenges from a program synthesis perspective. First, as hinted earlier, data visualization tasks almost always involve two distinct steps, namely (1) data wrangling (reshaping, aggregating, adding new columns etc.) and (2) invoking the appropriate visualization primitives on the transformed data. Asking users to manually decompose the problem into these two individual steps would defeat the point, as the user would have to at least understand which visualization primitives to use and what format they require. Thus, it is imperative to have a compositional technique that can *automatically decompose* the end-to-end task into two separate synthesis problems.

---

[1] https://financesonline.com/data-visualization/

One of the key contributions of this paper is to show how to automatically decompose an end-to-end visualization task into two separate synthesis problems over two different languages. Specifically, given an input data source T_{in} and a visual sketch S, our goal is to learn a *table transformation program* P_{T} and a *visual program* P_{V} such that executing $\mathsf{P}_{\mathsf{V}} \circ \mathsf{P}_{\mathsf{T}}$ on T_{in} yields a visualization that is consistent with the provided visual sketch S. In order to solve this problem in a compositional way, our method infers an intermediate specification ϕ that constrains the output (resp. input) of the target program P_{T} (resp. P_{V}). This intermediate specification is in the form of *table inclusion constraints* $\mathsf{T} \overset{\diamond}{\subseteq} t$ specifying that input t of the visual program must include all tuples in T but it can also contain additional rows and columns. As we demonstrate experimentally, having this intermediate specification is crucial for the scalability of our approach.

A second key contribution of this paper is a new algorithm for synthesizing table transformation programs. While there has been recent work on automating table transformation tasks using programming-by-example [192, 58], these techniques focus on the case where the specification is a pair of input and output tables. In contrast, the intermediate specification in our setting is a set of table inclusion constraints rather than a concrete output table, and pruning strategies used in prior work are not effective in this setting due to lack of precise information about the output table. To deal with this challenge, we introduce a new table transformation synthesis algorithm that uses lightweight bidirectional program analysis to prune the search space. As we demonstrate experimentally, this new table transformation algorithm results in much faster synthesis compared to prior work [58] for automating visualization tasks.

We have implemented the proposed "visualization-by-example" approach in a new tool called Viser and evaluated it on 83 visualization tasks collected from on-line forums and tutorials. Our experiments show that Viser can solve 84% of these benchmarks and, among those benchmarks that can be solved, the desired visualization is among the first 5 outputs generated by Viser in 70% of the cases. Furthermore, given that it takes on average of 11 seconds to generate the top-5 visualizations, we believe that Viser is fast enough to be beneficial to prospective users in practice.

To summarize, this paper makes the following key contributions:

- We introduce the *visualization-by-example* problem and present an algorithm for synthesizing visualization scripts given the original data set and a small visual sketch.

- We show how to decompose the synthesis task into two sub-problems by inferring an intermediate specification in the form of table inclusion constraints.

- We propose a new algorithm for synthesizing table transformations from table inclusion specifications. Our algorithm leverages lightweight bidirectional program analysis to effectively prune the search space.

- We evaluate our approach on over 80 tasks collected from on-line forums and tutorials and show that Viser can solve 84% of the benchmarks, and that the desired visualization is among the top-5 results in 70% of the solved cases.

2.2 Overview

In this section, we give an overview of our approach with the aid of a simple motivating example depicted in Figure 2.1. In this example, the user has two tables T_1 and T_2 that record the results of a scientific experiment. Specifically, Table T_1 stores an experiment identifier (`ID`), a so-called "experiment condition" (`Cond`), and the experiment result, which consists of an `A` value as well as an `Aneg` value. An additional table T_2 stores the gender of the participant in the corresponding study: That is, for each experiment (`ID`), the `Gender` column in T_2 indicates whether the participant is male (M) or female (F). The user wants to visualize the result of this experiment by drawing a scatter plot that shows how the sum of `A` and `Aneg` changes with respect to `Cond` for each of the two genders. In particular, the top right part of Figure 2.1 illustrates the desired visualization. In the remainder of this section, we explain how our approach synthesizes the desired visualization script in R using the `tidyverse` package collection, which includes both visualization primitives (e.g., provided by `ggplot2`) and data wrangling capabilities (e.g., provided by `tidyr` and `dplyr`).

User input. In order to use our visualization tool, Viser, the user needs to provide the data source (i.e., tables T_1 and T_2) as well as a visual sketch S, which is a partial visualization for a tiny subset of the original data source. In this case, since the user wants to draw a scatter plot, the visual sketch is very simple and consists of a few data points, as shown in the "Input" portion of Figure 2.1. Specifically, the visual sketch contains two points, one at $(1, 7)$ and the other at $(2, 6)$, and both points have label "`M`". In general, one can think of a visual sketch as a set of *visualization elements* (e.g., point, bar, line, ...) where each visualization element has various attributes, such as coordinates, color, etc. In particular, we can specify the visual sketch shown in Figure 2.1 as the following set of visual elements:

$$\{\mathsf{point}(v_x = 1, v_y = 7, v_{color} = \texttt{M}), \mathsf{point}(v_x = 2, v_y = 6, v_{color} = \texttt{M})\}$$

We refer to this alternative set-based representation of a visualization as a *visual trace*. In general, we can represent both visual sketches as well as complete visualizations in terms of

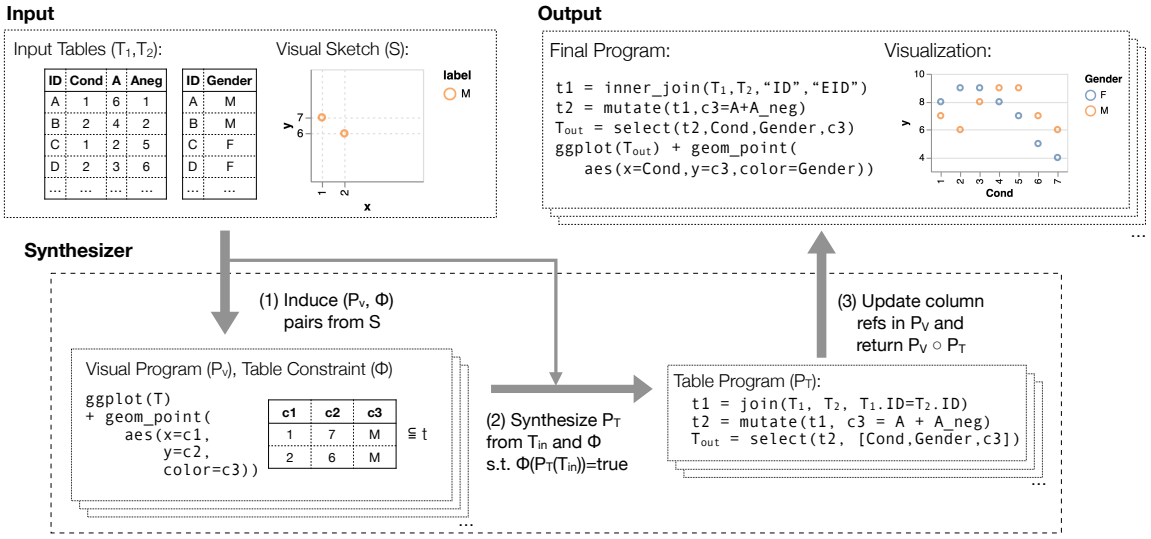


Figure 2.1: Overview of our synthesis algorithm: the system takes as input an input table T_{in} and a visual sketch S, and returns a list of candidate visualizations satisfying the inputs.

their corresponding visual trace; thus, we use the term "visual trace" interchangeably with both "visualization" and "visual sketch".

Synthesis problem and approach. Given the input data source I and a visual sketch S, our synthesis problem is to infer a *visualization script* P such that P(I) yields a visual trace that is *a superset* of S. As mentioned in Section 2.1, a visualization script consists of a pair of programs P_V ("visual program") and P_T ("table transformation program"), for plotting and data wrangling respectively. Since P_V and P_T do conceptually different things and are expressed in separate languages, we decompose the overall synthesis task into two sub-tasks, namely that of synthesizing a visual program P_V and separately synthesizing a table transformation program P_T.

Synthesis of visual programs. To achieve the decomposition outlined above, our synthesis algorithm first infers a *set* of visual programs that are *capable* of generating a visualization consistent with S. This inference step is based purely on the visual sketch and does not consider the input data (i.e., tables T_1, T_2). For our running example, there are multiple visual programs (expressible using `ggplot2`) that can generate the desired result; we show three of these programs in Figure 2.2. All three programs start with the code "`ggplot(T) + geom_point(...)`", which indicates that the resulting visualization is a scatter plot drawn from table T. However, the three visual programs differ in the following ways:

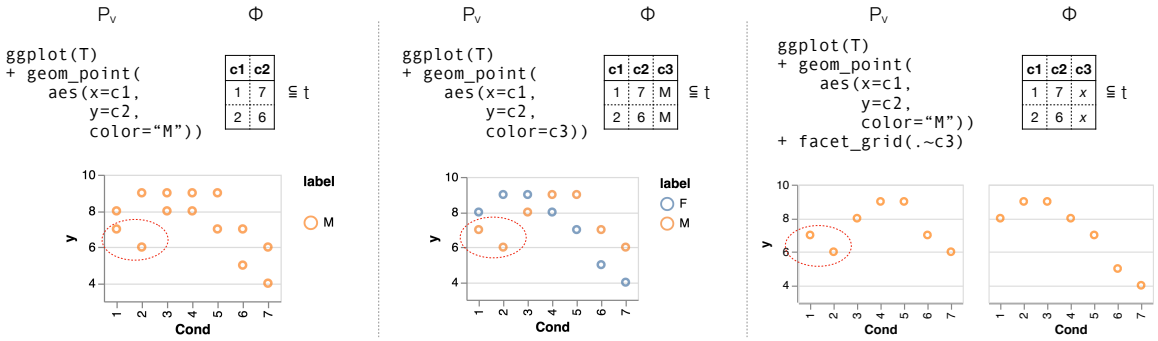


Figure 2.2: Sample visual programs and their corresponding intermediate specification for the visual sketch from Figure 2.1. The bottom part of each figure shows the corresponding visualization if the visual program is adopted; observe that all of these visualizations are consistent with the visual sketch.

- All points generated by the first visual program have the same color (indicated as `color="M"`)
- For the second visual program, the color of the points is determined by the corresponding value of column `c3` in table `T` (indicated as `color=c3`)
- The visualization generated by the last program contains multiple subplots determined by c_3. That is, the visualization is partitioned into a list of subplots according to different values in column c_3 of the input table.

As indicated in the bottom part of Figure 2.2, the visualizations generated by all three programs are consistent with the visual sketch in that they contain the two data points specified by the user.

Intermediate specification inference. Next, given the visual sketch S and a candidate visual program $\mathsf{P_V}$, our synthesis algorithm infers an intermediate specification ϕ that constrains the input that $\mathsf{P_V}$ operates on. Furthermore, ϕ has the property that executing $\mathsf{P_V}$ on any concrete table consistent with $\mathsf{T_{in}}$ yields a visual trace that is a superset of S. Going back to our running example, Figure 2.2 shows the intermediate specifications inferred for each of the three visual programs. These intermediate specifications are of the form $\mathsf{T} \overset{\diamond}{\subseteq} t$ indicating that the input t of the visual program must contain table T but can also include additional rows and columns. Looking at the intermediate specifications from Figure 2.2, we can make the following observations:

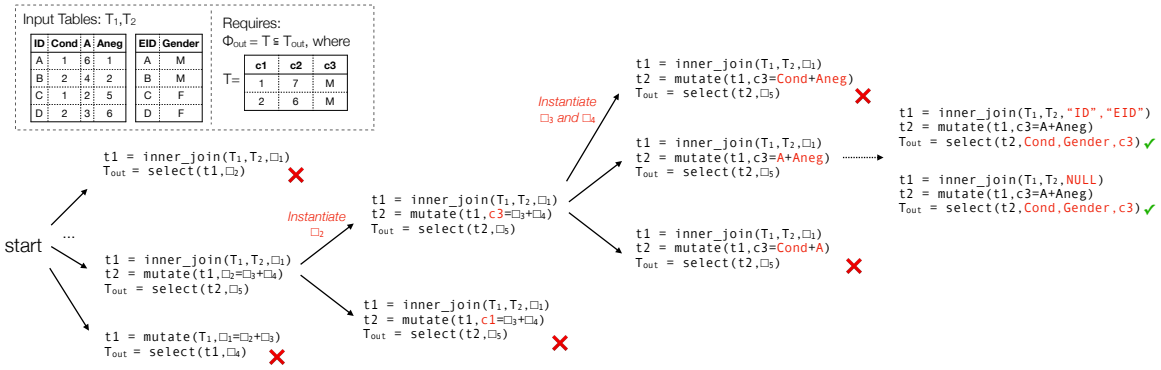


Figure 2.3: The synthesis process for P_T using T_1, T_2 and ϕ_{out} in Figure 2.1. At each step, the synthesis algorithm first picks a known variable and expands it (new values expanded at each step are labeled in red), then it evaluates each program sketch using abstract semantics of the table transformation language and prune it if the evaluation process results in conflicts.

- The inputs of the first visual programs must contain at least two columns (referred to as `c1`, `c2`) and these columns should contain the values 1, 2 and 7, 6 respectively. However, the input table can also contain additional attributes and values.

- The specification for the second visual program imposes one additional constraint over the other ones. In particular, the input table must contain an additional third column (referred to as `c3`), and this column must contain at least two occurrences of value `M`.

- The specification for the third visual program requires that the table should contain an additional column `c3` to specify which subplot each point belongs to. Since the visual sketch contains two points in the same subplot, column `c3` contains two duplicate values with the same subplot identifier.

Input for the second synthesis task. As mentioned earlier, the key reason for inferring an intermediate specification is to decompose the problem into two separate synthesis tasks. Thus, given an initial data source T_{in} and intermediate specification ϕ, the goal of the second synthesis task is to generate a table transformation program P_T such that applying P_T to T_{in} yields a table that is consistent with ϕ. To illustrate how our method synthesizes the desired table transformation program for our running example, let us consider the following data wrangling constructs:

- **Projection:** The construct `select`$(t, \bar{c})$ computes the projection of table t onto columns $\bar{c}$.
- **Join:** The construct `inner_join`(t_1, t_2, p) computes the product of tables t_1, t_2 and then filters the result based on predicate p.
- **Mutation:** The construct `mutate`$(t, c_{target}, c_{arg_1} + c_{arg_2})$ takes as input a table t and returns a table with a new column called c_{target}, where the values in c_{target} are obtained by summing up the two columns c_{arg_1} and c_{arg_2}; [2]

We will now illustrate how to synthesize the desired table transformation program $\mathsf{P_T}$ for the input tables $\mathsf{T}_1, \mathsf{T}_2$ shown in Figure 2.1 and the intermediate specification ϕ_{out} shown in the second column of Figure 2.3.

Table transformation synthesis overview. Viser employs an enumerative search algorithm to find a table transformation program that satisfies the specification. Similar to prior program synthesis techniques [58, 192, 57], Viser uses lightweight deductive reasoning to prune invalid programs during the search process. However, because the specification does not involve a concrete output table, pruning techniques used in prior work (e.g., [58]) are not effective in this context. As mentioned in Section 2.1, our algorithm addresses this issue by leveraging lightweight bidirectional program analysis and an (also lightweight) incomplete inference procedure over table inclusion constraints.

As illustrated schematically in Figure 2.3, Viser performs enumerative search over *program sketches*, where each program sketch is a sequence of statements of the form `v = op(`$\square_1, \ldots, \square_n$`)` where `op` is one of the data wrangling constructs (e.g., `select`, `inner_join` etc.) and $\square_i$ denotes an unknown argument. Since program sketches contain only table-level operators but not their arguments, the sketch enumeration process is tractable, and the main synthesis burden lies in searching the large number of parameters that each hole $\square_i$ can be instantiated with.

Sketch completion. Given a program sketch, Viser's sketch completion procedure alternates between *hole instantiation* and *pruning* steps until a solution is found or all possible sketch completions are proven *not* to satisfy the specification (see Figure 2.3). The first step (i.e., hole instantiation) is standard and makes the initial sketch iteratively more concrete by filling each hole with a program variable or constant. The pruning step, on the other hand, is more interesting, and infers table inclusion constraints of the form $e_1 \overset{\diamond}{\subseteq} e_2$ indicating that the table represented by expression e_1 can be obtained from the table represented by expression

[2] General `mutate` operator supports arbitrary column-wise computation besides '+', we only consider `mutate` with '+' in overview for simplicity.

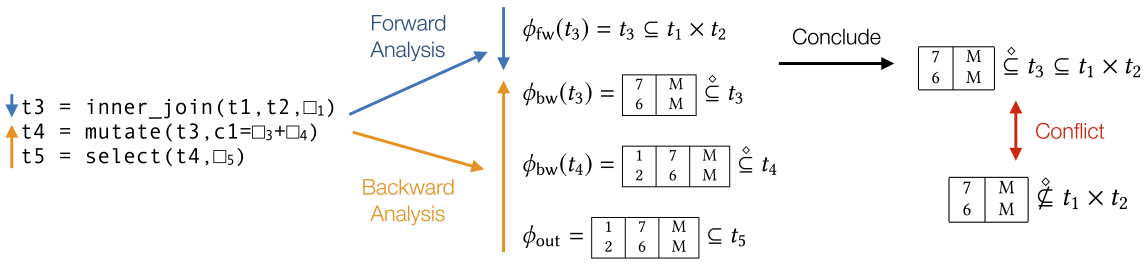


Figure 2.4: Demonstration of how Viser prunes invalid abstract programs in Figure 2.3 using forward and backward analysis. ϕ_{out} is the requirement from the synthesis task, and $\phi_{\text{fw}}(t), \phi_{\text{bw}}(t)$ refers to abstractions of t derived from forward / backward analysis.

e_2 by removing rows and/or columns. For the forward (resp. backward) inference, the generated constraints have the shape $t \mathrel{\hat{\subseteq}} \mathsf{T}$ (resp. $\mathsf{T} \mathrel{\hat{\subseteq}} t$), where t is a program variable and T is a concrete table. Thus, using a combination of forward and backward reasoning, we can obtain "inequality" constraints of the form $\mathsf{T}_1 \mathrel{\hat{\subseteq}} t \mathrel{\hat{\subseteq}} \mathsf{T}_2$ for each program variable t and use this information to reject a *partially completely sketch* whenever T_1 *cannot* be obtained from T_2 by deleting rows and/or columns.

Going back to our running example, let us consider the partially completed sketch shown in Figure 2.4, where t_1, t_2 represent the program's arguments and t_5 is the return variable. By considering the semantics of each construct, we can make the following deductions:

1. Since the program's output must conform to $\phi_{\text{out}} = (\mathsf{T} \mathrel{\hat{\subseteq}} t)$ (where T is the table from Figure 2.2) and we have $t = t_5$, we can generate the constraint $\mathsf{T} \mathrel{\hat{\subseteq}} t_5$ on variable t_5.

2. Together with the above constraint and semantics of `select`, we obtain $\mathsf{T} \mathrel{\hat{\subseteq}} t_4$.

3. Since we have $\mathsf{T} \mathrel{\hat{\subseteq}} t_4$ and `mutate(t3, c1, ? + ?)` generates t_4 by adding column `c1` to t_3, we can deduce all columns of T except for the first one should also be in t_3. Thus, using backwards reasoning, we obtain the constraint $\mathsf{T}' \mathrel{\hat{\subseteq}} t_3$ where T' is the table shown on the left-hand side of $\phi_{\text{bw}}(t_3)$.

4. Since t_3 is the result of `inner_join(t1, t2, ?)`, we cannot deduce something useful about t_1, t_2 in the backward direction without introducing expensive case splits because we do not whether each value in T' comes from t_1 or t_2. However, going in the *forward* direction, we can generate the constraint $t_3 \mathrel{\hat{\subseteq}} T_1 \times T_2$, where T_1, T_2 are the input tables from Figure 2.1.

Putting together the information obtained from the forwards and backwards analysis, we obtain the constraint $\mathsf{T}' \overset{\diamond}{\subseteq} t_3 \overset{\diamond}{\subseteq} \mathsf{T}_1 \times \mathsf{T}_2$. However, this creates a contradiction with $\mathsf{T}' \overset{\diamond}{\not\subseteq} \mathsf{T}_1 \times \mathsf{T}_2$, allowing us to prune the partially completed sketch from Figure 2.4.

Synthesis output. Continuing in this manner and alternating between more hole instantiation and pruning steps, Viser finds the sketch completion shown on the bottom right side of Figure 2.1. The final output of the synthesizer is shown on the top right of the same figure and generates the intended visualization, also shown in the top right of Figure 2.1.

2.3 Problem Definition

In this section, we formally define the visualization-by-example problem and then introduce two languages for automating table transformation and plotting tasks.

2.3.1 Key Concepts and Synthesis Problem

Tables. For the purposes of this paper, a table T with schema $[c_1, \ldots, c_n]$ is an unordered bag (i.e., multi-set) of tuples where each tuple $r = (v_1, \ldots, v_n)$ in T consists of n primitive values (number, string, datetime etc.). Given a tuple $r \in \mathsf{T}$, we use the notation $r[c]$ to denote the value v stored in attribute c of r. We also extend this notation to tables and write $T[\bar{c}]$ to denote the projection of table T on columns $\bar{c}$ and write $T[-\bar{c}]$ to denote the projection of table T on all columns *except* $\bar{c}$. Finally, we write $\mathsf{Mult}(r, \mathsf{T})$ to denote the multiplicity of tuple r in T.

Given a pair of tables $\mathsf{T}_1, \mathsf{T}_2$, we write $\mathsf{T}_1 \subseteq \mathsf{T}_2$ iff $\forall r \in \mathsf{T}_1.\ \mathsf{Mult}(r, \mathsf{T}_1) \leq \mathsf{Mult}(r, \mathsf{T}_2)$. As standard, we define equality to be containment in both directions, i.e., $\mathsf{T}_1 = \mathsf{T}_2$ iff $\mathsf{T}_1 \subseteq \mathsf{T}_2$ and $\mathsf{T}_2 \subseteq \mathsf{T}_1$. We further define a table inclusion constraint $\mathsf{T}_1 \overset{\diamond}{\subseteq} \mathsf{T}_2$ that allows projecting columns in addition to filtering rows. Specifically, we write $\mathsf{T}_1 \overset{\diamond}{\subseteq} \mathsf{T}_2$ iff there exists columns $\bar{c}$ in the schema of T_2 such that $\mathsf{T}_1 \subseteq \mathsf{T}_2[\bar{c}]$.

$$
\begin{array}{rcl}
\tau & = & \{e_1, \ldots, e_n\} \\
e & = & \mathsf{bar}(a_x, a_{y_1}, a_{y_2}, a_{color}, a_{subplot}) \\
 & | & \mathsf{point}(a_x, a_y, a_{color}, a_{size}, a_{subplot}) \\
 & | & \mathsf{line}(a_{x_1}, a_{y_1}, a_{x_2}, a_{y_2}, a_{color}, a_{subplot})
\end{array}
$$

Figure 2.5: The visualization trace language $\mathcal{L}_\tau$, where metavariable a refers to constants.

Visual traces. As stated in Section 2.2, we define the semantics of visualizations in terms of so-called *visual traces*. A visual trace, denoted τ, is a set of basic visual elements (point,

line, bar), together with the attributes of each element (position, size, color, etc.). More concretely, Figure 2.5 shows a small "language" in which we express visual traces. (The full language supported by our implementation is given in the Appendix.) Here, e denotes a visual element, and a is an *attribute* of that element:

- *Color attribute:* This attribute, denoted a_{color} specifies the color of a visual element.
- *Position attributes:* Position attributes, such as a_x, a_{x_1}, a_{y_2} etc., specify the canvas positions for a visual element. For example, for line, (a_{x_1}, a_{y_1}) specifies the starting point of a line segment, and (a_{x_2}, a_{y_2}) specifies the end point. For the bar visual element, a_{y_1}, a_{y_2} specify the start and end y-coordinates of a (vertical) bar.
- *Size attribute:* The attribute a_{size} specifies the size of a given point element.
- *Subplot attribute:* The attribute $a_{subplot}$ specifies the subplot that a given visual element belongs to. For instance, for the visualization shown in the last column of Figure 2.2, the points in the first plot have a different $a_{subplot}$ attribute than those in the second one.

In the remainder of this paper, we express both complete visualizations and visual sketches in terms of their corresponding visual trace, and we often use the symbol S to denote traces that correspond to visual sketches. Finally, since visual traces are *sets* of visual elements, the notation $\tau_1 \subseteq \tau_2$ indicates that visualization τ_2 is an *extension* of visualization τ_1.

Problem statement. Given this notion of visual traces, we can now state our *visualization-by-example problem*, which is defined by a pair $(\mathsf{T}_{\mathsf{in}}, \mathsf{S})$. Here, T_{in} is a table [3] and S is a visual trace (i.e., a "program" in the language of Figure 2.5). Now, let us fix a language $\mathcal{L}_{\mathsf{T}}$ for expressing table transformation programs and a visualization language $\mathcal{L}_{\mathsf{V}}$ for generating plots from a given table. Then, our goal is to synthesize a pair of programs $(\mathsf{P}_{\mathsf{T}}, \mathsf{P}_{\mathsf{V}})$ such that:

1. P_{T} and P_{V} are programs written in $\mathcal{L}_{\mathsf{T}}, \mathcal{L}_{\mathsf{V}}$ respectively,
2. the output visual trace $\mathsf{P}_{\mathsf{T}}(\mathsf{P}_{\mathsf{V}}(\mathsf{T}_{\mathsf{in}}))$ is consistent with S, i.e., $\mathsf{S} \subseteq \mathsf{P}_{\mathsf{V}}(\mathsf{P}_{\mathsf{T}}(\mathsf{T}_{\mathsf{in}}))$.

Note that our problem statement strictly generalizes conventional programming-by-example which requires the program output to be equal to the provided output (i.e.,

[3] We note that our implementation can handle multiple tables in the input. However, we consider a single input table in the formal development to simplify presentation and reduce notational overhead.

$$
\begin{array}{rcl}
\mathsf{P_T}(t) & = & t_1 = e_1; \ldots; \mathsf{T_{out}} = e_n; \\
e & = & T \mid \mathsf{filter}(T, f) \\
& \mid & \mathsf{select}(T, \bar{c}) \mid \mathsf{join}(T_1, T_2, f) \\
& \mid & \mathsf{mutate}(T, c_{target}, op, \bar{c}_{arg}) \\
& \mid & \mathsf{gather}(T, \bar{c}_{id}, \bar{c}_{target}) \\
& \mid & \mathsf{spread}(T, \bar{c}_{id}, c_{key}, c_{val}) \\
& \mid & \mathsf{summarize}(T, \bar{c}_{key}, \alpha, c_{target}) \\
T & = & \mathsf{T_{in}} \mid t \\
f & = & v_1 \ op \ v_2 \mid \mathsf{is\_null}(c) \\
v & = & \mathsf{const} \mid c \\
\alpha & = & \mathsf{min} \mid \mathsf{max} \\
& \mid & \mathsf{sum} \mid \mathsf{count} \mid \mathsf{avg}
\end{array}
$$

Figure 2.6: The table transformation language $\mathcal{L}_\mathsf{T}$, where $\mathsf{T_{in}}, \mathsf{T_{out}}$ refers to the input/output tables, t refers to table variables, and c refers to column names.

$\mathsf{S} = \mathsf{P_V}(\mathsf{P_T}(\mathsf{T_{in}}))$. Thus, the user is still free to provide a small (but complete) input-output example if this is more convenient for the user. However, our generalization has the advantage of freeing the user from the burden of modifying the input data in cases where doing so may be inconvenient.

2.3.2 Table Transformation Language

Our table transformation language is shown in Figure 2.6. This language is inspired by existing data wrangling libraries (e.g., tidyr and dplyr libraries for R), and similar languages have also been used in prior work for automating table transformation tasks using PBE [58, 117]. As shown in Figure 2.6, a table transformation program $\mathsf{P_T}$ is a sequence of side-effect free statements, where each statement produces a new table by performing some operation on its inputs. As standard in relational algebra, the constructs select and filter are used for selecting columns and rows respectively. As also standard in relational algebra, join is used for taking the cross product of two tables. That is, $\mathsf{join}(T_1, T_2, f)$ is semantically equivalent to $\mathsf{filter}(T_1 \times T_2, f)$, where $\times$ denotes the standard cross product operator in relational algebra.

Besides these standard relational algebra operators, our table transformation language contains four other constructs: spread, mutate, gather, and summarize. Since the semantics of these constructs are somewhat non-trivial, we illustrate their behavior in Figure 2.7.

1. The `mutate` construct creates a new column (c_{target}) in the output table by applying an operator *op* on argument columns $\bar{c}_{args}$. For example, in Figure 2.7, the new column c' is obtained by summing up columns c_2 and c_3.

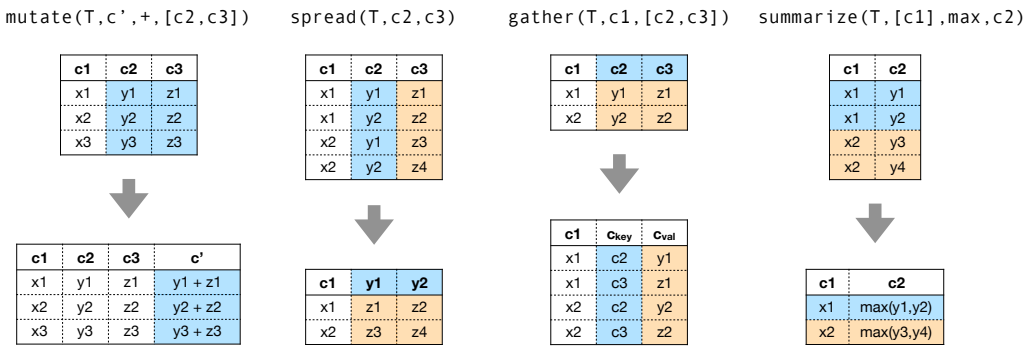


Figure 2.7: Examples of table transformation operators in $\mathcal{L}_{\mathsf{T}}$ and how they operate on example input tables. Colored cells shows how parts of the output table are computed from the input table.

2. The `spread` operator pivots a table by changing values to column names. Specifically, `spread` first eliminates the two columns c_{key} and c_{val}, then creates a new column for each value stored in the original column c_{key}, and finally fills new columns using values in the original c_{val} column. The second drawing in Figure 2.7 illustrates the semantics of `spread`.

3. The `gather` construct is the inverse of `spread`: It unpivots the input table by moving column names into the table body. Specifically, `gather` first eliminates all columns in $\bar{c}_{target}$, and then creates two new columns c_{key} and c_{val} where c_{key} is filled with column names in $\bar{c}_{target}$ and column c_{val} is filled with values in the eliminated columns. This is illustrated in the third drawing in Figure 2.7.

4. The `summarize` construct first partitions its input table into groups based on values in $\bar{c}_{key}$ and then applies the function α to each group to aggregate values in column c_{target}. For example, in the rightmost part of Figure 2.7, the table is partitioned into two groups based on values in c_1 (labeled with different colors), and column c_2 is populated by taking the maximum of all values in the corresponding partition.

2.3.3 Visualization Language

Our visualization language $\mathcal{L}_{\mathsf{V}}$ is shown in Figure 2.8, which formalizes core constructs in Vega-Lite [154], the `ggplot2` visualization library for R and VizQL [76] from Tableau. This formalization enables concise descriptions of visualizations by encoding data as properties of graphical marks. It represents single plots using a set of mappings that map data fields to

$$
\begin{array}{rcll}
\mathsf{P_V} & = & \mathsf{MultiPlot}(SP, c_{\mathsf{sub}}) \mid SP & \\
SP & = & \mathsf{MultiLayer}(\bar{L}) \mid L & \\
L & = & \mathsf{Scatter}(c_x, c_y, c_{color}, c_{size}) & \text{(Scatter Plot)} \\
 & \mid & \mathsf{Line}(c_x, c_y, c_{color}) & \text{(Line Chart)} \\
 & \mid & \mathsf{Bar}(c_x, c_y, c_{y_2}, c_{color}) & \text{(Bar Chart)} \\
 & \mid & \mathsf{Stacked}(c_x, c_h, c_{color}) & \text{(Stacked Bar Chart)} \\
c & = & column \mid \epsilon &
\end{array}
$$

Figure 2.8: The visualization language $\mathcal{L}_{\mathsf{V}}$.

visual properties, and supports combining single charts into compositional charts through layering and subplotting. A program $\mathsf{P_V}$ in this language takes as input a table T and outputs a visual trace τ. Throughout this paper, we refer to programs in this language as *visual programs*.

As shown in Figure 2.8, a visual program $\mathsf{P_V}$ either creates a grid of multiple plots using the MultiPlot construct or a single plot SP. Each plot can in turn consist of multiple layers (indicated by the MultiLayer construct) or a single layer. Each layer is either a scatter plot (Scatter), a line chart (Line), a bar chart (Bar), or a stacked bar chart (Stacked). The MultiLayer construct in this language is used to compose *different* kinds of charts in the same plot (e.g., a scatter plot and a line chart), but our visualization language is nonetheless rich enough to allow layering the same type of chart within a plot: For example, the Line primitive can be used to render multiple line charts where each individual line chart has a different color.

In terms of its semantics, a visual program $\mathsf{P_V}$ specifies how each tuple in the input table corresponds to a visual element in the output trace; thus, all constructs in $\mathcal{L}_{\mathsf{V}}$ refer to column names in the input table. For instance, for the MultiPlot construct, the column name c_{sub} specifies that tuples sharing the same value of c_{sub} are to be visualized in the same subplot, whereas tuples with different values of c_{sub} belong to two different subplots. Similarly, for the Line construct, tuples that agree on the value of c_{color} are rendered as part of the same line chart, whereas tuples that disagree on the c_{color} value correspond to different layers.

In what follows, we explain the semantics of our visualization language with the aid of the examples shown in Figure 2.9.

Example 3. The first example in Figure 2.9 shows a visual program for rendering a stacked bar chart visualization of a study report. This program specifies using `x=Task` that each different (stacked) bar on the x-axis should correspond to a different task (Q1, Q2 etc.) from the input table. The second argument, `y=Percent`, specifies that the height of each bar (within a stack) is determined by the `Percent` column in the input table. Finally, the

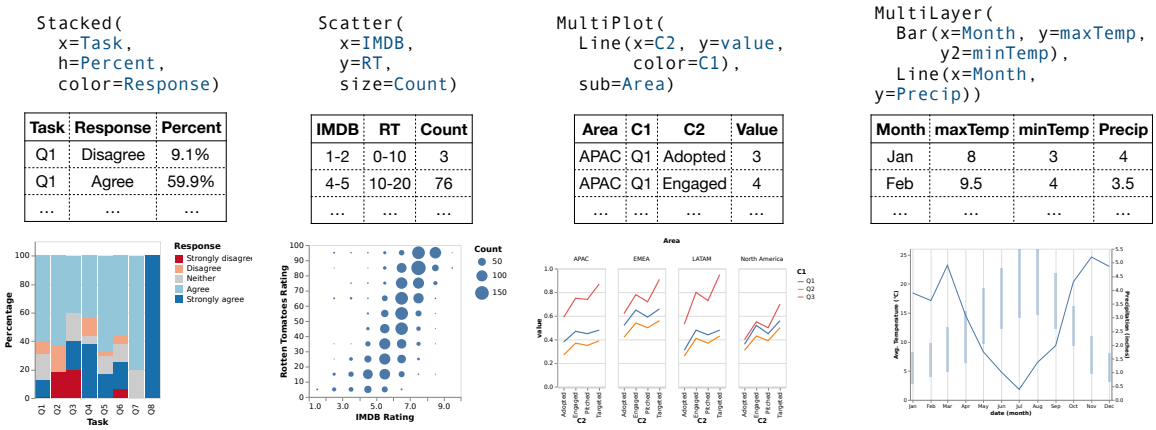


Figure 2.9: Examples of visualization operators in $\mathcal{L}_V$ and their corresponding visualizations.

third argument, `color=Response`, specifies that the color of each bar (within the stack) is determined by the value stored in the `Response` column of the input table.

Example 4. The second visual program in Figure 2.9 renders a scatter plot that visualizes the correlation between IMDB and Rotten Tomato reviews. Specifically, the `Scatter` construct draws a point for each row in the input table. In our example, the first (resp. second) argument specifies that the x (resp. y) coordinate is determined by the value in the `IMDB` (resp. `RT`) column of the input table. Finally, the third argument `size=Count` specifies that the size of the point is determined by the corresponding value stored in the `Count` column.

Example 5. The third program from Figure 2.9 renders multiple subplots, as specified by the `MultiPlot` construct. The second argument `subplot=Area` specifies that each subplot corresponds to a separate value in the `Area` column of the input table (APAC, North America etc.). The first argument, on the other hand, specifies that each subplot is a line chart. Furthermore, since the third argument of the `Line` construct is `color=C1`, each subplot consists of multiple line charts of different colors, determined by the value of the `C1` column in the input table. Finally, the (x, y) coordinates of the points within each line chart are determined by the values in `C2` and `Value` columns respectively.

Example 6. The last program in Figure 2.9 draws a layered chart consisting of a bar chart and a line chart using the MultiLayer construct. Here, bars show the temperature range for each month because the `x` value corresponds to the `Month` field in the input table, and `y` and `y2` correspond to the `maxTemp` and `minTemp` values for that month. On the other hand, the line chart shows the precipitation for each month; this is again specified using `x=Month` and `y=Precip`.

Algorithm 1 Top-level Synthesis Algorithm

```
1:  procedure Synthesize(T_in, S)
2:      input: Input table T_in, visual sketch S
3:      output: A table transformation program P_T and visual program P_V, or ⊥ if failure
4:      Ω ← LearnVisualProgs(S)
5:      for all (P_V, ψ) ∈ Ω do
6:          r ← LearnTableTransform(T_in, ψ)
7:          match r
8:              case ⊥: continue
9:              case (P_T, σ): return (P_T, P_V[σ])
10:     return ⊥
```

2.4 Synthesis Algorithm

In this section, we first give an overview of our top-level synthesis algorithm (Section 2.4.1) and then present techniques for learning visual programs (Section 2.4.2) and table transformation programs (Section 2.4.3) respectively.

2.4.1 Overview

Algorithm 1 describes our top-level visualization-by-example algorithm, which takes as input a table T_{in} and a visual sketch S (expressed as a visual trace in the notation of Figure 2.5) and returns a pair of programs $(\mathsf{P}_\mathsf{T}, \mathsf{P}_\mathsf{V})$ such that $\mathsf{S} \subseteq \mathsf{P}_\mathsf{V}(\mathsf{P}_\mathsf{T}(\mathsf{T}_{\mathsf{in}}))$ (or $\perp$ to indicate failure). As mentioned previously, our synthesis algorithm is compositional in the sense that it uses an intermediate specification to guide the search for table transformation programs.

Internally, the Synthesize procedure first uses the input visual sketch S to infer a *set* Ω of intermediate synthesis results. Each intermediate result $r \in \Omega$ is a pair $(\mathsf{P}_\mathsf{V}, \psi)$, where P_V is a visual program that is consistent with the provided visual sketch and ψ is a constraint that imposes certain requirements on the input to P_V. In other words, for a given visual program P_V, ψ serves as an intermediate specification that constrains the space of possible table transformation programs. However, since we do not know the column names used in this intermediate table, both P_V and ψ refer to "made-up" column names to be resolved in the next phase.

In the second phase of the algorithm (lines 5-9), we try to find a table transformation program that satisfies the intermediate specification. Specifically, for each intermediate synthesis result $(\mathsf{P}_\mathsf{V}, \psi)$, the LearnTableTransform procedure is used to synthesize a table transformation program P_T and a column mapping σ such that $\mathsf{P}_\mathsf{T}(\mathsf{T}_{\mathsf{in}})$ satisfies the

$$\frac{\tau_i \in \mathsf{partitionBySubplots}(\tau) \quad \mathsf{c_{sub}}\ \text{fresh} \quad \tau_i \Uparrow (p, \psi_i) \quad i \in [1,n]}{\tau \Uparrow (\mathsf{MultiPlot}(p, \mathsf{c_{sub}}), \bigwedge_i \psi_i)} \text{ (Multi-Plot)}$$

$$\frac{\tau_i \in \mathsf{partitionByType}(\tau) \quad \tau_i \Uparrow (l_i, \psi_i) \quad i \in [1,n]}{\tau \Uparrow \left(\mathsf{MultiLayer}(\bar{l_i}), \bigwedge_{i=1}^{n} \psi_i\right)} \text{ (Multi-Layer)}$$

$$\frac{\mathsf{T} = \bigcup_{i=1}^{n} (a_x^i, a_y^i, a_c^i, a_s^i, a_{sub}) \qquad \mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}, \mathsf{c_{size}}\ \text{fresh}}{\bigcup_{i=1}^{n} \left\{\mathsf{point}(a_x^i, a_y^i, a_c^i, a_s^i, a_{sub})\right\} \Uparrow \left(\mathsf{Scatter}(\mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}, \mathsf{c_{size}})\right), \mathsf{T} \mathbin{\hat{\subseteq}} t} \text{ (Scatter)}$$

$$\frac{\mathsf{T} = \bigcup_{i=1}^{n} (a_x^i, a_y^i, a_{y_2}^i, a_c^i, a_{sub}) \qquad \mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{y_2}}, \mathsf{c_{color}}\ \text{fresh}}{\bigcup_{i=1}^{n} \left\{\mathsf{bar}(a_x^i, a_y^i, a_{y_2}^i, a_c^i, a_{sub})\right\} \Uparrow \left(\mathsf{Bar}(\mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{y_2}}, \mathsf{c_{color}})\right), \mathsf{T} \mathbin{\hat{\subseteq}} t} \text{ (Simple Bar)}$$

$$\frac{\begin{array}{c}\mathsf{T} = \bigcup_{i=1}^{n} (a_x^i, a_{y_2}^i - a_y^i, a_c^i, a_{sub}) \qquad \mathsf{c_x}, \mathsf{c_h}, \mathsf{c_{color}}\ \text{fresh} \\ \psi_0 = \forall i \in [1,n].\ \sum_{r \in \{r \in \mathsf{T_{in}} \mid r[\mathsf{c_x}] = a_x^i \wedge r[\mathsf{c_{sub}}] = a_{sub} \wedge r[\mathsf{c_{color}}] < a_c^i\}} r[\mathsf{c_h}] = a_y^i\end{array}}{\bigcup_{i=1}^{n} \left\{\mathsf{bar}(a_x^i, a_y^i, a_{y_2}^i, a_c^i, a_{sub})\right\} \Uparrow \left(\mathsf{Stacked}(\mathsf{c_x}, \mathsf{c_h}, \mathsf{c_{color}}), \psi_0 \wedge \mathsf{T} \mathbin{\hat{\subseteq}} t\right)} \text{ (Stacked Bar)}$$

$$\frac{\begin{array}{c}\mathsf{T}_1 = \bigcup_{i=1}^{n} (a_{x_1}^i, a_{y_1}^i, a_c^i, a_p) \quad \mathsf{T}_2 = \bigcup_{i=1}^{n} (a_{x_2}^i, a_{y_2}^i, a_c^i, a_p) \quad \mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}\ \text{fresh} \\ \psi_0 = \forall i \in [1,n].\ \nexists r \in \mathsf{T_{in}}.\ (r[\mathsf{c_{color}}] = a_c^i \wedge r[\mathsf{c_{sub}}] = a_p) \rightarrow a_{x_1}^i \leq r[\mathsf{c_x}] \leq a_{x_2}^i\end{array}}{\bigcup_{i=1}^{n} \left\{\mathsf{line}(a_{x_1}^i, a_{y_1}^i, a_{x_2}^i, a_{y_2}^i, a_c^i, a_p)\right\} \Uparrow \left(\mathsf{Line}(\mathsf{c_x}, \mathsf{c_y}, \mathsf{c_{color}}), \psi_0 \wedge \mathsf{T}_1 \mathbin{\hat{\subseteq}} \mathsf{T_{in}} \wedge \mathsf{T}_2 \mathbin{\hat{\subseteq}} t\right)} \text{ (Line)}$$

Figure 2.10: Inference rules describing synthesis of visual programs

constraint $\psi[\sigma]$. Thus, if LearnTableTransform does not return $\perp$ to indicate failure, the program $\mathsf{P_V}[\sigma] \circ \mathsf{P_T}$ is guaranteed to satisfy the end-to-end specification defined by $(\mathsf{T_{in}}, \mathsf{S})$.

2.4.2 Synthesis of Visual Programs

In this section, we describe the LearnVisualProgs procedure used in Algorithm 1. This procedure is described using inference rules of the form $\tau \Uparrow (\mathsf{P_V}, \psi)$ where τ is a visual trace, $\mathsf{P_V}$ is a visual program, and ψ is a constraint. The meaning of this judgment is that, if the input table T satisfies constraint ψ, then $\mathsf{P_V}(\mathsf{T})$ yields a visualization that is consistent with τ (i.e., $\tau \subseteq \mathsf{P_V}(\mathsf{T})$). Observe that, for a given visual trace τ, there may be multiple programs that are consistent with it — i.e., we can have $\tau \Uparrow (\mathsf{P_V^i}, \psi^i)$ for multiple values of i. This is the reason why the LearnVisualProgs procedure used in Algorithm 1 returns a set rather than a singleton. In what follows, we explain each of the inference rules from Figure 2.10 in more detail.

Multiple plots. The first rule, labeled Multi-Plot, is used to synthesize programs for generating multiple subplots. Since each element in a visual trace has an attribute that identifies

which subplot it belongs to, we first partition the visual elements according to the value of this attribute. This allows us to obtain n different visual traces $\tau_1, \ldots, \tau_n$, and we recursively synthesize a visual program p_i and a constraint ψ_i for each visual trace τ_i. However, since the MultiPlot construct takes a *single* program as argument, this means that all subplots must be generated using the *same* visual program; thus, the premise of this rule stipulates that all p_i's must be the same program p. On the other hand, each subplot can impose different restrictions on the input table; thus, the input has to satisfy all of these constraints (i.e., $\bigwedge_{i=1}^{n} \psi_i$). Finally, since we do not know which column of the input table is used to generate different subplots, we make up a fresh column name called $\mathsf{c_{sub}}$ and return the synthesized program $\mathsf{MultiPlot}(p, \mathsf{c_{sub}})$ as the solution.

Multiple layers. The second rule, Multi-Layer, is similar to the Multi-Plot rule and is used to generate programs that compose different types of charts. Similar to the previous rule, we again partition elements in the visual trace according to their type (i.e., point, bar etc.) to obtain n different traces $\tau_1, \ldots, \tau_n$ and recursively synthesize a visual program l_i and a constraint ψ_i for each τ_i. Then, the synthesized program $\mathsf{MultiLayer}(\bar{l})$ will generate a visualization consistent with the visual sketch as long as the input table satisfies $\bigwedge_{i=1}^{n} \psi_i$.

Scatter plot. The next rule is used to synthesize a visual program that renders a scatter plot. Since all elements in a scatter plot must be points, the precondition of this rule requires that the visual trace is a set of points with the same subplot attribute. Furthermore, for each point p with attributes $\bar{a}^i$ in the visual sketch, there must be a corresponding row in the input table that contains exactly the values $\bar{a}^i$. To express this requirement on the input table, we construct a table T that contains rows $\bar{a}^i$ and generate the constraint $\mathsf{T} \overset{\diamond}{\subseteq} t$ where t refers to the input table for the synthesized visual program. Finally, since we do not know the names of the columns in the input table, we introduce placeholder column names $\bar{c}$ and return the program $\mathsf{Scatter}(\bar{c})$.

Bar charts. The next two rules, labeled Simple Bar and Stacked Bar, both generate bar charts and are very similar to the previous Scatter rule. For Stacked Bar, $\mathsf{c_h}$ represents the height of the bar rather than the absolute y-position; thus, we compute entries in column $\mathsf{c_h}$ as $a^i_{y_2} - a^i_y$ for the i'th row in the table sketch. Also, in addition to the constraint $\mathsf{T} \overset{\diamond}{\subseteq} t$, the Stacked Bar rule imposes an additional constraint on the input table. In particular, since the bars in a Stacked Bar chart must be stacked directly on top of each other, constraint ψ_0 essentially stipulates that the starting y position of one bar is precisely the end y-position of the previous stack below it. In practice, when computing constraint ψ_0, we compute the end y position of the stack below by summing the heights of all the individual bars below the current one.

Line chart. The final rule is used to synthesize a Line program in our visualization language. Recall that a line visual element is defined by its two end points (a_{x_1}, a_{y_1}) and (a_{x_2}, a_{y_2}), and these end points must correspond to two different rows in the input table. Thus, we generate two different constraints $\mathsf{T}_1 \overset{\diamond}{\subseteq} t$ and $\mathsf{T}_2 \overset{\diamond}{\subseteq} t$ that describe requirements imposed by the left end and right end of each line segment respectively. Finally, the constraint ψ_0 in the second line of the premise imposes the following additional restriction: If the visual sketch contains a line segment with (a_x, a_y) and (a'_x, a'_y) as its end points, there should not be another entry in the input table that belongs to the same line chart (i.e., same color and subplot) but where the x value is in the range (a_x, a'_x). Without this additional constraint ψ_0, the generated visualization would not be guaranteed to satisfy the provided visual sketch.

Properties. Our visual program inference procedure enjoys the following properties that are important for the soundness and completeness for the overall approach.

Property 1 (Decomposition)**.** Suppose that $\tau \Uparrow (\mathsf{P_V}, \psi)$ and T is a table that satisfies constraint ψ (i.e., $\mathsf{T} \models \psi$). Then, we have $\tau \subseteq \mathsf{P_V}(\mathsf{T})$.

The above property shows the soundness of the overall the synthesis algorithm. In particular, let $\mathsf{P_V}$ be a visual program synthesized in the first phase. Based on the above property, as long as we can find a table transformation program $\mathsf{P_T}$ that satisfies the specification $(\mathsf{T_{in}}, \psi)$, then we are guaranteed that the composition $\mathsf{P_V} \circ \mathsf{P_T}$ will satisfy the specification of the overall synthesis task.

Property 2 (Completeness)**.** Let τ be a visual sketch, and suppose that there exists a table T and a visual program $\mathsf{P_V}$ in $\mathcal{L}_\mathsf{V}$ such that $\tau \subseteq \mathsf{P_V}(\mathsf{T})$. Then, we have $\tau \Uparrow (\mathsf{P_V}, \psi)$ such that $\mathsf{T} \models \psi$.

This second property shows the completeness of the overall synthesis algorithm. In particular, it states that, if there exists a table T and visual program $\mathsf{P_V}$ such that $\mathsf{P_V}(\mathsf{T})$ is consistent with the given visual sketch, then our inference procedure will (a) return $\mathsf{P_V}$ as one of the solutions, and (b) T will satisfy the constraint ψ associated with $\mathsf{P_V}$.

2.4.3 Synthesizing Table Transformations via Bidirectional Reasoning

In this section, we describe the LearnTableTransform function used in Algorithm 1. This procedure is given in Algorithm 2 and takes as input the original input table $\mathsf{T_{in}}$ and the intermediate specification ψ_{out} generated during the first phase. LearnTableTransform either returns a program $\mathsf{P_T}$ such that $\mathsf{P_T}(\mathsf{T_{in}})$ is consistent with the specification ψ_{out} or yields $\perp$ to indicate failure. If synthesis is successful, LearnTableTransform additionally

returns a mapping σ from the made-up column names used in ψ_{out} to the actual column names used in $\mathsf{P_T}(\mathsf{T_{in}})$.

Algorithm 2 Table transformation synthesis algorithm.

```
1:  procedure LearnTableTransform(T_in, ψ_out)
2:      ψ_in ← (t_0 ⊆ T_in ∧ T_in ⊆ t_0)
3:      while existsNextSketch() do
4:          ℙ_0 ← getNextSketch();
5:          W ← {(ℙ_0, σ) | σ ∈ Mappings(Cols(ψ_out), Cols(ℙ_0))};
6:          while ¬W.isEmpty() do
7:              (ℙ, σ) ← W.next()
8:              if IsComplete(ℙ) then
9:                  if ℙ(T_in) ⊨ ψ_out[σ] then return (ℙ, σ)
10:                 else continue;
11:             φ ← Analyze⁺(ψ_in, ℙ) ∧ Analyze⁻(ψ_out[σ], ℙ);
12:             if UNSAT(φ) then continue
13:             □_k ← chooseHole(ℙ)
14:             W ← W ∪ {(ℙ[□_k ↦ v], σ') | v ∈ dom(□_k), Mappings(Cols(ψ_out), Cols(ℙ) ∪ {v})}
        return ⊥
```

From a high level, the outer loop of Algorithm 2 lazily enumerates program sketches based on the language from Section 2.3.2. In this context, a program sketch $\mathbb{P}$ is a sequence of instructions of the form $t = \mathsf{op}(\square_1, \ldots, \square_n)$ where t is a program variable, op is a construct in the table transformation language (e.g., `mutate`, `join`), and each $\square_i$ is a *hole* representing an unknown argument. To obtain a program that is a completion of $\mathbb{P}$, we need to fill each of the holes in the sketch with previously defined program variables or column names from the input table.

In more detail, the algorithm maintains a worklist W of elements $(\mathbb{P}, \sigma)$ where $\mathbb{P}$ is a (partially completed) program sketch and σ is a possible mapping from the made-up column names in ψ_{out} to actual column names in the output table. In particular, σ maps each column name used in ψ_{out} to an element in $\mathsf{Cols}(\mathbb{P})$, where $\mathsf{Cols}(\mathbb{P})$ includes both the columns used in $\mathsf{T_{in}}$ as well as any additional columns mentioned in $\mathbb{P}$. In each iteration of the inner while loop, the algorithm dequeues (at line 8) a pair $(\mathbb{P}, \sigma)$ and checks whether $\mathbb{P}$ is a complete program (i.e., no holes). If this is the case and $\mathbb{P}$ satisfies the specification under mapping σ (line 9), we then return $(\mathbb{P}, \sigma)$ as a solution. On the other hand, if $\mathbb{P}$ contains any remaining holes, we perform bidirectional program analysis (line 11) to check if there is any completion of $\mathbb{P}$ that *can* satisfy the (intermediate) specification ψ_{out}. In particular, line 11 of Algorithm 2 generates a constraint ϕ that is a conjunction of atomic predicates of the form $e_1 \subseteq^* e_2$ where $\subseteq^*$ represents the table inclusion relations defined earlier ($\subseteq$, $\overset{\diamond}{\subseteq}$), and each e_i is either a

$$\frac{\phi \Rightarrow t \subseteq^* \mathsf{T}}{\phi \downarrow t' = \mathsf{filter}(t, \_) : \phi \wedge (t' \subseteq^* \mathsf{T})} \qquad \frac{\phi \Rightarrow t \subseteq^* \mathsf{T}}{\phi \downarrow t' = \mathsf{select}(t, \_) : \phi \wedge (t' \subseteq^* \mathsf{T})}$$

$$\frac{\phi \Rightarrow t \subseteq^* \mathsf{T} \quad \mathsf{T}' = [\![\mathsf{mutate}(\mathsf{T}, c_t, op, \bar{c})]\!]}{\phi \downarrow t' = \mathsf{mutate}(t, c_t, op, \bar{c}) : \phi \wedge (t' \subseteq^* \mathsf{T}')} \qquad \frac{\phi \Rightarrow (t_1 \subseteq^* \mathsf{T}_1 \wedge t_2 \subseteq^* \mathsf{T}_2)}{\phi \downarrow t' = \mathsf{join}(t_1, t_2, \_) : \phi \wedge (t' \subseteq^* \mathsf{T}_1 \times \mathsf{T}_2)}$$

$$\frac{\phi \Rightarrow t \subseteq^* \mathsf{T} \quad \mathsf{T}' = [\![\mathsf{gather}(\mathsf{T}, \bar{c}_{id}, \bar{c}_{target})]\!]}{\phi \downarrow t' = \mathsf{gather}(t, \bar{c}_{id}, \bar{c}_{target}) : \phi \wedge (t' \subseteq^* \mathsf{T}')} \qquad \frac{\phi \Rightarrow t \subseteq^* \mathsf{T} \quad \mathsf{T}' = [\![\mathsf{spread}(\mathsf{T}, \bar{c}_{id}, c_{key}, c_{val})]\!]}{\phi \downarrow t' = \mathsf{spread}(t, \bar{c}_{id}, c_{key}, c_{val}) : \phi \wedge (t' \overset{\diamond}{\subseteq} \mathsf{T}')}$$

$$\frac{\forall i \in [1, n].\ \phi_{i-1} \downarrow t_i = e_i : \phi_i}{\phi_0 \downarrow \{t_1 = e_1; \ldots; t_n = e_n\} : \phi_n}\ \text{(Chain)}$$

Figure 2.11: Forward inference. Operator "$\subseteq^*$" refers to either $\subseteq$ or $\overset{\diamond}{\subseteq}$. In cases where the premise of no rule matches, we have an implicit judgment $\phi \downarrow s : \phi$ to propagate the input constraint.

program variable or a concrete table. If these generated constraints result in a contradiction, there is *no* sketch completion that is consistent with ψ_{out}; thus, the algorithm moves on to the next element in the worklist (line 12). On the other hand, if we cannot prove the infeasibility of $\mathbb{P}$, we pick one of the holes $\square_k$ used in the sketch and add a new set of partial programs to the worklist by instantiating that hole with some element in its domain (line 13). The domain of the hole is determined by its type, columns in the input schema for the given statement, and previously defined variables in the partial program. Since hole $\square_k$ may have been filled with a new column name $v \notin \mathrm{Cols}(\mathbb{P})$, we therefore also update the worklist to consider any new mappings σ' that we have not previously considered.

As is evident from the above discussion, a key part of our table transformation synthesis algorithm is the Analyze$^+$ and Analyze$^-$ procedures for performing forward and backward inference to generate table inclusion constraints. These procedures are described in Figure 2.11 and Figure 2.12 using inference rules of the form $\phi \downarrow s : \phi'$ (for the forward analysis) and $\phi \uparrow s : \phi'$ (for the backward analysis). The meaning of the judgment $\phi \downarrow s : \phi'$ is that, assuming ϕ holds *before* executing statement s, then ϕ' must hold *after* executing s. Similarly, $\phi \uparrow s : \phi'$ means that, if ϕ holds *after* executing s, then ϕ' must hold *before* s (i.e., ϕ' is a *necessary* precondition for ϕ but may not be *sufficient* to guarantee it). Since the inference rules shown in Figure 2.11 and Figure 2.12 follow from the semantics of the table transformation language, we do not explain them in detail. However, a key design decision is that our analysis *on purpose* does not compute strongest post-conditions (for the forward analysis) or strongest necessary preconditions (for the backward analysis) in order to ensure that the cost

of deductive reasoning does not overshadow its benefits. For example, in the reasoning rule for `summarize` in Figure 2.12, we over-approximate all aggregation functions as uninterpreted functions; thus the inferred pre-condition only requires that input table t include content from non-aggregated columns (T') in the output table t'. While a more precise analysis rule could consider the underlying semantics of different aggregation operators, this kind of reasoning would be prohibitively expensive [193] and outweigh the benefits obtained from better pruning.

For the same reason, our procedure for checking satisfiability of table inclusion constraints (described in Figure 2.13) is also incomplete and intentionally over-approximates satisfiability. Thus, while the unsatisfiability of the generated constraints ensures the infeasibility of a given partially completed sketch, the converse is not true – that is, our deductive reasoning technique may fail to prove infeasibility of a sketch even though no valid completion exists.

$$\frac{\phi \Rightarrow \mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t'}{\phi \uparrow t' = \mathsf{filter}(t, \_) : \phi \wedge (\mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t)} \qquad \frac{\phi \Rightarrow \mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t' \quad \mathsf{T}' = \mathsf{T}[-c_{target}]}{\phi \uparrow t' = \mathsf{mutate}(t, c_{target}, \_, \_) : \phi \wedge (\mathsf{T}' \mathbin{\overset{\diamond}{\subseteq}} t)}$$

$$\frac{\phi \Rightarrow \mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t'}{\phi \uparrow t' = \mathsf{select}(t, \_) : \phi \wedge (\mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t)} \qquad \frac{\phi \Rightarrow \mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t' \quad \mathsf{T}' = [\![\mathsf{gather}(\mathsf{T}, \bar{c}_{id}, \mathsf{schema}(\mathsf{T}) - \{\bar{c}_{id}\})]\!]}{\phi \uparrow t' = \mathsf{spread}(t, \bar{c}_{id}, \_, \_) : \phi \wedge (\mathsf{T}' \mathbin{\overset{\diamond}{\subseteq}} t)}$$

$$\frac{\phi \Rightarrow \mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t' \quad \mathsf{T}' = \mathit{RemoveDuplicates}(\mathsf{T}[\bar{c}_{id}])}{\phi \uparrow t' = \mathsf{gather}(t, \bar{c}_{id}, \_) : \phi \wedge (\mathsf{T}' \mathbin{\overset{\diamond}{\subseteq}} t)} \qquad \frac{\phi \Rightarrow \mathsf{T} \mathbin{\overset{\diamond}{\subseteq}} t' \quad \mathsf{T}' = \mathsf{T}[-c_{target}]}{\phi \uparrow t' = \mathsf{summarize}(T, \_, \_, c_{target}) : \phi \wedge (\mathsf{T}' \mathbin{\overset{\diamond}{\subseteq}} t)}$$

$$\frac{\forall i \in [1, n].\ \phi_i \uparrow t_i = e_i : \phi_{i-1}}{\phi_n \uparrow \{t_1 = e_1; \ldots; t_n = e_n\} : \phi_0} \text{ (Chain)}$$

Figure 2.12: Backward inference. Operator "$\subseteq^*$" refers to either $\subseteq$ or $\overset{\diamond}{\subseteq}$, and *RemoveDuplicates* removes duplicate tuples from the input table. As in the forward analysis, we assume an implicit rule $\phi \uparrow s : \phi$ that applies if none of the other premises are met.

Properties. We end this section by describing some salient properties of Algorithm 2 that are important for the soundness and completeness of the end-to-end synthesis approach.

Property 3 (Forward Analysis)**.** Let $\mathbb{P}$ be a partially completed sketch with argument t and return parameter t'. Then, if $t = \mathsf{T}_{\mathsf{in}} \downarrow \mathbb{P} : \phi$ and $\phi \Rightarrow (t' \mathbin{\overset{\diamond}{\subseteq}} \mathsf{T}')$, then *any* completion P of $\mathbb{P}$ satisfies $\mathsf{P}(\mathsf{T}_{\mathsf{in}}) \mathbin{\overset{\diamond}{\subseteq}} \mathsf{T}'$.

By design, our forward analysis rules exploits the fact that many table transformation operators are monotonic over the input. This property essentially captures the correctness

$$\frac{\phi = e_1 \subseteq^* e_2 \wedge \phi_0}{\phi \Rightarrow e_1 \subseteq^* e_2} \qquad \frac{\phi \Rightarrow e_1 \subseteq^* e_2 \quad \phi \Rightarrow e_2 \subseteq^* e_3}{\phi \Rightarrow e_1 \subseteq^* e_3} \qquad \frac{\phi \Rightarrow e_1 \subseteq e_2}{\phi \Rightarrow e_1 \overset{\diamond}{\subseteq} e_2}$$

$$\frac{\phi \Rightarrow T_1 \subseteq^* t \quad \phi \Rightarrow t \subseteq^* T_2 \quad T_1 \not\subseteq^* T_2}{\phi \Rightarrow \bot}$$

Figure 2.13: Inference rules for checking satisfiability of table inclusion constraints. Operator "$\subseteq^*$" refers to either $\subseteq$ or $\overset{\diamond}{\subseteq}$, and metavariable e refers to either a program variable t in ϕ or a concrete table T.

of forward inference. In particular, it says that, if we deduce that the output of $\mathbb{P}$ on T_{in} is a sub-table of T', then this is true for every completion of $\mathbb{P}$. The following property states something similar for the backward analysis:

Property 4 (Backward Analysis)**.** Let ϕ be a constraint and $\mathbb{P}$ be a partially completed sketch with input parameter t. Then, if $\phi \uparrow \mathbb{P} : \phi'$ and $\phi' \Rightarrow (\mathsf{T}' \overset{\diamond}{\subseteq} t)$, then for any completion P of $\mathbb{P}$ and any input table T such that $\mathsf{P}(\mathsf{T}) \models \phi$, we have $\mathsf{T}' \overset{\diamond}{\subseteq} \mathsf{T}$.

Similarly, our backward analysis rules by design conservatively propagate known values from the output to inputs. This property states that any conclusions reached by backward inference apply to all completions of $\mathbb{P}$. Finally, we can state the following property about the correctness of our pruning strategy:

Property 5 (Pruning Soundness)**.** Given a partially completed sketch $\mathbb{P}$, suppose we have $\psi \uparrow \mathbb{P} : \phi^-$ and $(t = \mathsf{T}_{\mathsf{in}}) \downarrow \mathbb{P} : \phi^+$. Let P be a completion of $\mathbb{P}$ such that $P(\mathsf{T}_{\mathsf{in}}) \models \psi$, and let σ be the resulting valuation after executing P on T_{in}. Then, we have $\sigma \models \phi^+ \wedge \phi^-$.

In other words, our pruning technique never rules out completions of $\mathbb{P}$ that actually satisfy the given specification $(\mathsf{T}_{\mathsf{in}}, \psi)$.

2.5 Implementation

We have implemented the proposed technique in a tool called Viser, which is written in Python. Viser takes two inputs, namely the original data source (which can consist of one or more tables) as well as a visual sketch. Currently, Viser requires the visual sketch to be expressed as a visual trace; however, with some additional engineering effort, it would be possible to integrate Viser with visual demonstration interfaces such as Lyra [152] or VisExemplar [151] to automatically generate visual traces from the demonstration.

Extension to visualization Language. Viser can generate visual programs in Vega-Lite [154], ggplot2, and a subset of `Matplotlib`. To handle all of these libraries, our implementation

supports a richer visualization DSL than the one given in Figure 2.8. In particular, Viser supports two additional visualization constructs, `AreaChart` and `StackedAreaChart`, which provide another mechanism for visualizing quantities that change over time. To support this richer visualization language, we also extend our visual trace language from Figure 2.5 with an element called `area`. Besides adding new constructors, we also extend existing constructors to take additional attributes as input. For example, the attribute a_{shape} allows specifying the shape associated with each point in a scatter plot. Another attribute, a_{order}, for line charts allows specifying a custom order instead of using the default x-axis value.

Extension to table transformation Language. The table transformation language used in our implementation extends Figure 2.6 with a few additional constructs inspired by commonly used operators in the `tidyverse` R package. For example, the table transnformation DSL in our implementation allows another construct called `separate` that is commonly used for table reshaping as well as a construct called `cumsum` for computing cumulative sum for a given column. Our implementation also allows a more expressive version of the `mutate` construct that supports a broader set of binary computations including arithmetic operations and string concatenation.

Multi-layered visualizations. To simplify presentation in the technical section, we assumed that a visual program takes a single table as input. However, in many visualization libraries (e.g., `ggplot2`), the semantics of the `MultiLayer`$(l_1, \ldots, l_n)$ construct is that each different nested visual program l_i operates on the i'th input table. To support these richer semantics, our implementation synthesizes multiple different table transformation programs for each layer. To achieve this goal, the inference procedure for visual programs generates n different intermediate specifications, one for each layer in the visual program, and we use the same table transformation procedure to synthesize n different programs.

Ranking. Following the Occam's razor principle, Viser explores programs in increasing order of program size up to a fixed bound K. In practice, to leverage the inherent parallelism of our algorithm, Viser uses multiple threads to search for solutions of different sizes and ranks programs according to their size.

2.6 Evaluation

In this section, we evaluate the effectiveness of our approach on 83 real world visualization tasks collected from online forums and tutorials for advanced users. The goal of our evaluation is to examine the following research questions:

1. Can Viser solve real-world visualization tasks based on small visual sketches?

2. Does the decomposition of the synthesis task into two sub-problem improve synthesis efficiency?

3. Are there any advantages to using our proposed table transformation algorithm compared to re-using an existing state-of-the-art technique?

2.6.1 Benchmarks

We evaluate Viser on 83 visualization benchmarks [4], 63 of which are collected from highly-reputed visualization tutorials for Excel [5] and Vega-Lite [6], and 20 of which are collected from the `ggplot2` sub-forum on StackOverflow. To collect these benchmarks, we went through a few hundred visualization examples and retained all tasks that conform to the following criteria:

1. The target visualization is expressible in our language. ($> 80\%$ visualizations are expressible in our language, and we discuss the remaining ones in Section 2.6.6.)

2. The example contains the actual input table.

3. The task requires table transformation to generate the intended visualization.

4. There is a way to produce the target visualization based on information in the example.

We have these criteria because (1) tasks that cannot be achieved using our visualization language are out of scope for this work, (2) we need the raw data as an input to our tool, (3) we do not want to evaluate on trivial benchmarks, and (4) we need the target visualization to determine if our tool can produce the correct program.

Among our 83 benchmarks, 40 of them contain subplots or multi-layered charts. Furthermore, for most benchmarks, the original data source consists of a single table whose size ranges from 4×3 to 3686×9, with average size 100×10.

2.6.2 Key Results

To evaluate Viser on these benchmarks, we programmatically generated small visual sketches consisting of 4 randomly sampled visual elements per layer. Specifically, given a

[4] Available at https://chenglongwang.org/falx-project

[5] https://sites.google.com/site/e90e50/, https://chandoo.org/wp/category/visualization/, https://peltiertech.com/

[6] https://vega.github.io/vega-lite/examples/

target visualization expressed in our visual trace language, we sampled 4 elements from the corresponding visual trace and used this as our visual sketch. While the number 4 is somewhat arbitrary, we believe that a visual sketch with four elements is small enough that it would not be too onerous for users to construct such a sketch.

Given these randomly generated visual sketches, we evaluated Viser using the following methodology. We fixed a time budget t, and we let Viser explore multiple visualization scripts consistent with the provided sketch within this time budget. Then, for a given value of t, we consider the benchmark to be solved if any of the programs explored by Viser within that time budget generates the intended visualization.

Table 2.1: Experimen Summary.

Time budget	# solved
1s	26
10s	49
60s	62
600s	70

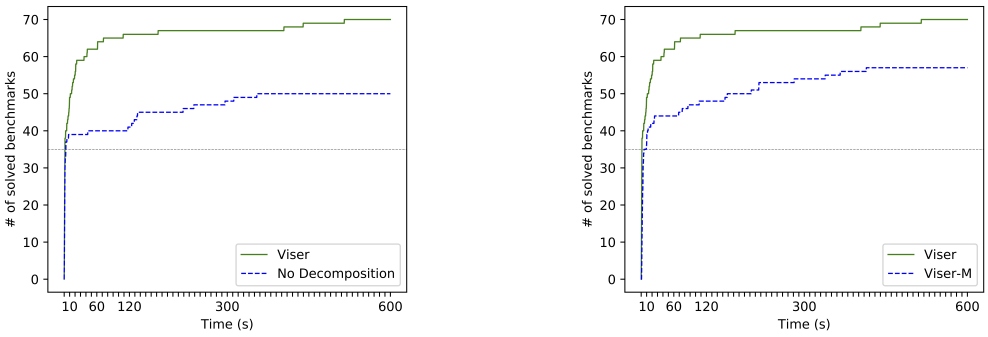
Table 2.2: The impact of sketch size on ranking.

# samples	top-1	top-5	top-10	>10
1	14	29	36	66
2	20	37	43	68
3	22	45	53	69
4	26	49	57	70
6	31	52	57	70
8	30	58	63	70

The results of this experiment are summarized in Table 2.1. For a time budget of 600 seconds, Viser is able to solve 70 out of 83 benchmarks (84%). If we reduce the time budget to a minute, then Viser can solve 75% of the benchmarks. Furthermore, 59% of the benchmarks can be solved within 10 seconds, and 31% can be solved within one second.

Table 2.2 explores the same experimental data from a different perspective. Specifically, given a value k, let us consider a benchmark to be "solved" if the desired visualization is one of the first-k visualizations returned by the tool. As shown in the first row of Table 2.2, among the 70 benchmarks that can be solved within the 600 second time limit, 26 (37%) of them are ranked as the top-1 solution, and 49 (70%) and 57 (81%) are ranked as top-5 and top-10 respectively. Given that a user can quickly look through 10 visualization results and decide if any of them is the desired visualization, we believe these results affirmatively answer our first research question.

In Table 2.2, we also explore the impact of sketch size on synthesis results. Specifically, recall that we generate the visual sketches by randomly sampling n elements from the target visualization, and, so far, our discussion focused on the results for $n = 4$. Table 2.2 shows the ranking of the desired visualization as we increase n to 6 and 8 and decrease to 1, 2, 3 respectively. For cases with more visual trace samples, since the visual sketch contains more information as we increase n, Viser synthesizes fewer spurious programs and the



(a) Baseline: with no decomposition

(b) Baseline: Viser-M

Figure 2.14: Comparison of Viser against different baselines

ranking of the target visualization improves as a result. [7] This finding indicates that users can incrementally add more visual elements to the output example and gradually refine the synthesis results when the initial top-ranked solutions fail to meet the user's expectation.

2.6.3 Evaluating Impact of Decomposition

As mentioned throughout the paper, a key design choice underlying our technique is to decompose the visualization task by inferring an intermediate specification for each possible visual program. In this section, we aim evaluate the empirical significance of this decomposition.

To perform this study, we implement a baseline using the following methodology: Similar to Viser, the baseline first infers a visual program P_V consistent with the sketch as discussed in Section 2.4.2; however, the baseline approach does not generate an intermediate specification. Then, during table transformation synthesis, for every enumerated program P_T, the baseline checks whether $P_V(P_T(T_{in}))$ is consistent with the visual sketch. In other words, without the intermediate specification, table transformation synthesis in the baseline approach degenerates into enumerative search.

Figure 2.14a compares the performance of Viser against this baseline without decomposition. Here, the x-axis shows the time budget per benchmark, and the y-axis shows the percentage of benchmarks that can be solved within the given budget. Furthermore, the solid green line corresponds to Viser, and the dashed blue line corresponds to the

[7] The reader may notice that $n = 6$ does better compared to $n = 8$ for the top-1 result; this is caused by the random sampling of visual elements.

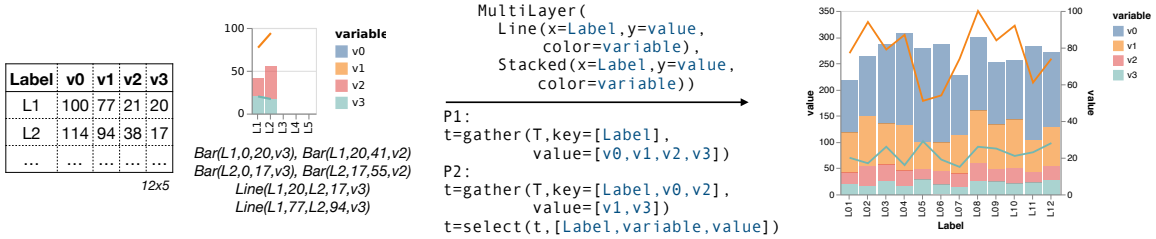


Figure 2.15: Illustration of visualization task #1.

baseline without decomposition. As we can see from this figure, having an intermediate specification greatly benefits our synthesis algorithm. In particular, without decomposition, the percentage of benchmarks solved within a 600s (resp. 120s) time-limit drops from 84% (resp. 80%) to 60% (resp. 49%).

2.6.4 Evaluating Table Transformation Algorithm

In this section, we evaluate the impact of using our new table transformation algorithm over an existing technique that addresses the same problem. To perform this evaluation, we use a variant of Viser that we refer to Viser-M that uses Morpheus [58] as its table transformation back-end. However, since the original Morpheus tool is written in C++, we instead use a newer implementation of Morpheus written in Python [117] (by the original Morpheus authors). Furthermore, since Morpheus does not support our table inclusion constraints, we "translate" the generated intermediate specification to Morpheus' constraint language. In particular, given an intermediate specification ϕ, our "translation" infers the strongest formula expressible in Morpheus' language, which consists of equality and inequality constraints on the number of rows or columns of the output table.

The results of this comparison are presented in Figure 2.14b, which plots the number of benchmarks that can be solved within a given time budget for both Viser and Viser-M. As we can see from this figure, the table transformation synthesizer proposed in this paper yields much better results compared to Morpheus. In particular, within a 600s (resp. 120s) time-limit, Viser-M can solve 69% (resp. 58%) of the benchmarks compared to 84% (resp. 80%) for Viser.

2.6.5 Example Tasks

To give the reader some intuition about the class of tasks that can be automated using Viser, we highlight three representative visualization tasks from our benchmark set.

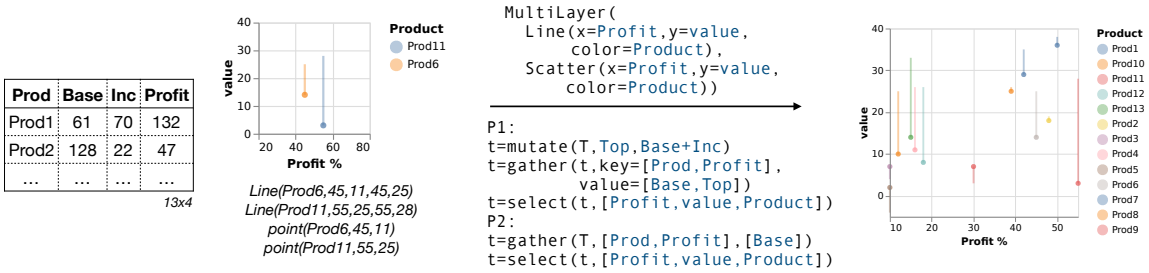


Figure 2.16: Illustration of task #2.

Task #1. Figure 2.15 shows a visualization task involving multiple layers, consisting of a stacked bar chart and a (multi-layered) line chart. The left-hand side of the figure shows the original data source, and right next to it, we show the visual sketch (and its corresponding visual trace) that we use to automate this visualization task. The synthesized visualization script is indicated on both sides of the arrow (visual program on top; table transformation at the bottom). Finally, the right-most part of the figure shows the resulting visualization that is obtained by applying the synthesized script to the input table.

Observe that the synthesized visual program refers to columns such as `variable` and `value` that do not exist in the original table and that are introduced by the table transformation program. Further, as discussed in Section 2.5, Viser synthesizes as many table transformation programs as there are layers; thus, we have two separate table transformation programs for this example. The visualization shown on the right is one of the top-2 visualizations produced by Viser for this example.

Task #2. Figure 2.16 shows a scenario in which a user has data on corporate profits and wants to generate a so-called "cherry chart". Since most visualization libraries do not have a "cherry chart" primitive, generating this plot requires layering a line chart with a scatter plot. The left half of Figure 2.16 shows the input to Viser, and right half shows the synthesized programs and the corresponding visualization of the entire dataset. As in the previous example, we have multiple table transformation programs, one for each layer, and both the visual program and the table transformation programs refer to a column called `value` that does not exist in the original table and that is introduced by the `gather` operation. In this case, the intended visualization shown on the right is top result returned by Viser.

Task #3. Figure 2.17 shows a visualization task that requires drawing a so-called "heat map" that visualizes the number of visits to various websites at each hour on different days. In this case, the input data is very large and contains close to 2000 rows. Furthermore, the corresponding data transformation program is quite complex and requires both computation

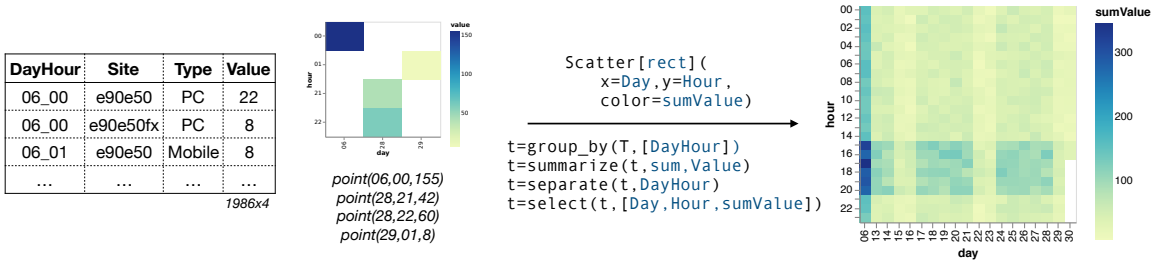


Figure 2.17: Illustration of task #3.

as well as reshaping. Specifically, the table transformation program computes the total number of website visits for each one-hour time period, which is stored in a column called `sumValue` introduced by `summarize`. The visual program refers to this new `sumValue` column to generate the desired heat map. Given the visual trace shown in Figure 2.17, the intended visualization (on the right) is ranked within the top 20 visualizations, but if we provide a visual sketch with 8 elements instead of 4, then the intended visualization is ranked number one.

2.6.6 Discussion of Limitations

As reported in Section 2.6.2, Viser cannot synthesize 13 of the 83 benchmarks within a time limit of 10 minutes. To understand the limitations of Viser in practice, we manually inspected these benchmarks and explain the insights we gained from our examination. Specifically, we highlight two reasons that are responsible for Viser not finding the desired visualization within the given time budget.

- *Size of the input table.* The size of the input table can affect the performance of Viser in two ways. First, the search space covered by the table transformation language grows exponentially as we increase the size of the table. This is because constructs in the table transformation language use names of columns as arguments, and, furthermore, rows can become columns during the reshaping process. Second, the table transformation synthesis engine tracks table inclusion constraints where one size of the inclusion is a concrete table. Thus, the larger the initial input table, the more overhead associated with program analysis.

- *Complex table transformations.* Some tasks in our benchmark suite require very sophisticated table transformations that Viser is unable to explore within the given time budget. For instance, some benchmarks that Viser cannot solve within 10 minutes

require a combination of relational operations, string manipulation, column-wise arithmetic computations, and table pivoting.

In addition, recall from Section 2.6.1 that approximately 20% of the visualization tasks we inspected are not expressible in our visualization language. These benchmarks fall into roughly three classes: (1) visualizations involving continuous functions, (2) visualizations that require custom shapes provided by the user (e.g., emoji icons), and (3) visualizations that cannot be placed in a standard coordinate system, (e.g., tree-maps and parallel coordinates).

2.7 Related Work

In this section, we survey closely related work on data visualization and program synthesis.

Automation for visualization. There has been significant recent interest in (semi-)automating various types of visualization tasks. These efforts include both visualization recommendation systems as well as visualization exploration tools. Among these, visualization recommendation systems like Draco [125], CompassQL [208], and ShowMe [112] recommend top completions of an incomplete visualization program. On the other hand, visualization exploration tools, such as VisExamplar [151], Visualization-by-Sketching [158], Polaris [175], and Voyager [209, 210], aim to generate diverse visualizations based on user demonstrations, which can include graphical sketches, manipulation trajectories, and constraints. All of these existing systems focus on creating visualizations for a fixed dataset and require the user to prepare the data for a specific visualization API. In contrast, our approach also handles the data preparation and wrangling aspect of data visualization and can be viewed as being more user-friendly in this respect. However, it is worth noting that many of these systems are complementary to the approach proposed in this paper. For example, our approach can be used in conjunction with existing systems to rank synthesis results that are consistent with the demonstration. Furthermore, our approach can work with existing visualization demonstration interfaces [152, 151] to reduce user effort in creating a visual sketch.

Automating table transformations. Our technique for synthesizing table transformations is related to several recent techniques for automating data wrangling [77, 192, 58, 57, 218, 182]. Among these, Scythe generates SQL queries from input-output examples and prunes the search space by grouping partial queries into equivalence classes [192]. The Morpheus system automates table transformation tasks that arise in R programming and leverages logical specifications of R library functions to prune the search space using SMT-based reasoning [58]. Morpheus's successor, Neo, generalizes this technique to other domains and further uses logical specifications to learn from failed synthesis attempts [57]. A unifying theme among all these prior efforts is that the specification is a pair of concrete input and

output tables. In contrast, our specification does not involve a concrete output table but rather a set of table inclusion constraints; furthermore, our approach works with the original (potentially very large) dataset and does not require the user to craft a small representative input table. The large input table assumption is likely to be problematic for systems like Scythe and Morpheus that require evaluating the partial program on the input table. Furthermore, since the output specification is much weaker in our context compared to existing systems, forward reasoning alone is not sufficient to meaningfully prune the search space, as demonstrated in our evaluation.

Program analysis for program synthesis. Given the large search space that must be explored by program synthesizers, a common trick is to perform lightweight program analysis to prune the search space [198, 139, 196, 61]. The particular flavor of program analysis varies between different synthesizers and ranges from domain-specific deduction [61, 196] to abstract interpretation [198] to SMT-based reasoning [57, 139]. Furthermore, some of these techniques leverage program analysis to construct a compact version space [198, 140] while others use it to prune partial programs in enumerative search [57, 196]. Similar to these efforts, we also use program analysis to prove infeasibility of partial programs but with two key differences: First, our analysis is tailored to table transformation programs and infers inclusion constraints between tables. Second, since neither forward nor backward reasoning is sufficient to meaningfully prune the search space on their own, we use bi-directional analysis to improve pruning power without having to resort to heavy-weight semantics motivated by prior work in program analysis [147, 25, 51] and verification [193]. In this respect, our synthesis method is similar to Synquid [139] which uses a form of bidirectional refinement type checking to prune its search space. However, unlike Synquid which requires precise refinement type specifications of components, our method uses lightweight semantics that are tailored specifically for our table transformation DSL. Furthermore, in contrast to Synquid which leverages an SMT solver, our method uses a custom, and deliberatively incomplete, solver for checking satisfiability at low cost.

Compositional program synthesis. As mentioned throughout the paper, our technique decomposes the synthesis task into two separate sub-problems. In this respect, our method is similar to prior efforts on compositional program synthesis [61, 139, 145, 140, 138, 113, 124]. Among these techniques, λ^2 uses domain knowledge about the DSL constructs to infer input-output examples for sub-expressions whenever feasible [61], FlashMeta (and its variants) use inverse semantics of DSL constructs to propagate examples backwards [140], and Optician [113, 124] decomposes the synthesis process using DNF regular expression outlines. Synquid also tries to decompose the overall specification into sub-goals using a technique referred to as "round-trip type checking" [139]. On a slightly different note,

the technique of Raza et al. [145] also performs synthesis in a compositional way, but it leverages natural language to identify sub-problems and asks the user to provide input-output examples for each auxiliary task. In contrast to all of these techniques, our method decomposes the visualization synthesis task into two sub-problems over *different* DSLs and uses the inverse semantics of the visualization DSL to infer precise constraints on the input table. The inferred specification is precise in the sense that any table transformation program that satisfies this specification is guaranteed to be a valid solution.

2.8 Summary

In this chapter, we introduced *visualization-by-example*, a new program synthesis technique for generating visualizations from visual sketches. Given the original raw data and a visual sketch consisting of a few visual elements, our technique can automatically synthesize visualization scripts that yield a visualization consistent with the user's visual sketch. Our technique decomposes the synthesis problem into two sub-tasks by inferring an intermediate specification in the form of table inclusion constraints. This intermediate specification is then used to guide the synthesis of table transformation programs using a combination of bi-directional program analysis and lightweight inference over table inclusion constraints.

We have implemented the proposed method as a new tool called Viser that allows users to explore different visualizations for the entire data set based on a small visual sketch. Notably, and unlike any other visualization tool, Viser can perform any necessary data wrangling tasks, including reshaping and aggregation. We have evaluated Viser on a benchmark suite consisting of 83 visualization tasks obtained from on-line forums and tutorials. Given a visual sketch consisting of four visual elements and a time budget of 600 seconds, Viser can solve 84% of these tasks. Furthermore, among the 70 tasks that can be solved within the time budget, the desired visualization is ranked within top 5 in 76% of the cases. Beyond showing that Viser can help automate real-world data visualization tasks, our evaluation also confirms the importance of decomposing the synthesis task as well as the necessity of our proposed table transformation synthesizer.

In the near future, we are interested in integrating Viser with visual demonstration interfaces proposed in the visualization literature. Such interfaces can make Viser more user-friendly by providing a graphical user interface that allows users to draw visual sketches rather than write visual traces in a semi-formal language. We also plan to improve the search heuristics underlying Viser so that visualization scripts that generate the intended visualization are more likely to be explored first.

Appendix

2.8.1 Full Visualization Language

In this section, we present definitions of the full visualization language $\mathcal{L}_{\mathsf{V}}$ and the full trace language $\mathcal{L}_{\tau}$ we use in practice for the visualization by example task.

$$
\begin{array}{rcll}
\mathsf{P_V} & = & \mathsf{MultiPlot}(SP, c_{\mathsf{row}}, c_{\mathsf{col}}) \mid SP & \text{(Faceted Chart)} \\
SP & = & \mathsf{MultiLayer}(\bar{L}) \mid L & \text{(Layered Chart)} \\
L & = & \mathsf{Scatter}[mark](c_x, c_y, c_{shape}, c_{color}, c_{size}, c_{text}) & \text{(Scatter Plot)} \\
 & \mid & \mathsf{Line}(c_x, c_y, c_{width}, c_{order}, c_{color}, c_{detail}) & \text{(Line Chart)} \\
 & \mid & \mathsf{Bar}(c_x, c_{x_2}, c_y, c_{y_2}, c_{color}, c_{width}) & \text{(Bar Chart)} \\
 & \mid & \mathsf{StackedBar}[orient](c_x, c_h, c_{color}, c_{width}) & \text{(Stacked Bar Chart)} \\
 & \mid & \mathsf{Area}(c_x, c_{x_2}, c_y, c_{y_2}, c_{color}) & \text{(Area Chart)} \\
 & \mid & \mathsf{StackedArea}[orient](c_x, c_h, c_{color}) & \text{(Stacked Area Chart)} \\
c & = & column \mid \epsilon & \\
mark & = & \mathsf{point} \mid \mathsf{circle} \mid \mathsf{text} \mid \mathsf{rect} \mid \mathsf{tick} & \\
orient & = & \mathsf{horizontal} \mid \mathsf{vertical} &
\end{array}
$$

Figure 2.18: The full visualization language $\mathcal{L}_{\mathsf{V}}$.

Figure 2.18 defines our full visualization language $\mathcal{L}_{\mathsf{V}}$. A visual program $\mathsf{P_V}$ either creates a grid of multiple plots using the MultiPlot construct or a single plot SP (where grid index for each subplot is determined by its value in column c_{col} and c_{row}). Each plot can in turn consist of multiple layers (indicated by the MultiLayer construct) or a single layer. Each layer is either a scatter plot (Scatter[*mark*], where the paramter *mark* decides the shape of scatter points), a line chart (Line), a bar chart (Bar), a stacked bar chart (StackedBar), an area chart (Area) or a stacked area chart (StackedArea). The MultiLayer construct in this language is used to compose *different* kinds of charts in the same plot (e.g., a scatter plot and a line chart), as our visualization language is already rich enough to allow layering the same type of chart within a plot.

Visual traces encode semantics of visualizations. A visual trace, denoted τ, is a set of basic visual elements, i.e., point, line, barH (horizontal bar), barV (vertical bar), or area, together with the attributes of each element (position, size, color, etc.). Figure 2.19 shows the language in which we express visual traces. Here, e denotes a visual element, and a is an *attribute* of that element:

- *Color attribute:* This attribute, denoted a_{color} specifies the color of a visual element.
- *Position attributes:* Position attributes, such as a_x, a_{x_1}, a_{y_2} etc., specify the canvas positions for a visual element. For line, (a_{x_1}, a_{y_1}) and (a_{x_2}, a_{y_2}) specifies the starting and

$$
\begin{array}{rcll}
\tau & = & \{e_1, \ldots, e_n\} & \\
e & = & \mathsf{barV}(a_x, a_{y_1}, a_{y_2}, a_{width}, a_{color}, a_{col}, a_{row}) & \text{(Vertical Bar)} \\
 & | & \mathsf{barH}(a_y, a_{x_1}, a_{x_2}, a_{width}, a_{color}, a_{col}, a_{row}) & \text{(Horizontal Bar)} \\
 & | & \mathsf{point}(a_x, a_y, a_{shape}, a_{color}, a_{size}, a_{col}, a_{row}) & \\
 & | & \mathsf{line}(a_{x_1}, a_{y_1}, a_{x_2}, a_{y_2}, a_{width}, a_{color}, a_{col}, a_{row}) & \\
 & | & \mathsf{area}(a_{x_{tl}}, a_{y_{tl}}, a_{x_{bl}}, a_{y_{bl}}, a_{x_{tr}}, a_{y_{tr}}, a_{x_{br}}, a_{y_{br}}, a_{color}, a_{col}, a_{row}) & \\
mark & = & \mathsf{point} \mid \mathsf{circle} \mid \mathsf{text} \mid \mathsf{rect} \mid \mathsf{tick} &
\end{array}
$$

Figure 2.19: The full trace language $\mathcal{L}_\tau$, where metavariable a refers to constants.

the end points of a line segment. For the bar visual element, a_{y_1}, a_{y_2} specify the start and end y-coordinates of a (vertical) bar. Similarly, attributes $a_{x_{tl}}, a_{y_{tl}}, a_{x_{bl}}, a_{y_{bl}}, a_{x_{tr}}, a_{y_{tr}}, a_{x_{br}}, a_{y_{br}}$ specity x, y positions of top left / bottom left / top right / bottom right corners of an area element.

- *Size / Shape attributes:* Attributes a_{size} and a_{shape} specify the size and the shape variation of a given point element in a scatter plot.
- *Width attribute:* The attribute a_{width} specifies the width of a given barH / barV / Line element.
- *Subplot attribute:* Attributes a_{col}, a_{row} specify the subplot that a given visual element belongs to. For instance, a point with $a_{col} = 1$ and $a_{row} = 2$ belongs to the subplot located the first column and second row of visualization containing multiple plots.

Part II

SYNTHESIS-POWERED DATABASE QUERYING

Chapter 3

Scythe: Synthesizing Highly Expressive SQL Queries

SQL is the de facto language for manipulating relational data. Though powerful, SQL queries can be difficult to write due to their highly expressive constructs. Using the programming-by-example paradigm to help users write SQL queries presents an attractive proposition, as evidenced by online help forums such as Stack Overflow. However, developing techniques to synthesize SQL queries from input-output (I/O) examples has been difficult due to SQL's rich set of operators.

In this chapter, we present a scalable and efficient algorithm to synthesize SQL queries from I/O examples. Our key innovation is the development of a language for abstract queries: queries with uninstantiated operators that can express a large space of SQL queries efficiently. Using abstract queries to represent the search space nicely decomposes the synthesis problem into two tasks: (1) searching for abstract queries that can potentially satisfy user examples, and (2) instantiating candidate abstract queries and ranking the results. We implemented the algorithm in a tool, called Scythe, and evaluated it on 193 benchmarks collected from Stack Overflow. Our results showed that Scythe efficiently solved 74% of the benchmarks, most in just a few seconds. Queries synthesized by Scythe range from simple ones involving a single selection to complex ones with six levels of nested queries.

3.1 Introduction

Relational databases serve an important role in modern data management, and SQL remains one of the most commonly used query languages to manipulate relational data. SQL can be used for many basic tasks, such as selecting columns from a table; its rich features also make it useful for solving complex data manipulation tasks, such as computing arg max and joining multiple tables together with aggregates. However, the various operators available in SQL and the many ways that they can be combined to form advanced idioms (e.g., correlated subqueries, unions, nested queries, groupings, various types of joins, etc) make the language difficult to master, as evidenced by over 10,000 Stack Overflow SQL related posts. In fact, many problems are so common among end users that they are grouped with popular tags, such as "greatest-n-per-group," "argmax," and "moving-average."

Though end users find solving these problems to be challenging, they can often specify the problems using input-output (I/O) examples, as observed in many Stack Overflow posts. Given the recent advances in programming-by-example (PBE) systems [71, 163, 77, 98, 102], building a tool that helps users write SQL by soliciting I/O examples from users would alleviate their need to learn complex SQL constructs and idioms.

Prior work [218, 182] has developed automatic synthesizers for SQL queries using I/O examples. However, it handles only a small subset of the language and does not cover a wide range of practical tasks. We observe that the difficulty in developing automatic synthesizer for SQL queries results from several of its unique features. First, the space of SQL queries is huge: many SQL operators (selection, projection, joins, grouping, etc) are parameterized by predicates, and they can be composed with each other. Second, the first-class value in SQL, table, is a type of compound value that is expensive to compute and memoize: tables computed from joins and unions can contain hundreds to thousands scalar values. Third, unlike spreadsheet data transformation languages, whose operators can be decomposed and inferred backwardly from output examples, SQL operators cannot be trivially decomposed and learned in this way. For the example in Figure 3.1, the Join predicates cannot be inferred independently from its nested subqueries as they jointly affect the output table. Thus, it is unclear whether the "divide-and-conquer" synthesis algorithms used in other domains [71, 166, 77, 140] would remain scalable in SQL.

Our key insight to address the above challenges is to develop a new abstraction – the language of abstract queries – to decompose the originally challenging synthesis problem. Abstract queries in this language are syntactically similar to SQL queries except that filter predicates are replaced with *holes* that can be instantiated with any valid predicate. Since operators in abstract queries are no longer parameterized by predicates, the search space of abstract queries is significantly reduced than the original one. Furthermore, the language contains a set of evaluation rules: given an abstract query, the rules evaluate it into a table that over-approximates the results of all queries that can be instantiated from it, allowing us to prune the search space earlier.

With this abstraction, we decompose the SQL synthesis problem into two phases. In the first phase, we search for abstract queries that can potentially be instantiated into SQL queries that satisfy the given I/O examples, and we prune away others based on their evaluation result. Then, in the second phase, we search for proper predicates for each synthesized abstract query to instantiate it into desired SQL queries and return top candidates to the user. To make the predicate search process efficient, we cluster predicates into semantically equivalence classes and encode tables using bit-vectors.

We implemented our algorithm in a PBE tool called Scythe. To evaluate Scythe, we collected 165 real-world benchmarks from Stack Overflow and 28 benchmarks from previous

work [218]. Results showed that Scythe quickly and precisely solved more than 74% of the benchmarks and 80% out of these solved benchmarks were solved within a few seconds. Our algorithm solved 51 more cases within 600 seconds compared to the enumerative search algorithm [186, 138] and outperformed SQLSynthesizer [218] on their project benchmarks with 4 more cases solved.

In sum, our paper makes the following contributions:

- We present a new approach to decompose the SQL query synthesis problem. Our key innovation is the design of the language of abstract queries and rules to evaluate them. (Section 3.4)
- We describe efficient synthesis algorithms optimized using properties of abstract queries to solve the two subproblems after decomposition, i.e., searching for abstract queries and instantiating them. (Section 3.5)
- We implemented our algorithm in a PBE system Scythe and evaluated it on 193 real-world benchmarks. Results show that Scythe makes substantial improvement compared to the enumerative search algorithm and other prior tools for synthesizing SQL queries. (Section 3.6)

3.2 Overview

We first demonstrate our algorithm with a running example.

Problem Statement. We formalize a user's query as a triple (I, T_{out}, C), where $I = \{T_1, ..., T_n\}$ stands for input tables, T_{out} stands for the output table, and $C = \{v_1, ..., v_k\}$ stands for a set of predicate constants. We seek to synthesize a query q such that $q(I) = T_{\text{out}}$ with the additional constraint that all constants used in predicates in q must come from C.

Note two special characteristics of our problem formalization that are tailored to relational queries in constrast to PBE systems in other domains [71, 162, 199]. First, we ask users to provide constants that will be used in predicates to make the problem less ambiguous as well as to boost the search efficiency. Based on our study of Stack Overflow questions, we find that users are usually willing to provide such constants along with table examples (e.g., return values that are between dates "12/24" and "12/25") because failing to provide them would result in a degree of ambiguity that must be resolved in a dialogue with experts.

Second, our problem formulation allows only one I/O example pair (I, T_{out}): the rationale is that users tend to resolve the ambiguities in the provided example by revising the example rather than creating a completely new one, so our algorithm needs to accept only one I/O pair.

Running Example. We combine two Stack Overflow posts into a running example to demonstrate the full features of our algorithm.[1] This example contains two input tables $I = \{T_1, T_2\}$, an output table T_{out} (as shown below), and constants {"12/25", "12/24", 50}.

T_1

id	date	uid
1	12/25	1
2	11/21	1
4	12/24	2

T_2

oid	val
1	30
1	10
1	10
2	50
2	10

T_{out}

c_0	c_1	c_2	c_3	c_4
1	12/25	1	1	30
4	12/24	2	2	10

The Solution. Figure 3.1 shows one correct solution. The query contains three steps. It (1) selects the rows in T_1 whose `date` field is `"12/25"` or `"12/24"`, (2) groups T_2 on `oid` and calculates the maximum `val` below `50` for each group, and (3) joins the results computed from steps 1 and 2 using the predicate "`uid = oid`". Note the use of the provided constants as part of the selection predicates in this case.

```
Select id, date, uid, oid, maxVal
From (Select * From T1
      Where T1.date = "12/24" Or T1.date = "12/25") As T3
Join (Select oid, Max(val) As maxVal
      From (Select * From T2 Where T2.val < 50) As T4
      Group By oid) As T5
On T3.uid = T5.oid
```

Figure 3.1: A solution for the running example.

The Subset of SQL Used in This Section. To focus on the key ideas without loss of generality, we restrict our synthesis algorithm to consider only a subset of SQL operators: selection, join, and aggregation (Figure 3.2); other features such as projection and union are discussed later in Section 3.3.

3.2.1 Enumerative Search with Equivalence-Class

Before explaining our algorithm, we first discuss the enumerative search approach, which is a class of widely adopted algorithms used to solve many synthesis problems. Such algorithms enumerate all programs in the program space of a given depth limit and retain only those consistent with the provided I/O examples. Previous enumerative synthesizers [138, 3, 186]

[1] http://stackoverflow.com/questions/39761697,
http://stackoverflow.com/questions/14995024

```
--select                --join                --aggregation
Select *                Select *              Select   c, α(c_t)
From   q                From   q1             From     q
Where  f                Join   q2             Group By c
                        On     f              Having   f
```

Figure 3.2: A subset of SQL grammar; q refers to an input table or a nested subquery, f refers to a predicate, α refers to an aggregation function, and c refers to a column name.

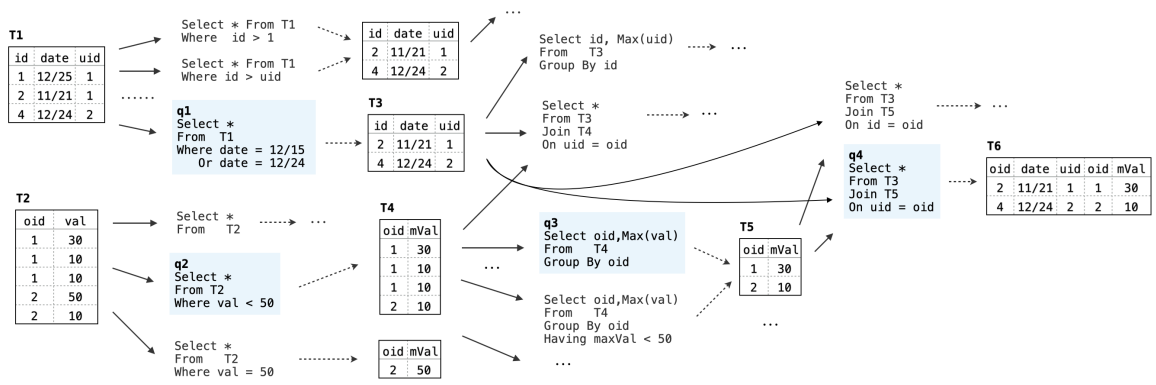


Figure 3.3: Equivalence-class based enumerative search algorithm, the subtree (queries in dash boxes) from T_1, T_2 to T_6 corresponds to the solution shown in Figure 3.1.

adopt the concept of equivalence classes to optimize the search process: they group programs into equivalence classes based on their behaviors on the input example to compress the search space. The search then proceeds iteratively: within each stage, the algorithms enumerate all programs in the current search space, group them based on an equivalence metric, and proceed to the next stage. Assume that r is the average reduction rate from programs to values per stage; the total reduction rate at stage d is r^d, which makes the algorithms much more scalable than simple enumeration.

Figure 3.3 illustrates how this technique is applied to SQL query synthesis to solve the running example. Queries are grouped into equivalence classes based on their evaluation results on the input example, and each iteration enumerates all queries that can be constructed from these equivalence class representatives using an operator from Figure 3.2. However, when applied to SQL, this algorithm performs inefficiently because grouping queries into equivalence classes cannot effectively compress the search space. There are a number of reasons for this. First, the main complexity of query synthesis comes from the generation of a large number queries per-stage rather than a large number of stages. For instance, the number of intermediate queries generated at the last stage of Figure 3.3 is 554,856 even if

we only consider the grammar shown in Figure 3.2. More importantly, grouping queries into equivalence classes represented by tables cannot effectively reduce search complexity: since tables are compound values that may contain thousands of cells (e.g., tables evaluated from nested `Join`s), evaluating queries and memoizing tables during the search process both contribute to large algorithmic overhead.

3.2.2 Our Approach

Our key insight for the challenges is to design a language of abstract queries to break down the synthesis process. Abstract queries resemble SQL queries except that they can contain uninstantiated filter predicates in the form of *holes*. This language lets us decompose the original synthesis problem into the following two subproblems that can be efficiently solved:

1. Synthesizing all abstract queries that can *potentially* be instantiated into queries satisfying the given I/O examples and pruning away the rest.

2. Searching for predicates to fill holes in these abstract queries, instantiating them into concrete ones, and determining which ones are consistent with the I/O examples.

We next describe the language of abstract queries and how this decomposition makes the synthesis problem tractable.

The Language of Abstract Queries

The grammar for abstract queries resembles the SQL grammar in Figure 3.2, except that all filter predicates (in `Where`, `Having`, `On` clauses) are replaced with holes "□" (we use $\tilde{q}$ to refer to abstract queries). As in SQL, abstract queries can also be composed (as in the case of `Join`).

Evaluating abstract queries is similar to evaluating SQL queries. For instance, an abstract `Select` or `Join` query is evaluated as a SQL query with its predicate hole replaced with `true`. We define the formal evaluation rules in Section 3.4. All evaluation rules satisfy the following *over-approximation property*: asumme $\tilde{q}$ is an abstract query; then, for any concrete query q instantiated from $\tilde{q}$, i.e., with all holes replaced with any syntactically valid predicates, the result of q is contained in the result of $\tilde{q}$. Thus, any abstract query whose result does not contain the output example will not lead to a valid query and can be pruned.

Problem 1: Searching for Valid Abstract Queries

We first search for abstract queries whose evaluation results contain the output example via enumerative search. Figure 3.4 shows the search process for the running example. The search process is similar to that shown in Figure 3.3, yet with different grammar and evaluation rules. Table $\tilde{T}_6$ in the figure contains the output example, so that the tree of queries from

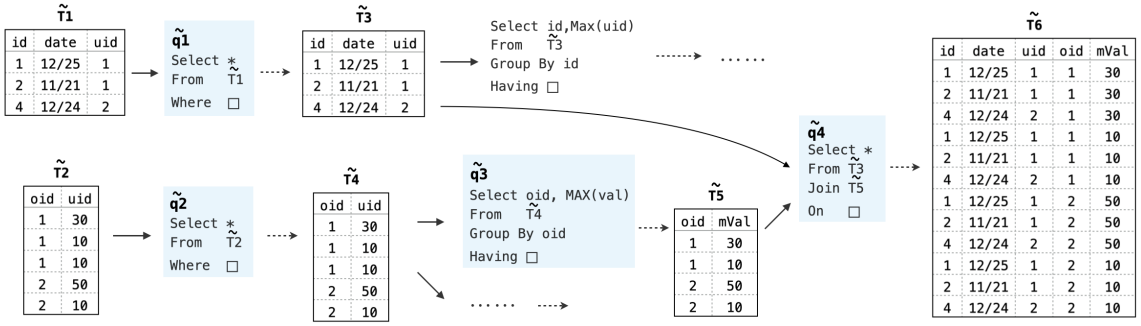


Figure 3.4: Searching for candidate abstract queries, where dash line arrows show evaluation of abstract queries. The tree from $\tilde{T}_1, \tilde{T}_2$ to $\tilde{T}_6$ corresponds to a candidate abstract query. Note the reduction in the number of tables and queries as compared to Figure 3.3.

input tables to $\tilde{T}_6$ forms a candidate abstract query. All abstract queries that do not contain T_{out} are removed.

Since abstract queries do not contain filter predicates, far fewer intermediate tables are generated: for the running example, only 105 different intermediate tables (in total 2,710 cells) are generated in the last stage of Figure 3.4 compared to 1,889 tables and 42,680 cells for the search process in Figure 3.3. Furthermore, the over-approximation property prunes as many as 90% of the abstract queries generated in the last stage.

Problem 2: Predicate Synthesis

Once candidate abstract queries are identified, we synthesize predicates to instantiate each of them. Specifically, for the running example, we need to find predicates for holes $\square^{a-d}$ (examples shown in Figure 3.5) to instantiate the abstract query to a SQL query whose evaluation result is T_{out}.

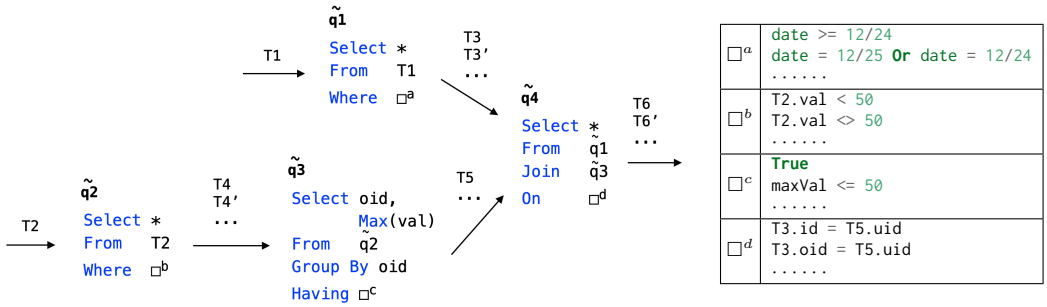


Figure 3.5: The abstract query in Figure 3.4 and candidate predicates for each hole.

This search remains highly challenging since: (1) the number of candidate predicates is huge (4,692 syntactically different predicates for $\square^c$ alone, due to the large number of compound predicates built from conjunction/disjunction/negation) and (2) evaluating and memoizing intermediate tables remain expensive. We use two optimizations to address these issues.

Locally grouping candidate predicates. First, for each subquery of the given abstract query, we group its candidate predicates into equivalence classes. The idea is that if two candidate predicates behave the same on the evaluation result of a given abstract subquery,[2] their behaviors on the whole abstract query remains identical, no matter how other holes in the query are instantiated. Hence, we need to retain only one such predicate as a representative of the equivalence class.[3]

For example, the two candidate predicates "`True`" and "`maxVal <= 50`" for $\square^c$ in $\tilde{q}_3$ in Figure 3.5 produce the same results when evaluated on $\tilde{T}_5$ in Figure 3.4; hence, only one of them needs to be retained to reduce search complexity. In total, the 4,692 candidate predicates for the hole $\square^c$ in Figure 3.5 are grouped into 21 equivalence classes, with a reduction rate of 200$\times$ compared to direct enumeration.

Encoding tables using bit-vectors. Our second optimization encodes intermediate tables using bit-vectors to improve search efficiency. The insight stems from the over-approximation property of evaluating abstract queries. Because of that, when searching for the instantiation of an abstract query $\tilde{q}$, we can use its abstract evaluation result $\tilde{T}$ together with a bit-vector β to represent the evaluation result of every instantiation of $\tilde{q}$: the size of β is same as the number of rows in $\tilde{T}$, and the i-th bit in β represents whether the the row i in $\tilde{T}$ appears in T.

For example, table T_4 in Figure 3.3 (evaluated from q_2) can be represented using the pair $(\tilde{T}_4, [11101])$ ($\tilde{T}_4$ is the evaluation result of the abstract query $\tilde{q}_2$ in Figure 3.4), since rows 0,1,2,4 of $\tilde{T}_4$ appear in T_4. Likewise, $T_{\texttt{out}}$ can be represented as the pair $(\tilde{T}_6, [010000000001])$ as shown in Figure 3.4.

Encoding tables into bit-vectors has two benefits. First, the intermediate results of the predicate search process can be fully represented using bit-vectors to reduce the memoization overhead, since the tables evaluated from abstract queries are shared among many and bit-vectors are cheap to memoize. Second, we can optimize operators on queries into bit-vectors

[2]The behavior of a predicate on a table means evaluating the table and the predicate as a simple `Select` query.

[3]To ensure the algorithm's completeness, for each synthesized candidate query, our algorithm generates all different versions of the query by replacing predicate equivalence class representatives with all predicates in the group.

operators benefiting from this representation; since many bit-vector operators require no materialization of tables, the computation overhead is also significantly reduced, as we will show in Section 3.5.2.

With these optimizations, the predicate synthesis algorithm efficiently searches for instantiations of abstract queries that are consistent with the provided I/O example. These candidate queries are later ranked and returned to the user.

3.2.3 Ranking and Interaction

Since the provided I/O example often does not completely specify a task, our algorithm returns multiple candidate queries that are consistent with the example. We use a heuristic to rank [71, 162] the queries returned from the main synthesis algorithm based on simplicity, naturalness and constant coverage, as will be discussed in Section 3.5.3. We present top-ranked queries to the user, who can provide a new example to the system and re-run the tool if needed.

3.3 The SQL Language

We introduce the definition of tables and briefly review our target language SQL in this section.

Table. A table is a pair consisting of schema and content (Figure 3.6), where the schema is a list of name-type pairs and the content is a list of rows. Values we support include typed scalars or `null`. Additionally, as we adopt *bag-semantics* [130] for SQL, where duplicate rows are allowed in tables, and the equivalence between two tables is defined as bag equivalence, i.e., two tables are equal iff they mutually contain each other regardless of row ordering.

For readability, we use the notation $T_1 \subseteq T_2$ to represent that all rows in T_1 are contained by T_2, and the multiplicity of each row in T_1 is smaller or equal than its multiplicity in T_2.We also use $T_1 \cup T_2$ to refer to the union of contents in T_1 and T_2 (when their schema type are compatible); the schema of $T_1 \cup T_2$ is the same as T_1.

$$
\begin{array}{rcll}
T & ::= & \mathsf{Table}(schema, content) & \text{(Table)} \\
schema & ::= & [c_1 : \tau_1, ..., c_m : \tau_m] & \text{(Schema)} \\
content & ::= & [r_1, ..., r_n] & \text{(Content)} \\
r & ::= & [v_1, ..., v_m] & \text{(Row)} \\
\tau & ::= & \mathsf{int} \mid \mathsf{double} \mid \mathsf{string} \mid \mathsf{date} \mid \mathsf{time} & \text{(Type)}
\end{array}
$$

Figure 3.6: The definition of tables and auxiliary functions on tables. Metavariable c ranges over column names and v ranges over values.

$$
\begin{aligned}
q &::= T \mid \mathsf{Proj}(\bar{c}, q) \mid \mathsf{Dedup}(q) \mid \mathsf{Select}(q, f) \mid \mathsf{Join}(q_1, q_2, f) \mid \mathsf{Union}(q_1, q_2) \\
&\quad \mid \mathsf{Aggr}(\bar{c}, c_t, \alpha, q, f) \mid \mathsf{LeftJoin}(q_1, q_2, \bar{c} = \bar{c}') \mid \mathsf{Rename}(q, name, \bar{c}) \\
f &::= \mathsf{True} \mid v \ binop \ v \mid \mathsf{Exists}\ q \mid \mathsf{IsNull}\ c \mid f\ \mathsf{And}\ f \mid f\ \mathsf{Or}\ f \mid \mathsf{Not}\ f \\
v &::= c \mid const \mid \mathsf{null} \\
\alpha &::= \mathsf{max} \mid \mathsf{min} \mid \mathsf{avg} \mid \mathsf{count} \mid \mathsf{sum} \mid \mathsf{count\_distinct} \mid \mathsf{concat} \\
binop &::= = \mid > \mid < \mid \leq \mid \geq \mid \neq
\end{aligned}
$$

Figure 3.7: SQL grammar. The metavariable T ranges over tables, c ranges over column names, $const$ ranges over constant values, and $name$ ranges over fresh table names. Bar notation is used to represent repetitive elements.

$$
\begin{aligned}
\tilde{q} &::= T \mid \mathsf{Proj}(\bar{c}, \tilde{q}) \mid \mathsf{Dedup}(\tilde{q}) \mid \mathsf{Select}(\tilde{q}, \square) \mid \mathsf{Join}(\tilde{q}_1, \tilde{q}_2, \square) \mid \mathsf{Union}(\tilde{q}_1, \tilde{q}_2) \\
&\quad \mid \mathsf{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square) \mid \mathsf{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}') \mid \mathsf{Rename}(\tilde{q}, name, \bar{c}) \\
\alpha &::= \mathsf{max} \mid \mathsf{min} \mid \mathsf{avg} \mid \mathsf{count} \mid \mathsf{sum} \mid \mathsf{count\_distinct} \mid \mathsf{concat}
\end{aligned}
$$

Figure 3.8: The grammar of abstract queries. The symbol "□" refers to an uninstantiated predicate, and the metavariable c ranges over column names.

***SQL.*.** Figure 3.7 presents the grammar of SQL: a query q is formed by one of Projection, Dedup, Select, Join, Aggr, LeftJoin, Union or Rename constructors. The constructor $\mathsf{Aggr}(\bar{c}, c_t, \alpha, q, f)$ corresponds to the aggregation query "`Select` $\bar{c}, \alpha(c_t)$ `From` q `Group By` $\bar{c}$ `Having` f", the constructor $\mathsf{Proj}(\bar{c}, q)$ corresponds to the projection query `Select` $c_1, ..., c_n$ `From` q, and others can be directly mapped to their concrete forms.

We omit the evaluation rules for SQL in our formal definition due to space limitations. We use the notation $[\![q]\!]$ to denote evaluating the query q into a table.

3.4 The Language of Abstract Queries

The language of abstract queries is key to the decomposition of the query synthesis problem. Figure 3.8 presents its grammar: an abstract query in the language is similar to a concrete SQL query except that filter predicates are replaced by uninstantiated holes (□). Abstract queries and concrete queries have the following *instantiation/abstraction* relation.

Definition 1. *(Instantiation and Abstraction)* Given an abstract query $\tilde{q}$ and a query q, we call q an instantiation of $\tilde{q}$, if there exists a substitution $\phi = \{\square_1 \mapsto f_1, ..., \square_k \mapsto f_k\}$ (where k is the number of holes in $\tilde{q}$), such that substituting holes in $\tilde{q}$ with ϕ results in q, i.e., $\tilde{q}/\phi = q$. We also call $\tilde{q}$ the abstraction of q (by definition, only one abstraction exists for a query q).

Evaluation Rules. Figure 3.9 shows evaluation rules for abstract queries, and the over-approximation property for these rules (as presented in Section 3.2.2) is formally defined below (Property 6). These rules are designed to ensure the satisfaction of the over-approximation property:

- Evaluating a table results in the table itself.
- When evaluating an abstract Join or Select query, the result is obtained by evaluating it with the predicate True.
- Given an abstract LeftJoin query, we first compute the evaluation results T_1, T_2 of its abstract subqueries and then return the union of (1) the left join result of T_1, T_2 and (2) the left join result of T_1 and an empty table whose schema is the same as T_2's. The second part ensures that the over-approximation property is satisfied no matter how $\tilde{q}_2$ is instantiated.
- Given an abstract Aggr query, we first evaluate the inner abstract subquery into a table T; we next compute the aggregation result for *all* tables that are contained by T with Having clause set to True, and we finally union the results. We must consider all possibilities in the rule to ensure the over-approximation property, since the grouping result is dependent to how its abstract subquery is instantiated.
- Evaluation rules for other abstract queries resemble their concrete version, since the over-approximation property propagates automatically from their subqueries.

Complexity and Optimization. We measure the complexity of evaluating an abstract query using the number of SQL operators executed in the evaluation process and the output table's size. Assume the abstract queries are evaluated on an input table (or input tables) with r rows and c columns. From the rules in Figure 3.9, the worst-case measures for evaluating abstract Aggr queries are both exponential in r since we must compute the aggregation result of all tables contained by the input table and union the results, while the worst-case measures for all other abstract queries are polynomial in $r \times c$. Thus, the bottleneck of evaluating abstract queries lies in evaluating Aggr subqueries in an abstract query.

Fortunately, we can optimize the evaluation rules for abstract Aggr queries formed with many commonly used aggregation functions, including max, min, count and count_distinct to avoid the bottleneck. Since max and min return only existing values from the input tables, the output table size of an abstract Aggr query that uses such aggregates is polynomial to the size of its input table. This property lets us to simplify the evaluation rule into $[\![\mathsf{Dedup}(T)]\!]$ (where T is the evaluation result of the inner abstract subquery) without violating the over-approximation property. An example is the evaluation of $\tilde{q}_3$ to $\tilde{T}_5$ in Figure 3.4. Similarly, the

$$\widetilde{eval}(T) = T \qquad \widetilde{eval}(\mathsf{Proj}(\bar{c}, \tilde{q})) = [\![\mathsf{Proj}(\bar{c}, \widetilde{eval}(\tilde{q}))]\!]$$

$$\widetilde{eval}(\mathsf{Dedup}(\tilde{q})) = [\![\mathsf{Dedup}(\widetilde{eval}(\tilde{q}))]\!] \qquad \widetilde{eval}(\mathsf{Select}(\tilde{q}, \square)) = [\![\mathsf{Select}(\widetilde{eval}(\tilde{q}), \mathsf{True})]\!]$$

$$\widetilde{eval}(\mathsf{Join}(\tilde{q}_1, \tilde{q}_2, \square)) = [\![\mathsf{Join}(\widetilde{eval}(\tilde{q}_1), \widetilde{eval}(\tilde{q}_2), \mathsf{True})]\!]$$

$$\widetilde{eval}(\mathsf{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square)) = \left[\!\!\left[\mathsf{Dedup} \left(\bigcup_{T \subseteq \widetilde{eval}(\tilde{q})} [\![\mathsf{Aggr}(\bar{c}, c_t, \alpha, T, \mathsf{True})]\!] \right) \right]\!\!\right]$$

$$\widetilde{eval}\,(\mathsf{Union}(\tilde{q}_1,\ \tilde{q}_2)) = [\![\mathsf{Union}(\widetilde{eval}(\tilde{q}_1), \widetilde{eval}(\tilde{q}_2))]\!]$$

$$\widetilde{eval}(\mathsf{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}')) = \mathsf{let}\ T_1 = \widetilde{eval}(\tilde{q}_1), T_2 = \widetilde{eval}(\tilde{q}_2), T_3 = [\![\mathsf{Select}(T_2, \mathsf{False})]\!]\ \mathsf{in}$$
$$[\![\mathsf{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!] \cup [\![\mathsf{LeftJoin}(T_1, T_3, \bar{c} = \bar{c}')]\!]$$

$$\widetilde{eval}(\mathsf{Rename}(\tilde{q}, name, \bar{c})) = [\![\mathsf{Rename}(\widetilde{eval}(\tilde{q}), name, \bar{c})]\!]$$

Figure 3.9: The evaluation rules for abstract SQL. Notation $[\![q]\!]$ refers to evaluating a concrete query based on SQL semantics.

only new values produced by count and count_distinct are column counts, and the evaluation rules for abstract queries containing them can likewise be simplified.

Property 6. *(Over Approximation)* Given an abstract query $\tilde{q}$ and a query q instantiated from $\tilde{q}$, we have $[\![q]\!] \subseteq \widetilde{eval}(\tilde{q})$.

Proof Sketch. By induction on the abstract query constructors, we can prove that every row in any instantiation of the abstract query is contained in its evaluation result. □

3.5 Synthesis Algorithm

We now introduce our synthesis algorithm (Algorithm 3). Given an example containing input tables I, output table T_{out}, and a set of constants C, the Synthesis algorithm constructs a set of candidate queries within the given time limits. For each $depth$, the algorithm first searches for all abstract queries that can potentially be instantiated into candidate queries (line 5); then, for each synthesized abstract query $\tilde{q}$, it constructs all of instantiations of $\tilde{q}$ that are consistent with the I/O example (lines 6,7); finally, the algorithm selects all synthesized queries whose score is beyond a system predefined threshold, and returns candidates to the user after ranking (lines 9,10).

Algorithm 3 The main synthesis algorithm.

```
1:  procedure Synthesis(I, T_out, C)
2:      input: input tables I, output table T_out, constants C
3:      output: Queries that are consistent with the input-output examples.
4:      depth ← 1;
5:      while timeout() = false do
6:          S_q ← ∅;
7:          S_q̃ ← SynthesizeAbstractQuery(I, T_out, depth);
8:          for all q̃ ∈ S_q̃ do
9:              S_q ← S_q ∪ PredSynthesis(q̃, I, T_out, C);
10:         candidates ← {q | q ∈ S_q ∧ Score(q) > threshold};
11:         if candidates ≠ ∅ then
12:             return Rank(candidates);
13:         depth ← depth + 1;
14:     return ∅;
```

3.5.1 Abstract Query Synthesis

The first part of the algorithm is abstract query synthesis, the goal of which is presented by the completeness condition below.

Definition 2. *(Completeness Condition)* Given an example (I, T_{out}, C) and search depth d, suppose $S_{\tilde{q}}$ is the set of abstract queries returned by the abstract query synthesis algorithm, then for all q in the query space that are consistent with the input output example whose length is with d, its corresponding abstract query $\tilde{q}$ is contained in $S_{\tilde{q}}$.

Our algorithm achieves this goal with the help of the over-approximation property of the abstract queries: as long as every abstract query whose evaluation result contains T_{out} is included in the result, our algorithm will not miss any abstract queries that can potentially be instantiated into queries that are consistent with the I/O example.

The algorithm (Algorithm 4) adopts an enumerative search approach for all abstract queries satisfying this condition: starting from the input tables I with depth $d = 1$, the algorithm iteratively (1) enumerates all abstract queries can be constructed from tables in S_T (by iterating over all query constructors for all tables in S_T) (line 4), (2) maintains a mapping between the abstract evaluation result of these abstract queries and their syntactical form using the map M (line 5), and (3) updates the S_T with newly generated tables (line 6). When the given search depth is reached, on lines 8-9, the algorithm retrieves all tables that fully contains T_{out} and decodes them into abstract queries with the help of the mapping M (by recursively substituting intermediate tables with their corresponding abstract subqueries). At the end of the phase, the algorithm returns a set of abstract queries.

The enumerative search approach can be applied efficiently in this phase since the size of the abstract query space is much smaller than that of the concrete query space, as all predicates are kept as holes. On the other hand, Algorithm 4 has one bottleneck: the number of tables enumerated at each stage is exponential to the maximum column size of input tables since our algorithm enumerates all possible grouping by columns for each aggregation query to ensure completeness. Fortunately, the input examples provided by the user are typically reasonably small and a majority of these abstract queries are immediately pruned in this phase if they cannot be instantiated into candidate queries. As a result, the number of abstract queries sent to the second phase algorithm is sufficiently small to ensure that the overall algorithm runs efficiently.

Algorithm 4 The abstract query synthesis algorithm; the subroutine EnumOneStepAbstractQuery enumerates all abstract queries that can be directly constructed table(s) in S_T.

1: **procedure** SynthesisAbstractQuery($I, T_{\text{out}}, depth$)
2: **input:** input output tables (I, T_{out}), search depth ($depth$).
3: **output:** all abstract queries constructed from I within depth $depth$, whose evaluation result fully contains T_{out}.
4: $d \leftarrow 1$, $S_T \leftarrow I$, $M \leftarrow \emptyset$
5: **while** $d \leq depth$ **do**
6: $S_{\tilde{q}} \leftarrow S_{\tilde{q}} \cup \mathsf{EnumOneStepAbstractQuery}(S_T)$
7: $M \leftarrow M \cup \{(T, \tilde{q}) \mid T = \widetilde{eval}(\tilde{q}) \wedge \tilde{q} \in S_{\tilde{q}}\}$
8: $S_T \leftarrow \{T \mid \tilde{q} \in S_{\tilde{q}} \wedge T = \widetilde{eval}(\tilde{q})\}$
9: $d \leftarrow d + 1$
10: $candidates \leftarrow \{T \mid T \in S_T \wedge T_{\text{out}} \subseteq T\}$
11: **return** DecodeToAbstractQuery($candidates, M$)

Lemma 1. Algorithm 2 is complete (Definition 2).

Proof Sketch. This condition is guaranteed since (1) if q is a query consistent with the I/O example, its evaluation result contains the output example according to Property 6, and (2) Algorithm 4 searches for every abstract queries whose evaluation result contains T_{out}. □

3.5.2 Predicate Synthesis

Given an abstract query synthesized by the previous algorithm, the predicate synthesis algorithm synthesizes predicates for the abstract query to instantiate it into candidate queries. We first present a simple (but inefficient) algorithm that can solve this problem (Algorithm 5). First, the simple algorithm searches (with memoization, as in Algorithm 4) for all tables that can be obtained from queries instantiated from the abstract query $\tilde{q}$, by enumerating all predicates that can be filled into the predicate holes (line 1). Then, if the

$$\mathsf{DFS}(T) = \{T\} \qquad \mathsf{DFS}(\mathsf{Union}(\tilde{q}_1, \tilde{q}_2)) = \{[\![\mathsf{Union}(T_1, T_2)]\!] \mid T_i \in \mathsf{DFS}(\tilde{q}_i)\}$$

$$\mathsf{DFS}(\mathsf{Proj}(\bar{c}, \tilde{q})) = \{[\![\mathsf{Proj}(\bar{c}, T)]\!] \mid T \in \mathsf{DFS}(\tilde{q})\} \qquad \mathsf{DFS}(\mathsf{Dedup}(\tilde{q})) = \{[\![\mathsf{Dedup}(T)]\!] \mid T \in \mathsf{DFS}(\tilde{q})\}$$

$$\mathsf{DFS}(\mathsf{Select}(\tilde{q}, \square)) = \{[\![\mathsf{Select}(T, f)]\!] \mid T \in \mathsf{DFS}(\tilde{q}) \wedge f \in \mathsf{EnumAllPreds}(\mathsf{Select}(\tilde{q}, \square), I, C)\}$$

$$\mathsf{DFS}(\mathsf{Join}(\tilde{q}_1, \tilde{q}_2, \square)) = \{[\![\mathsf{Join}(T_1, T_2, f)]\!] \mid T_i \in \mathsf{DFS}(\tilde{q}_i) \wedge f \in \mathsf{EnumAllPreds}(\mathsf{Join}(\tilde{q}_1, \tilde{q}_2, \square), I, C)\}$$

$$\mathsf{DFS}(\mathsf{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square)) = \{[\![\mathsf{Aggr}(\bar{c}, c_t, \alpha, T, f)]\!] \mid T \in \mathsf{DFS}(\tilde{q}) \wedge f \in \mathsf{EnumAllPreds}(\mathsf{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square), I, C)\}$$

$$\mathsf{DFS}(\mathsf{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}')) = \{[\![\mathsf{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!] \mid T_i \in \mathsf{DFS}(\tilde{q}_i)\}$$

$$\mathsf{DFS}(\mathsf{Rename}(\tilde{q}, name, \bar{c})) = \{[\![\mathsf{Rename}(T, name, \bar{c}))]\!] \mid T \in \mathsf{DFS}(\tilde{q}_i)\}$$

Figure 3.10: The DFS algorithm searching for all tables that can be evaluated from queries instantiated from an abstract query $\tilde{q}$. C and I refer to constants and input tables from the user. The function EnumAllPreds enumerates all candidate predicates for the given abstract query.

output table is found in the search process, the algorithm generates queries from the output table, according to its memoization result (function GenQuery in line 3). Algorithm 3.10 shows the enumeration rules. The function EnumAllPredicates in the rule enumerates all possible syntactically different filter predicates for Select, Join and Aggr abstract queries, by iterating over all valid predicates defined by the SQL grammar (Figure 3.7).

Algorithm 5 A simple predicate synthesis algorithm.

```
1: procedure SimplePredSynthesis(q̃, I, T_out, C)
2:     input: an abstract query q̃, input output example I, T_out, C.
3:     output: candidate queries instantiated from q̃.
4:     S_T ← DFS(q̃, I, C);
5:     if T_out ∈ S_T then
6:         return GenQuery(T_out);
7:     return ∅;
```

Though simple, without any optimization, this algorithm is prohibitively expensive to be used in practice due to the challenges that arise from: (1) the large number of filter predicates to be searched, and (2) the expensive process of evaluating and memoizing tables during the search process.[4]

Our algorithm takes advantages of the over-approximation properties of abstract queries to speed up the algorithm by (1) locally grouping predicate candidates into equivalence

[4]Not doing so will make algorithm even less efficient as the number of different programs in the space is several magnitudes more than the number of their evaluation results.

classes for each hole to reduce the number of predicates to be enumerated and (2) encoding intermediate results into bit-vectors to increase search efficiency.

Predicate Enumeration and Grouping. Given an abstract query $\tilde{q}$ constructed from Select, Join or Aggr that contain a predicate hole, the predicate enumeration algorithm (Algorithm 6) enumerates all filter predicates that can be filled into the hole of $\tilde{q}$ (lines 4, 10), groups them into equivalence classes based on their behavior on the evaluation result of $\tilde{q}$, (i.e., $\widetilde{eval}(\tilde{q})$) (lines 7-8, lines 11-12), and returns the representatives of all predicate groups.

The reason we can group these candidate predicates without losing completeness of the filter behaviors is shown below in Property 7. The key idea is that filter predicates in each equivalence class behaves indistinguishably in all possible instantiations of $\tilde{q}$ (no matter how subqueries of $\tilde{q}$ are instantiated).

Property 7. *(Predicate Equivalence)* Let $\tilde{q}$ be an abstract query formed by one of Select, Join, Aggr constructor with a hole $\square_0$, $T = \widetilde{eval}(\tilde{q})$, and f_1, f_2 be two predicate candidates for $\square_0$ that are equivalent on T, i.e., $[\![\mathsf{Select}(T, f_1)]\!] = [\![\mathsf{Select}(T, f_2)]\!]$. Then, for any ϕ, a substitution of holes in subqueries in $\tilde{q}$, we have $[\![\tilde{q}/(\phi \circ \{\square_0 \mapsto f_1\})]\!] = [\![\tilde{q}/(\phi \circ \{\square_0 \mapsto f_2\})]\!]$.

Proof Sketch. First, we have $T_0 = [\![\tilde{q}/(\phi \circ \{\square_0 \mapsto \mathsf{True}\})]\!] \subseteq T$ according to the over-approximation property of abstract evaluation. Since f_1, f_2 are equivalent on T, they are also equivalent on any table that is contained by T, i.e., they are also equivalent on T_0, so that the equation is satisfied. □

With this property, instead of needing to search all predicates from EnumAllPredicates as in Algorithm 3.10, we only need to search predicate representatives returned by EnumAndGroupPred, which reduces the search space size. Note that when candidate queries are synthesized. We also expand representatives to all predicates in their equivalence classes to obtain different versions of the candidates, which ensures the completeness of syntactically different queries.

Encoding Tables into Bit-Vectors. The second optimization is to encode of intermediate results in the search process to avoid the expensive computation and memoization caused by inefficient table representation. Suppose q is a query instantiated from an abstract query $\tilde{q}$, $T_0 = [\![q]\!]$ and $T = \widetilde{eval}(\tilde{q})$; according to Property 6, we have $T_0 \subseteq T$, so that we can represent T_0 as a bit-vector based on T. As shown in Figure 3.11, we use a bit-vector β of length $\mathsf{rowNum}(T)$ to represent T_0: we mark the i-th bit b_i as 1, if row i of T is also a row in T_0 (the second condition ensures that duplicates are correctly handled). Also, we can decode a bit-vector β of length $\mathsf{rowNum}(T)$ into a table together with T, and the result is a table that contains only rows whose index bit is marked as '1' in T.

With this encoding, we reduce the original predicate synthesis problem (Algorithm 5) into a search problem whose intermediate results are bit-vectors, (Algorithm 7): given an

Algorithm 6 Predicate Enumeration Algorithm. The function EnumPrimitivePred enumerates primitive predicates using given values V and tables (tables are used for enumerating Exists predicates) and the function EnumCompoundPred generates compound predicates (and, or, not) from given predicates.

1: **procedure** EnumAndGroupPred($\tilde{q}$, $Const$, I)
2: **input:** an abstract subquery $\tilde{q}$, constants $Const$, input tables I.
3: **output:** representative predicates.
4: $T \leftarrow \widetilde{eval}(\tilde{q})$;
5: $V \leftarrow \mathsf{schema}(T) \cup Const$;
6: $primitives \leftarrow \mathsf{EnumPrimitivePred}(V, I)$;
7: $rep \leftarrow \emptyset$;
8: **for** $p \in primitives$ **do**
9: **if** $\nexists f \in rep.(\llbracket \mathsf{Select}(T, f) \rrbracket = \llbracket \mathsf{Select}(T, p) \rrbracket)$ **then**
10: $rep \leftarrow rep \cup \{p\}$;
11: $compound \leftarrow \mathsf{EnumCompoundPred}(rep)$;
12: **for** $p \in compound$ **do**
13: **if** $\nexists f \in rep.(\llbracket \mathsf{Select}(T, f) \rrbracket = \llbracket \mathsf{Select}(T, p) \rrbracket)$ **then**
14: $rep \leftarrow rep \cup \{p\}$
15: **return** rep

abstract query $\tilde{q}$, our goal is to search over the space of bit-vectors that encodes instantiations of $\tilde{q}$ for ones that can be decoded to T_{out}.

Besides making memoization more efficient, this reduction also brings us the opportunity to simplify the computation shown in Algorithm 3.10, since many operators on table can be simplified into operators on bit-vectors that require no materialization of the tables (Figure 3.11). For example, when computing a bit-vector representation of a Join query result, we do not need to instantiate the Cartesian product table; instead, we only need to merge bit-vectors from its subqueries using the bit-wise crossproduct operator shown in Figure 3.11.

Additionally, our algorithm also conducts pruning in the bit-vector decoding process (line 3 of Algorithm 7) using output table T_{out}. Our algorithm precomputes a set S containing all bit-vectors that encode T_{out} based on the evaluation result of $\tilde{q}$; thus, we only need to check whether a bit-vector is in S to determine whether it leads to a candidate, which avoids the materialization of all tables in the last step.

Algorithm 7 The Predicate Synthesis Algorithm.

1: **procedure** PredSynthesis($\tilde{q}, I, T_{out}, Const$)
2: $B \leftarrow \mathsf{BVDFS}(\tilde{q}, I, Const)$
3: $T \leftarrow \widetilde{eval}(\tilde{q})$
4: $candidates \leftarrow \{\beta \mid \beta \in B \wedge \mathsf{Decode}(\beta, T) = T_{out}\}$
5: **return** $\mathsf{GenQuery}(candidates)$

$$\mathsf{Encode}(T_0, T) = [b_1, ..., b_n]_n \ \textit{where}\ n = \mathsf{rowNum}(T),\ T_0 \subseteq T$$

$$b_i = \begin{cases} 1 & \textit{if}\ T[i] \in T_0\ \textit{and}\ \mathsf{occur}(T[i], T[1, ..., i-1]) < \mathsf{occur}(T[i], T_0) \\ 0 & \textit{otherwise} \end{cases}$$

$$\mathsf{Decode}([b_1, ..., b_n]_n, T) = \mathsf{Table}(\mathsf{schema}(T), [r_i \mid r_i \in T \wedge b_i = 1])\ \textit{where}\ n = \mathsf{rowNum}(T), i \in [1, n]$$

$$[b_1, ..., b_n]_n\ \&\ [b'_1, ..., b'_n]_n = [b_1\ \&\ b'_1, ..., b_n\ \&\ b'_n]$$

$$[b_1, ..., b_n]_n +\!\!+\ [b'_1, ..., b'_m]_m = [b_1, ..., b_n, b'_1, ..., b'_m]$$

$$[b_1, ..., b_n]_n \times [b'_1, ..., b'_m]_m = [c_{i*m+j}\ | c_{i*m+j} = b_{i+1}\ \&\ b_j]_{m \times n}\ \textit{where}\ i \in [0, n-1], j \in [1, m]$$

Figure 3.11: Bit-vector operators, where $\&$ refers to bit-'and', $+\!\!+$ refers to list concatenatior and $\times$ refers crossproduct operator.

The correctness of the search algorithm is ensured by the property below. The property suggests that given an abstract query $\tilde{q}$, if a table T can be found by the original algorithm, the new algorithm that operates on bit-vectors is also able to find a bit-vector whose decoding result is T, so that no table that the original algorithm found is missed after optimization.

Property 8. *(Encoding Soundness)* Given an abstract query $\tilde{q}$, DFS and BVDFS returns the same set of output tables, i.e., $\mathsf{DFS}(\tilde{q}) = \{\mathsf{Decode}(\beta, \widetilde{eval}(\tilde{q})) \mid \beta \in \mathsf{BVDFS}(\tilde{q})\}$.

Proof Sketch. The proof can be achieved by induction on constructors of $\tilde{q}$: for each abstract query constructor, we can prove that for each table T in $\mathsf{DFS}(\tilde{q})$ there exists at least one bit-vector $\beta \in \mathsf{BVDFS}(\tilde{q})$ whose decoding result on $\widetilde{eval}(\tilde{q})$ is T. □

At the end of this phase, our algorithm returns all possible instantiations of candidate abstract queries that are consistent with the I/O example. These queries are passed to the ranking and user interaction phase of our algorithm.

We present the main theorem (completeness property) of our synthesis algorithm below.

Theorem 1. *(Completeness)* Given an example (I, T_{out}, C), suppose q is a query consistent with the provided I/O example with subquery depth is d; then, given unlimited timeout, Algorithm 3 can find q in the d-th iteration.

Proof Sketch. This theorem is the direct conclusion of Lemma 4 and Properties 7,8: the former ensures that the abstract query $\tilde{q}$ corresponding to q is included in the result at depth d of Algorithm 4, and the latter two ensure that the two optimizations do not change the result of the predicate enumeration algorithm (Algorithm 5). □

$$\mathsf{BVDFS}(T) = \mathsf{Encode}(T, T) \qquad \mathsf{BVDFS}(\mathsf{Proj}(\bar{c}, \tilde{q})) = \mathsf{BVDFS}(\tilde{q})$$

$$\mathsf{BVDFS}(\mathsf{Dedup}(\tilde{q})) = \{\mathsf{Encode}([\![\mathsf{Dedup}(\mathsf{Decode}(\beta, T))]\!], T) \mid \beta \in \mathsf{BVDFS}(\tilde{q})\} \;\; \textit{where } T = \widetilde{eval}(\mathsf{Dedup}(\tilde{q}))$$

$$\mathsf{BVDFS}(\mathsf{Select}(\tilde{q}, \square)) = \{\beta \& \beta_1 \;\mid\; \beta \in \{\mathsf{FiltersToBV}(\mathsf{Select}(\tilde{q}, \square), f) \mid f \in F\} \wedge \beta_1 \in \mathsf{BVDFS}(\tilde{q})\}$$

$$\mathsf{BVDFS}(\mathsf{Join}(\tilde{q}_1, \tilde{q}_2, \square)) = \{(\beta_1 \times \beta_2) \& \beta \mid \beta_i \in \mathsf{BVDFS}(\tilde{q}_i) \wedge \beta \in \{\mathsf{FiltersToBV}(\mathsf{Join}(\tilde{q}_1, \tilde{q}_2, \square), f) \mid f \in F\}\}$$

$$\begin{aligned}&\mathsf{BVDFS}(\mathsf{Aggr}(\bar{c}, \widetilde{c_t, \alpha}, \tilde{q}, \square)) = \{\beta_1 \& \beta \mid \beta_1 \in B_1 \wedge \beta \in B\} \\ &\quad \textit{where } T = \widetilde{eval}(\mathsf{Aggr}(\bar{c}, c_t, \alpha, \tilde{q}, \square)),\; B = \{\mathsf{FiltersToBV}(\mathsf{Aggr}(\bar{c}, \widetilde{c_t, \alpha}, \tilde{q}, \square), f) \mid f \in F\} \\ &\qquad B_1 = \{\mathsf{Encode}([\![\mathsf{Aggr}(\bar{c}, c_t, \alpha, t, \mathsf{true})]\!], T) \mid t \in \{\mathsf{Decode}(\beta, \widetilde{eval}(\tilde{q})) \mid \beta \in \mathsf{BVDFS}(\tilde{q})\}\}\end{aligned}$$

$$\mathsf{BVDFS}(\mathsf{Union}(\tilde{q}_1, \tilde{q}_2)) = \{\beta_1 +\!\!+\; \beta_2 \mid \beta_1 \in \mathsf{BVDFS}(\tilde{q}_1) \wedge \beta_2 \in \mathsf{BVDFS}(\tilde{q}_2)\}$$

$$\begin{aligned}&\mathsf{BVDFS}(\mathsf{LeftJoin}(\widetilde{\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}'})) = \{\mathsf{Encode}([\![\mathsf{LeftJoin}(T_1, T_2, \bar{c} = \bar{c}')]\!], T) \mid T_i \in S_{Ti}\} \\ &\quad \textit{where } T = \widetilde{eval}(\mathsf{LeftJoin}(\tilde{q}_1, \tilde{q}_2, \bar{c} = \bar{c}')),\; S_{T_i} = \{\mathsf{Decode}(\beta, \widetilde{eval}(\tilde{q}_i)) \mid \beta \in \mathsf{BVDFS}(\tilde{q}_i)\}\end{aligned}$$

$$\mathsf{BVDFS}(\mathsf{Rename}(\tilde{q}, name, \bar{c})) = \mathsf{BVDFS}(\tilde{q})$$

$$\begin{aligned}&\mathsf{FiltersToBV}(\tilde{q}) = \{\widetilde{[b_1 ... b_r]_r} \mid f \in F \wedge b_i = \mathsf{Eval}(f, T[i])\} \\ &\quad \textit{where } T \leftarrow \widetilde{eval}(\tilde{q}),\; r \leftarrow \mathsf{rowNum}(T), F \leftarrow \mathsf{EnumAndGroupPred}(\tilde{q}, I, \mathit{Const})\end{aligned}$$

Figure 3.12: The optimized version of Algorithm 3.10 that searches over bit-vectors instead of tables. (All BVDFS functions are passed with the input table I and constant set C, we omitted them in the algorithm for simplicity consideration.)

3.5.3 Ranking and User Interaction

After obtaining candidate queries from the main synthesis algorithm, our system heuristically scores and ranks them before returning them to users. Queries are scored based on the following criteria:

- *Simplicity.* Queries with simpler structures and filter predicates are scored higher than more complex ones. For example, queries with predicates formed by column comparisons are scored higher than those containing compound predicates or predicates formed by Exists. Similarly, queries with fewer nested subqueries are scored higher.

- *Predicate naturalness.* Queries containing more natural predicates are scored higher, e.g., the predicate `T1.id = T2.id` has a higher score than `T1.id > T2.value` since equi-join predicates are more commonly seen in practice.

- *Constant coverage.* Queries with a better coverage of constants are scored higher than those that lacks such coverage.

After ranking, our system returns the top candidates to the user. If none of top queries is correct, the user can return our system to refine the result by providing new I/O examples or aggregation functions that the query could use.

In general, crafting a new I/O example from scratch can be costly and ineffective at refining synthesis results. However, in the case of SQL, the new I/O example can be easily created by incrementally modifying the old one. As often observed from Stack Overflow, users typically create new examples by appending new rows or modifying contents of a few table cells in previously provided tables.

While our current ranking model is simple, it can already effectively rank the correct solution among top 5 of the candidates for a relatively large number of real-world benchmarks (see Section 3.6). In addition, our synthesis algorithm is orthogonal to ranking algorithms and interaction models. Hence, it can be integrated with other interaction models [119, 163, 165] to further increase usability. For example, our system can be integrated with Cosette [34, 35], a solver for SQL queries, to further reduce the need to provide further I/O examples: given the top k synthesized queries, Cosette can be used to compute a distinguishing set of input tables S such that applying each of the top k queries on S yields different query output. Using distinguishing tables, the user only need to choose the correct result that matches the input example to obtain the correct candidate query.

3.6 Evaluation

We implemented our algorithm in Java as a system called Scythe. In this section, we report the evaluation of Scythe on a set of 193 real-world benchmarks. The evaluation was performed on a quad-core Intel Core i7 2.67GHz CPU with 4GB memory for the Java VM.

3.6.1 Implementation Optimization

We adopted the following two additional optimizations in our implementation. First, our implementation avoided enumerating semantically equivalent queries, i.e., queries that are equivalent on all possible inputs, as much as possible to increase search efficiency. For example, we avoided enumerating both "`Select T1.a, T2.b From T1 Join T2`" and "`Select T1.a, T2.b From T2 Join T1`" by restricting the order of tables in Join. Second, we performed the grouping of predicates for abstract queries during the first phase (directly when they are enumerated) and made the grouping result sharable among different abstract queries, since different abstract queries may contain same abstract subqueries.

3.6.2 Benchmarks

We collected 193 benchmarks for evaluation, including 165 benchmarks from Stack Overflow and 28 benchmarks from prior work [218]. Each benchmark includes an input-output

example pair and a reference solution (accepted answers on Stack Overflow or solutions provided in [218]). The statistics of benchmarks are shown in Figure 3.13 (for example size) and Figure 3.15 (for feature statistics). Besides, among the 193 benchmarks, there are 61 benchmarks contain constants provided by the user: 44 benchmarks contain exactly 1 constant, 15 benchmarks contain 2 constants, and 2 benchmarks contain 4 constants.

Stack Overflow. Our main evaluation benchmarks are the 165 Stack Overflow posts, divided into three groups.

- *Development set* (so-dev): These 57 benchmarks are collected from posts under tags "sql", "moving-average", "great-n-per-group" in our development time.
- *Top-voted posts* (so-top): These 57 benchmarks are *all* top-voted (i.e., vote greater than 30) Stack Overflow posts containing input-output examples and are not marked as "duplicate posts,"[5] featuring most common tasks that end-users have trouble with. In our collection process, we *exhaustively* went through the search result and picked all posts about SQL programming that contained input-output examples. We excluded posts that were not related to SQL programming (e.g., how to avoid SQL inject attack) or posts about how to write queries to update database (e.g., queries that start with `Set` or `Update`).
- *Recent posts* (so-recent): These 51 benchmarks are *all* the posts containing input-output examples posted during the 14 day period between 2016-10-09 and 2016-10-23, with additional constraints that the posts should contain an accept answer and they are not marked as duplicate posts.[6] These tasks are more specialized and typically involve more complex features compared against top-voted questions, featuring long-tail end-user SQL problems. The collection process are the same as that for top voted posts.

ASE'13 Benchmarks. We obtained an additional 28 benchmarks from SQLSynthesizer [218], containing 5 forum posts and 23 textbook questions.

3.6.3 Evaluation Process

We compared Scythe to the implementation of the equivalence-class based enumerative search algorithm described in Section 3.2 (Enum). The same algorithm was used to rank the output in both cases.

---

[5] With the search term "*[sql] is:question score:30.. lastactive:5y.. hasaccepted:yes duplicate:no*".

[6] With search term "*table result [sql] score:0.. is:question created:2016-10-09..2016-10-23 duplicate:no hasaccepted:yes*"

	No.	Size	Scythe	Enum	Zhang.
so-dev	57	31.6	55 (96%)	33 (58%)	-
so-top	57	24.7	41 (72%)	33 (58%)	-
so-recent	51	34.6	29 (57%)	18 (35%)	-
ase13	28	56.8	18 (64%)	8 (29%)	15(53%)
total	193	34	143 (74%)	92 (48%)	-

Figure 3.13: Benchmark statistics and the the number of benchmarks that can be solved by different algorithms given 600 seconds limit. The "Size" column shows the average size of the benchmark examples (the size for a benchmark refers to the the number of cells in the input output tables plus the number of provided constants).

For the ASE'13 benchmark, we also compared our algorithm to SQLSynthesizer [218] based on their paper report.[7] SQLSyntheiszer is a PBE system that synthesizes SQL queries using decision trees. Queries in its grammar has a fixed template "`Select` $\bar{c}$ `From` $\bar{T}$ `Where` p `Group By` $\bar{c}$ `Having` p"; SQLSynthesizer heuristically enumerates tables ($\bar{T}$) to be joined and then adopts a decision tree designed for the template to learn column names and predicates to fill the the template.

We ran each benchmark using the different algorithms by feeding the provided input-output example provided by the user, subject to a 600 seconds time limit. If the algorithm terminated within the time limit, we checked the returned candidate queries against the reference solution: we marked the problem "solved" if the reference solution (or a semantically equivalent one determined manually) was among the top 5 returned result. If the algorithm failed to terminate or the correct query was not among top 5, we either 1) manually modified the original example based on the text description shown in the posts and re-evaluated the algorithm on the new example or 2) extracted the aggregation functions from the post and supplied it to the algorithm (if exists) and reran it with only provided aggregation functions. If the algorithm continued to fail after this interaction, we marked the problem as "failed". The statistics we collected during the evaluation process are reported below.

3.6.4 Number of Solved Benchmarks

Figure 3.13 shows the number of benchmarks solved by different algorithms, Figure 3.14 shows the performance comparison between Scythe and Enum, and Figure 3.15 shows the statistics of the queries synthesized by Scythe.

Scythe solved 143 cases within the 600 seconds time limit (114 within 10 seconds). Enum solved 92 cases within the 600 seconds time limit (18 within 10 seconds). Cases that Scythe

[7] We did not compare our algorithm against SQLSynthesizer on the Stack Overflow benchmarks as the tool is not publicly available.

solved but Enum did not are typically those containing higher subquery nesting levels or large example sizes. A comparison between Scythe and Enum on benchmarks that both algorithms solved shows that Scythe is on average 57× faster (ranging from 7-200× faster).

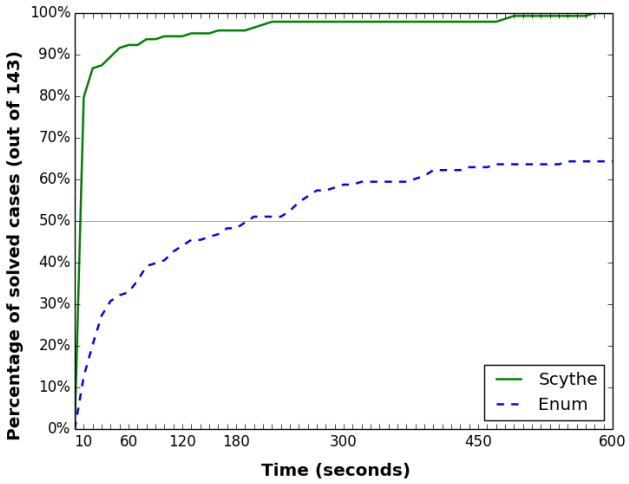


Figure 3.14: The percentage of benchmarks solvable with the increasing time limit specified by the x-axis (out of the 143 cases that Scythe successfully solved).

For the ASE'13 benchmark, Scythe solved 3 more cases than SQLSynthesizer since our algorithm supports more operators (`Exists`, `Left Outer Join`, and `Union`). Furthermore, although Scythe searched over a significantly larger space of queries (as it supports more types of subquery nesting, more operators, and have no template restriction in comparison to SQLSynthesizer), it showed no compromise in performance: for the 18 benchmarks that Scythe solved, 12 were solved in 10 seconds, and 6 of them were solved with over 10 seconds but within 120 seconds. In comparison, SQLSynthesizer solved 14 cases within 10 seconds and 1 with 120 seconds.

3.6.5 Algorithm Statistics

We present several statistics of our algorithm to demonstrate how different parts of our algorithm contributes to the overall efficiency improvements.

Abstract Grammar. We measured the effectiveness of the abstract query synthesis algorithm in Scythe (Section 3.5) by measuring (1) the number of intermediate results generated by Scythe compared to Enum, and (2) the search space reduction rate resulted from pruning away abstract queries that were inconsistent with the output.

	so-dev	so-top	so-recent	ase13	total
Join	40	31	18	10	90
Aggr	43	39	19	13	113
Left-Join	2	1	3	1	7
Union	1	0	9	1	11
Feature(>1)	35	30	18	9	92

Figure 3.15: Statistics for the occurrences of advanced SQL in the solutions synthesized by Scythe. Row "Feature(>1)" refer to queries containing more than one of the advanced operators shown above.

For the first question, on the benchmarks that both algorithms can solve, the average number of tables generated by Scythe is 996, while that generated by Enum is on average 7× more (ranging from 1.5-36×). Furthermore, for the 50 cases that only Scythe successfully solved, the average number of intermediate tables is 27,832 (max 471,316), thus the reduction is essential to allow Scythe to find answers to these more complex cases.

For the second question, pruning with abstract grammar reduces the size of search space by a factor of 2145× (ranging from 1.1-169,028×). The reduction rate is typically higher for cases whose output table size is larger or those whose solution contains aggregation subquery.

Predicate Synthesis. We measured the effectiveness of the predicate synthesis algorithm (Section 3.5.2) in Scythe by measuring the reduction rate due to enumerating predicates that were in the same equivalence class rather than all syntactically equivalent ones. For each predicate hole in an abstract query, the average reduction rate from candidate predicates to their equivalence classes is 45,179× (ranging from 51-2,170,150×).

3.6.6 Effectiveness of Ranking

Next, among the 143 cases that Scythe solved, we need to provide additional input-output examples for 22 cases, specify aggregation functions for 9 cases, and provide both extra information (a second example or aggregation functions) for 3 cases to help Scythe disambiguate consistent queries. For every one of the cases that requires additional input-output examples, the average number of cells added to example (considering cells added to both input and output example) is 7.8 (maximum 17).

Besides, we also found 1 out of the 50 cases that Scythe failed to solve was failed due to our interaction model design limitation, as the query can only be constraint with at least two I/O example pairs at the same time, while our system allows only one I/O pair at a time.

Thus, though imperfect, our ranking algorithm remains effective in finding correct solutions from the large number of candidates.

3.6.7 Failed Cases

We classified the 50 cases that Scythe failed to solve into five categories to summarize the limitations of our algorithm: (1) cases requiring adding other standard SQL features to our grammar [f-exp1], (2) cases requiring additional non-standard SQL features (e.g., arithmetic expressions, date/string transformations, pivoting, window functions or dense ranking) [f-exp2], (3) cases that our language is able to express but failed due to scalability issues (e.g., increasing the level of nested queries) [f-scale], and (4) cases expressible but the corrected answer is unable to be disambiguated even after providing new examples [f-rank].

	f-exp1	f-exp2	f-scale	f-rank	total
so-dev	1	0	1	0	2
so-top	0	16	0	0	16
so-recent	0	17	5	0	22
ase13	0	0	9	1	10
total	1	33	15	1	50

Figure 3.16: Benchmarks that Scythe fails to solve.

As shown in Figure 3.16, a major fraction of the unsolvable cases (33 cases) are due to non-standard SQL features. Adding support for these specialized features requires that our algorithm work cooperatively with synthesizers from other domains, e.g., arithmetic expression synthesizer [164] or string solvers [95, 71], which we consider as future work.

There are also 15 cases Scythe failed to solve due to scalability. Those cases either requires a solution with highly nested subqueries (> 5) or contains large I/O example table size (> 60 cells). They meet the bottleneck of our synthesis algorithm, making the pruning costly and relatively ineffective. We noticed that a majority of them (9 cases) are the textbook questions from the ASE'13 benchmarks, which are designed for teaching purposes and appear rarely in online forums.

We also found 1 failure case caused by the unsupported SQL feature (keyword `Limit`) and 1 caused by interaction model limitation. Note that although other operators like `Except`, `All` do not directly appear in our grammar, they can be reformulated into queries written using the keyword `Exists` such that they can be expressed using our language.

3.6.8 Threat of Validity

Our main study and evaluation were based on the benchmarks from Stack Overflow, but it is possible that these benchmarks do not representative all practical end-user tasks. In analyzing the ranking effectiveness of Scythe, we measured only the number of cells needs to be added to the original example, not how hard it might be for end-users to provide them. Studying end-user behavior could help us better address this problem.

3.7 Related Work

Programming by Examples. Scythe is inspired by Programming by Example (PBE) applications: users of such systems specify a program using I/O examples, and the system subsequently synthesizes a program consistent with the examples. This approach has been used to synthesize data transformation programs [71, 163, 166, 77, 32], data extraction scripts [102], map-reduce programs [1, 167], data structure transformations [211] and also SQL queries [218, 182, 30].

SQLSynthesizer [218] and Query By Output [182] are two PBE systems for SQL queries. Both of them synthesize queries using the decision tree algorithm: the systems come with a set of templates and they complete holes in the template using the decision tree algorithm. Since the decision tree algorithm is limited by the facts that (1) the algorithm needs to build a different decision tree for every query template and (2) the decision tree size for a template is exponential to the number of rows and columns of the I/O example. Thus, neither systems support advanced SQL features like `Left Join`, `Union`, `Exists` and free-form subquery nesting, which is essential in solving real-world problems. In contrast, due to the scalability improvement of our two-phase synthesis algorithm, Scythe supports a wider range of SQL features used in practical settings.

Program Synthesis Algorithms. Inductive program synthesis algorithms [16, 3, 70] can be roughly classified into the following five categories: (1) enumerative search algorithms [138, 136, 63, 186, 103], (2) constraint-solver aided synthesis [181, 169], (3) type-directed synthesis [132, 62], (4) version space algebra algorithm [100, 140, 71], and (5) stochastic search [157].

Our synthesis algorithm is most closely related to enumerative search algorithms, which were adopted in Transit [186], Lenses [138], CodeHint [63] and AlphaRegex [103]. The first three systems are optimized using the concept of equivalence-class reduction where intermediate results are grouped together based on their evaluation results so that behaviorally equivalent programs are visited only once. Our algorithm differs from them in our development of abstract queries to decompose the search process, which is essential to solve the challenge of memoizing tables to scale up the synthesis algorithm. Besides, AlphaRegex, an enumerative synthesizer for regular expressions whose pruning strategy resembles the

first phase of our algorithm, enumerates all regular expressions within the search space to find those consistent with user provided examples; it prunes intermediate regex skeletons based on their under/over-approximation to increase scalability. Our first phase algorithm differs in how over-approximations are computed: evaluating query skeletons requires the design of the abstract query language with different evaluation rules, while evaluating regular expression skeletons can be achieved purely using built-in operators.

Interactive Refinement. Statistical ranking algorithms [162, 165] and interactive disambiguation interfaces [119, 104] have been proposed to improve PBE system accuracy. Scythe can potentially integrate such techniques to enhance the quality of the synthesized programs.

Other Related Techniques. *Inductive logic programming* (ILP) [126, 48] is an approach adopted by the AI community for learning general logic representations from demonstrations: given N I/O examples together with their logical representations, ILP algorithms learn the set of programs that generalizes all examples using bottom-up searches. However, ILP cannot be directly applied to SQL query synthesis. First, it requires multiple examples to learn a general form. In our case, each task is specified using only one I/O pair, and tables cannot be trivially decomposed into smaller I/O examples for query synthesis (since doing so will likely change the user's intention). Second, using ILP requires logical representations of each I/O example, and constructing them from I/O examples for SQL is highly non-trivial, as shown by [140].

3.8 Summary

In this chapter, we presented a system called Scythe, which efficiently synthesizes SQL queries from I/O examples. The key idea of our approach is the design an abstract language of queries to decompose the original complex synthesis problem into easier-to-solve subproblems. The evaluation of Scythe on a set of 193 real-world benchmarks shows that it can effectively and efficiently solve real world SQL query synthesis tasks.

Chapter 4

Speeding up Symbolic Reasoning for Relational Queries

The ability to reason about relational queries plays an important role across many types of database applications, such as test data generation, query equivalence checking, and computer-assisted query authoring. Unfortunately, symbolic reasoning about relational queries can be challenging because relational tables are multisets (bags) of tuples, and the underlying languages, such as SQL, can introduce complex computation among tuples. We propose a space refinement algorithm that soundly reduces the space of tables such applications need to consider. The refinement procedure, independent of the specific dataset application, uses the abstract semantics of the query language to exploit the provenance of tuples in the query output to prune the search space. We implemented the refinement algorithm and evaluated it on SQL using three reasoning tasks: bounded query equivalence checking, test generation for applications that manipulate relational data, and concolic testing of database applications. Using real world benchmarks, we show that our refinement algorithm significantly speeds up (up to 100×) the SQL solver when reasoning about a large class of challenging SQL queries, such as those with aggregations.

4.1 Introduction

The relational model [37] is one of the most popular ways to represent data. Under the relational model, data is organized into tables. Each table consists of a bag of tuples that contains multiple attributes and their corresponding values, with all tuples in the same table sharing the same number of attributes. The simplicity of the relational model has led to its widespread adoption among database systems, with numerous commercial and open-source implementations available.

The popularity of the relational model has also led to the development of various development tools and applications that utilize relational databases. Many such applications require reasoning about tables. For instance, one way to determine whether two relational queries, q_1 and q_2, are semantically equivalent is to check if there exists a table T such that the two queries return different results when evaluated on T. Furthermore, database testing tools require the generation of test inputs from the provided query and test conditions

to check whether the query can produce an ill-formed output on some input tables: for example, an application might return an error if a query in the application can return empty results or results containing NULL values when evaluated on a non-empty input employee relation, and these errors can be caught if we have corresponding test inputs.

Obviously, given the large number of possible tables for a database schema, exhaustively enumerating and explicitly storing them in program memory for query reasoning is infeasible. Hence, prior work has focused on reasoning about tables *symbolically*. For instance, Cosette [35], a query equivalence checker, leverages Satisfiability Modulo Theories (SMT) solvers for bounded equivalence checking. Given two queries q_1 and q_2, Cosette encodes the outputs of both queries symbolically as an SMT formula and sends the formula to an SMT solver to either: (1) prove that the two formulas are semantically equivalent (and hence the input queries are equivalent), or (2) show that the two queries are inequivalent by finding an input table as the counterexample (i.e., the two queries will return different results when evaluated on it). Similar techniques have been employed in testing frameworks as well [179, 31, 189]: these frameworks aim to generate test inputs from the given query and test conditions. These test conditions can be path conditions from a database application or unit test assertions for output properties that we are interested in.

While representing tables symbolically indeed allows such tools to reason about different database applications that arise in practice, we believe a substantial opportunity remains for further improvement. To our knowledge, none of such relational query reasoning tools leverage the properties of tables or the domain-specific aspects of relational query languages. An example property is the independence between groups in a query with Group By: in these queries, tuples are partitioned into different groups according to the value of the grouping keys, and different groups are reduced into single tuples independent of each other. As a result, without exploiting such properties, many existing tools explore an unnecessarily large number of tables during execution. As we will show, this significantly hampers such tools' ability to analyze more complex real-world database applications.

In this chapter, we describe a way to *systematically identify and exploit* properties of tables and the SQL query language for relational query reasoning.[1] Specifically, we focus on scaling up SQL reasoning tasks aimed at finding input tables T_I for a given query q (or multiple queries) such that the output table T_O satisfies a property expressible using a subset of first-order logic with a single existential quantifier: $\exists t_O \in T.\psi(t_O, T_O)$ (i.e., there exists a tuple t_O in the output table T_O satisfying the property ψ). As we will discuss in Section 4.2, while this subset does not include properties such as "the output table has exact size 5," it nonetheless allows us to check for properties such as the following: (1) "the output table

[1]While we focus on SQL in this thesis given its popularity, we believe the techniques can be applicable to other relational query languages as well.

contains a tuple with multiplicity greater than zero" (under bag semantics, multiplicity of a tuple if the number of times it appears in the table), which arise in many situations in the unit test generation task for database applications [189]; (2) "the output table does not contain any attribute with value equals to NULL," which is useful for query optimization; and (3) "queries q_1 and q_2 outputs contain the same tuple t but with different multiplicities," meaning that the two queries are semantically different [24].

To determine the validity of such properties, our key idea is to *backwardly* compute the *provenance* of tuples, i.e., the lineage of how a given tuple is derived from the input tables, using *the abstract semantics of SQL* to refine the space of input tables that symbolic reasoning tools need to consider. We next present two motivating examples to demonstrate our insights.

Motivating Example 1. We first show a simple example in the context of unit test generation [189]. In this task, given the following query q parameterized with `@x`, we aim to either find an input table `Bonus` such that the output of q is non-empty for a given concrete parameter, or prove that no such input table exists. If such input exists for the given parameter, the input table, the parameter, and the query can be combined as a unit test for the database.

```
-- q
Select job, sal
From Bonus
Where sal <= @x And sal > 2;
```

Bonus_1

job	sal
2	11

Bonus_2

job	sal
3	5
3	5

Bonus_3

job	sal
2	11
3	5
3	5

Figure 4.1: Given the query q, whether q would produce a non-empty output when applied to $\texttt{Bonus}_3$ is subsumed by whether q produces non-empty outputs when evaluated on $\texttt{Bonus}_1$ and $\texttt{Bonus}_2$.

Figure 4.1 shows a concrete example. The query q (parameterized with `@x`) filters tuples in table `Bonus` using the condition "`sal > 2 And sal <= @x`". Assuming that the number of tuples considered by the query solver is bounded to k, the solver would encode the search space consisting of all tables with at most k tuples as a symbolic table and search for desirable assignments for all $2k$ attributes in the table, say by invoking an SMT solver.

However, we actually do not need to consider the full space of all tables with at most k tuples to search for the desirable table. Instead, given that the query q contains only one filter operation, we only need to consider the space of all tables with exactly *one* tuple (note that the tuple may appear multiple times in the table due to bag semantics). This is true because the query q does not introduce interactions among different tuples during computation, i.e., whether a tuple t in the input table would be included in the output does not depend

on whether another tuple t' would be included or not. In other words, the *provenance* of an output tuple (a tuple in the output table) with `job`=j_1 and `sal`=s_1 is exactly those tuples in the input table with `job` and `sal` equal j_1 and s_1 respectively, but not any other tuples.

To leverage the provenance of tuples, observe that if we have already examined in the table search space that neither $\texttt{Bonus}_1$ nor $\texttt{Bonus}_2$ in Figure 4.1 is an input table that satisfies the test condition for the given `@x` (e.g., when `@x`$= 4$), we don't need to check $\texttt{Bonus}_3$. Meanwhile, if query q returns a non-empty output when applied to $\texttt{Bonus}_3$, then at least one of $\texttt{Bonus}_1$ or $\texttt{Bonus}_2$ would be a desirable input when searching in the space of all tables containing only one unique tuple. For example, if `@x`$= 10$, $\texttt{Bonus}_1$ is a valid unit test input, then smaller input tables such as $\texttt{Bonus}_2$ and $\texttt{Bonus}_3$ are valid test inputs as well.

Given this insight, we need to consider only tables containing k tuples where all tuples share the same `job` and `sal` values. The SQL solver only needs to find one concrete value for each field, rather than $2k$ different values for both fields in the table. As we will show in Section 4.6, this reduction dramatically accelerates the test generation process, especially when the bound k is large.

Motivating Example 2. For bounded verification [34] or query disambiguation [24], the goal is either to prove the equivalence of two queries, q_1 and q_2, within a bounded space (i.e., q_1 and q_2 always return the same output when applied to the same set of input tables in the space), or to construct input tables that distinguish q_1 from q_2 (i.e., q_1 and q_2 return different results when evaluated on the same constructed distinguishing input tables).

Figure 4.2 shows two semantically equivalent queries, q_1 and q_3, and query q_2, which is inequivalent to them. Query q_1 first filters the input table `Bonus` by `sal > 5`, then groups the result by `job` and `dept` to calculate the maximum `sal` for each group, and finally keeps only the groups with `job` values less than 10. Query q_3 differs from q_1 only in its order of evaluating the filter predicate `job < 10` and grouping. Since the predicate refers to only columns appeared in the Group-By clause (i.e., `job`), q_1 and q_3 are equivalent. Query q_2 is semantically different from them since it groups tuples only by `job`, not by both `job` and `dept`.

To check the equivalence between queries q_1, q_3 within the bounded space of all tables with at most k tuples, the solver first encodes the search space as a symbolic table and queries the underlying SMT solver to check whether q_1 and q_3 always produce the same output when applied to the symbolic input. In this example, the SQL solver faces the challenge to reason about grouping and aggregation: it needs to consider all 2^k possibilities of groups in both queries (or 2^k number of possibilities to partition the input table according to the grouping keys). Verification time increases exponentially as the bound increases.

```
-- q1
Select job, dept,
       Sum(sal)
From Bonus
Where sal > 5
Group By job, dept
Having job < 10;
```

```
-- q2
Select job, Max(dept),
       Sum(sal)
From Bonus
Where sal > 5
Group By job
Having job < 10;
```

```
-- q3
Select job, dept,
       Sum(sal)
From Bonus
Where sal > 5
  And job < 10
Group By job, dept;
```

Figure 4.2: Three queries with the relation $q_1 \equiv q_3 \not\equiv q_2$. Query q_3 differs from q_1 only by evaluating the filter `job < 10` before grouping. Since the predicate `job < 10` only refers to columns appeared in the Group By clause, this transformation is semantics preserving. q_2 is semantically different from q_1 since it groups tuples only by `job` but not both `job` and `dept`.

Fortunately, we can also refine the search space in a way similar to that in the first case, after realizing that both queries do not introduce interplay among different $(\texttt{job}, \texttt{dept})$ groups during evaluation: different groups are aggregated independent of each other. In other words, each output tuple depends only on tuples in the input table belonging to the same $(\texttt{job}, \texttt{dept})$ group: the provenance of tuple $(job_1, dept_1, sumSal_1)$ are tuples in the input table satisfying $\texttt{job} = job_1$ and $\texttt{dept} = dept_1$).

Thus, we can restrict the search space to just tables with one $(\texttt{job}, \texttt{dept})$ group and restrict all tuples to satisfy `job < 10` and `sal > 5` (since no other tuples would pass through the filter to the output). The key insight behind this refinement process is this: assuming there exists an input table $\texttt{Bonus}_1$ such that q_1 and q_3 return different outputs T_{out1} and T_{out2} when applied to it, then there must be a tuple t whose multiplicity in T_{out1} differs from its multiplicity in T_{out2}. Then, we can also construct another input table $\texttt{Bonus}_2$, one that contains only tuples in $\texttt{Bonus}_1$ whose `job` and `dept` are the same as those in the tuple t, to reproduce the difference between q_1 and q_3. Apparently, since $\texttt{Bonus}_2$ contains just one $(\texttt{job}, \texttt{dept})$ group, it is included in the refined search space. On the other hand, if we prove that the two queries are equivalent in the refined search space, we also prove their equivalence in the original search space.

Similarly, to find a distinguishing input that distinguishes q_2 from q_1 in Figure 4.3, we can refine the search space in a similar fashion. Note that while q_2 could introduce interplay among tuples belonging to different $(\texttt{job}, \texttt{dept})$ groups since it groups only by `job` key (unlike q_1 and q_3), q_2 will not introduce interplay among tuples belonging to different `job` groups during evaluation (i.e., tuples in the input table belonging to different `job` groups won't be aggregated to the same tuple in the output). Thus, although we cannot restrict the new search space to tables with only one $(\texttt{job}, \texttt{dept})$ group, we can restrict it to tables consisting of tuples within one `job` group that satisfy `job < 10`, `sal > 5`. For example, $\texttt{Bonus}_1$

is a distinguishing input that distinguishes q_1 from q_2, as they both produce different results for tuples whose $\texttt{job} = 2$. Meanwhile, the other table $\texttt{Bonus}_2$ from the refined search space can also reproduce the difference between q_1 and q_2 based on their difference in multiplicities of tuples in the group with job equals 2. In fact, tuples in $\texttt{Bonus}_2$ are provenance tuples collected from $\texttt{Bonus}_1$ for the distinguishing output tuple $(2, 2, 8)$, which explains why $\texttt{Bonus}_2$ can reproduce the difference in multiplicities of $(2, 2, 8)$ in both query outputs, as does $\texttt{Bonus}_1$.

Since the refined search space no longer contains a table with multiple job groups for the solver to consider, the solver avoids the exponential encoding of the search space with respect to the job column, which provides a speedup in solving the problem.

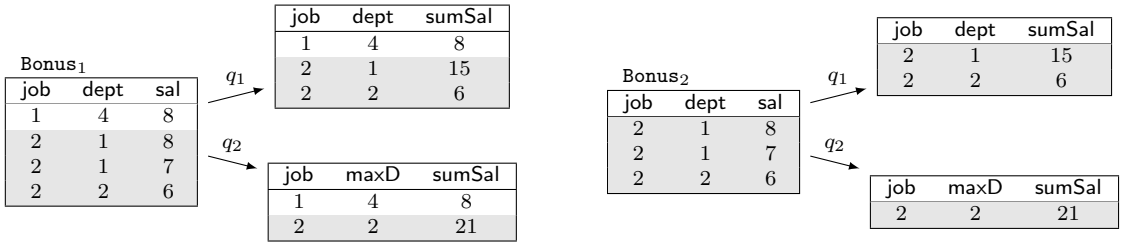


Figure 4.3: The two concrete tables $\texttt{Bonus}_1, \texttt{Bonus}_2$ are both distinguishing inputs that show the difference between q_1 and q_2 in Figure 4.2. Different colors in the table indicates different provenance groups in the input table.

Our approach. As mentioned above, in this paper we introduce a systematic approach for identifying and utilizing provenance information to refine the search space to scale up symbolic SQL reasoning. The key insight of our search space refinement algorithm is that we can perform a *symbolic provenance analysis* of the input queries to identify which tuples in the symbolic table alone are sufficient to prove or disprove the verification condition. Throughout this analysis, we construct a predicate ϕ that describes which tuples of an input table T are responsible for data generation or bounded verification.

We then use the provenance predicate ϕ to refine the original search space $\mathcal{S}$ into a new search space $\mathcal{S}'$ that is equivalent to $\mathcal{S}$ in terms of the verification condition: if there exists an input table $T \in \mathcal{S}$ satisfying the verification condition for the given queries, then there exists an input table $T' \in \mathcal{S}'$ that also satisfies the verification condition.

In particular, our provenance analysis uses only *the abstract semantics* of SQL to maintain efficiency, because computing the strongest provenance predicate for given queries requires full symbolic reasoning of the queries that can be as difficult as solving the reasoning task itself [21]. For example, we over-approximate all aggregation functions as uninterpreted

functions, so that the provenance tuples of tuple t_O (a tuple in the output table) are all tuples in the input table belonging to the same group as t_O (but in reality, certain aggregation functions like max depend only on the tuple in the input with the largest value). As we will show in Section 3.5, such abstraction introduces a sound over-approximation of the query semantics that can be used to efficiently and soundly prune the search space for symbolic SQL reasoning. In the future, we could potentially redesign different abstractions to discover better trade-offs between analysis overhead and pruning effectiveness.

Furthermore, our space refinement process relies only on the semantics of the input queries but not the underlying symbolic reasoning tool's design and implementation, which allows it to be applied to speed up different symbolic SQL reasoning tools [189, 34, 160].

We evaluated the space refinement algorithm for three symbolic SQL reasoning scenarios: bounded verification (as utilized in verifying query rewriting rules), distinguishing input generation (for query disambiguation), and unit test generation (for testing frameworks). Using 61 real-world benchmarks, we compare the performance of SQL solvers using the search space with and without refinement. Results show that our refinement algorithm effectively speeds up the reasoning of a large class of queries used in real-world applications.

In sum, this paper makes the following contributions:

- We devised a new way to utilize tuple provenance to identify tables that are equivalent to each other with respect to the reasoning process of the given relational queries.
- We designed a space refinement algorithm that utilizes the provenance property to soundly prune the space of tables that need to be explored.
- We implemented our space refinement algorithm and evaluated it on various tools that apply symbolic reasoning to reason about relational queries. Results show that our search space refinement algorithm can effectively improve symbolic reasoning for SQL across different usage scenarios, and speeds up to 100× SQL solver reasoning for complex queries with aggregation.

We next review symbolic SQL reasoning (Section 4.2), describe our approach with a query equivalence checking example (Section 4.3), then formally introduce our space refinement algorithm (Section 4.4, Section 4.5), and finally evaluate our algorithm in the context of SQL equivalence checking and test generation (Section 4.6).

4.2 Problem Definition

We start out by briefly reviewing backgrounds in symbolic SQL reasoning and defining the space refinement problem.

Table and SQL. Table is the first class value in SQL consisting of a schema and a bag of tuples [130].[2] A table schema defines the number of columns and the type of each column. A tuple t is a list of values with the same size as the schema. In our paper, we use $[\![T]\!]t$ to denote computing the multiplicity of t in table T: if t is in T, it returns the number of times t appears; otherwise, it returns 0. SQL queries are functions over tables, and we use $[\![q(\overline{T})]\!]$ to represent evaluating q against a list of input tables $\overline{T}$ (the result $[\![q(\overline{T})]\!]$ is a table T_{out}).

Under bag semantics, two tables are equal if and only if every tuple in them has the same multiplicities. Formally, table equality can be defined as $T_1 = T_2 \iff \forall t. [\![T_1]\!]t = [\![T_2]\!]t$. Two queries q_1, q_2 are equivalent if and only if evaluating them returns the same results for *all* possible input tables that are compatible with the schema. This equivalence relation can be defined as $q_1 \equiv q_2 \iff \forall \overline{T}.\ [\![q_1(\overline{T})]\!] = [\![q_2(\overline{T})]\!]$.

Symbolic SQL Reasoning. We focus on SQL reasoning tasks in the form of finding input tables $\overline{T}_{\text{in}}$ for a given query q (or queries q_1, q_2) such that the output table $[\![q(\overline{T}_{\text{in}})]\!]$ satisfies a property in the form of $\Psi(T_{\text{out}}) = \exists t_O.\psi(t_O, T_{\text{out}})$, where the only use of T_{out} in ψ is to check multiplicity of t_O (i.e., used as $[\![T_{\text{out}}]\!]t_O$). This problem is solved by finding the satisfiability problem of $\exists \overline{T}_{\text{in}}.\Psi([\![q(\overline{T}_{\text{in}})]\!])$ using the formula Ψ above.

- This formula restriction prevents us from reasoning about properties like "the output table T_{out} has exactly 5 tuples" where $\Psi(T_{\text{out}}) = (|T_{\text{out}}| = 5)$, or "every tuple in T_{out} has the same multiplicity" where $\Psi(T_{\text{out}}) = \exists m \forall t_O \in T_{\text{out}}.([\![T_{\text{out}}]\!]t_O = m)$.
- On the other hand, we can still use the formula Ψ to express many practical reasoning tasks. For unit test generation, the property $\Psi(T_{\text{out}}) = \exists t_O.[\![T_{\text{out}}]\!]t > 0$ (there exists a tuple in the output table with multiplicity > 0). For equivalence checking, the property $\Psi(T_{\text{out1}}, T_{\text{out2}}) = \exists t_O.([\![T_{\text{out1}}]\!]t_O \neq [\![T_{\text{out2}}]\!]t_O)$, i.e., exists a tuple t_O has different multiplicities in two query outputs.

As we will show later, our algorithm exploits the fact the we only need to find one t_O that satisfies the property ψ to witness the satisfaction of the property Ψ to conduct search space refinement.

Search Space Refinement. We define the search space refinement problem as follows. Given a query q (or queries q_1, q_2), a property $\Psi(T_{\text{out}})$ and a search space $\mathcal{S}$ of input tables, we want to find a new search space $\mathcal{S}'$ such that $\mathcal{S}'$ is equivalent to $\mathcal{S}$: i.e., if there exists $\overline{T}_{\text{in}} \in \mathcal{S}$ s.t. $\Psi([\![q(\overline{T}_{\text{in}})]\!])$ holds, then there exists $\overline{T}'_{\text{in}} \in \mathcal{S}'$ that also satisfies the property Ψ. Our goal is to construct $\mathcal{S}'$ such that $\mathcal{S}'$ is smaller than $\mathcal{S}$ and can be explored faster by the SQL solver.

[2]There are other semantics of SQL, e.g., set semantics, but bag semantics is the most commonly implemented in modern commercial database systems.

In the rest of this chapter, we explain our approach using the query equivalence checking problem (i.e., checking whether two queries q_1 and q_2 are semantically equivalent within some given space $\mathcal{S}$) as an illustrative example of reasoning tasks of the form $\Psi(T_{\text{out}}) = \exists t_O.\psi(t_O, T_{\text{out}})$. Given two queries q_1 and q_2, we call $\overline{T}_{\text{in}}$ a counterexample if $[\![q_1(\overline{T}_{\text{in}})]\!] \neq [\![q_1(\overline{T}_{\text{in}})]\!]$. If q_1 and q_2 are semantically inequivalent, then there must exist at least one tuple t such that $[\![q_1(\overline{T}_{\text{in}})]\!]t \neq [\![q_2(\overline{T}_{\text{in}})]\!]$ (i.e., its multiplicity differs in the outputs of q_1 and q_2). We call such t a *distinguishing output tuple* that demonstrates the semantic difference between the two queries.

4.3 Overview

We now use a concrete example for query equivalence checking to walk through our space refinement algorithm. As shown in Figure 4.4, our space refinement algorithm takes as input two queries q_1, q_2 and a search space $\mathcal{S}$, and it outputs the search space after refinement $\mathcal{S}'$ that is equivalent to $\mathcal{S}$ for the purpose of checking q_1 and q_2.

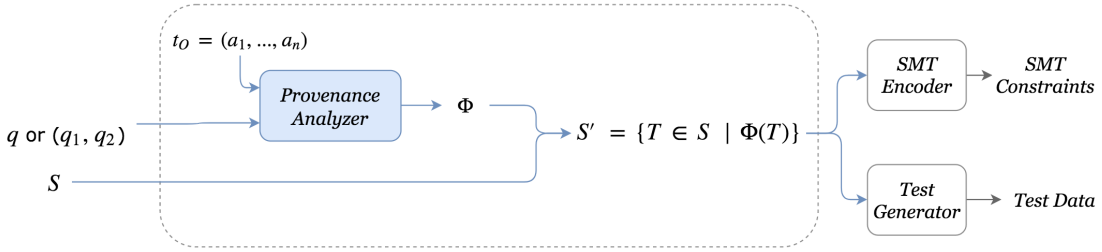


Figure 4.4: The search space refinement algorithm (dotted box). The algorithm takes as input a query q (or two queries q_1, q_2) and a space of tables $\mathcal{S}$ for space refinement. It outputs a refined search space $\mathcal{S}'$ that is equivalent to $\mathcal{S}$. $\mathcal{S}'$ can be used in bounded verification or test data generation. Internally, the predicate Φ is computed using backward provenance analysis of the given query using a symbolic output tuple $t_O = (a_1, ..., a_n)$. The predicate is then used by the space refinement module to refine $\mathcal{S}$ into $\mathcal{S}'$.

To compute $\mathcal{S}'$, our algorithm contains the following two main modules:

- *(Provenance Analysis)* The provenance analysis process derives a predicate ϕ from the queries q_1, q_2 that describes a condition such that if two tables in the search space $\mathcal{S}$ satisfy it, they would be equivalent with respect to finding an distinguishing output tuple t_O. The predicate ϕ is computed by combining the provenance predicate ϕ_1 for t_O in q_1 (ϕ_1 determines which tuples in the input table T contribute to the multiplicity of the output tuple t_O) and the provenance predicate ϕ_2 for t_O in q_2.

- *(Space Refinement)* Using the provenance predicate ϕ, we construct a refinement predicate Φ describing what properties a table T need to satisfy to refine the search space and then use it to construct the refined search space $\mathcal{S}'$ from $\mathcal{S}$.

Example. We reuse the equivalence checking example between q_1 and q_2 shown in Figure 4.2 to demonstrate the space refinement algorithm. Given the queries q_1 and q_2 in Figure 4.2 and a search space $\mathcal{S}$, our goal is to determine whether q_1 and q_2 are equivalent within $\mathcal{S}$. As mentioned in Section 4.1, since the two queries use different grouping keys, they are inequivalent, and our goal is to find a counterexample for them. An counterexample is shown in Figure 4.3. Note that both queries operate on input tables with schema Bonus(job, dept, sal).

4.3.1 Provenance Analysis

As shown in Figure 4.4, the first step is to perform a provenance analysis to determine which tuples in T contribute to the multiplicity of t_O in the outputs of q_1 and q_2 ($[\![q_1(T)]\!]$ and $[\![q_2(T)]\!]$). We start out by analyzing the provenance of t_O with regard to q_1.

We first convert the query into the sequential expressions shown in Figure 4.5 (left), where q_1 is computed from its subexpression q_{11}, $q_1 1$ is computed from q_{12}, and q_{12} is directly computed from input table T. The operator $\mathsf{Select}(q, f)$ represents filtering the result of q using the predicate f (for Having and Where) and Aggr represents the SQL operator Group By. The key idea behind provenance analysis process is as follows: given an expression $q = \mathsf{op}(q')$ where q' is a subexpression of q, the key is to determine which tuples in $[\![q'(T)]\!]$ are sufficient to determine the multiplicities of (or, "contribute to") the tuples in $[\![q(T)]\!]$ that we are interested in. The goal is to "propagate" the deduced provenance relation from the outermost query (e.g., q_1) to the input table T, and subsequently to represent the provenance information as a predicate over tuples. The concrete analysis process is as follows:

- *(Step 1).* We start the analysis process by analyzing which tuples in $[\![q_1(T)]\!]$ contribute to the multiplicity of the output tuple t_O. Obviously, such tuples should have the same values for each attribute as t_O, and they can be represented as all tuples in $[\![q(T)]\!]$ satisfying the predicate $\phi_{10}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{sum} = t_O.\mathsf{sum})$ ($t.c$ stands for referencing column c in t). We denote these tuples as $[\![q(T)]\!]^{\phi_{10}}$.

- *(Step 2).* For the expression $q_1 = \mathsf{Select}(q_{11}, \mathsf{job} < 10)$, we analyze which tuples in the subexpression result $[\![q_{11}(T)]\!]$ determine multiplicities of tuples in $[\![q(T)]\!]^{\phi_{10}}$. Since the latter determines the multiplicity of t_O in $[\![q_1(T)]\!]$, we can use this analysis result to propagate the provenance information one level back. Since q_1 contains the filter predicate $\mathsf{job} < 10$, only tuples in $[\![q_{11}(T)]\!]$ and satisfying both $\phi_{10}(t)$ and $\mathsf{job} < 10$ would contribute

The decomposition of q_1	Analysis steps	Provenance predicates
	(1) initialize	$\phi_{10}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{sum} = t_O.\mathsf{sum})$
$q_1 = \mathsf{Select}(q_{11}, \mathsf{job} < 10)$	(2) $q_1 \rightarrow q_{11}$	$\phi_{11}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{sum} = t_O.\mathsf{sum} \wedge t.\mathsf{job} < 10)$
$q_{11} = \mathsf{Aggr}(q_{12}, [\mathsf{job}, \mathsf{dept}], \mathsf{Sum}(\mathsf{sal}))$	(3) $q_{11} \rightarrow q_{12}$	$\phi_{12}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10)$
$q_{12} = \mathsf{Select}(T, \mathsf{sal} > 5)$	(4) $q_{12} \rightarrow T$	$\phi_1(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)$

$$t_O \longleftarrow [\![q_1(T)]\!]^{\phi_{10}} \longleftarrow [\![q_{11}(T)]\!]^{\phi_{11}} \longleftarrow [\![q_{12}(T)]\!]^{\phi_{12}} \longleftarrow T^{\phi_1}$$

Figure 4.5: The analysis process of which tuples in input T contribute to the multiplicity of t_O in the output $[\![q(T)]\!]$. The analysis result is shown as a chain: for each arrow, the right hand side tuples evaluated from each subexpression determine the multiplicities of tuples on the left hand side. We use $[\![q(T)]\!]^{\phi}$ to denote tuples in $[\![q(T)]\!]$ satisfying the predicate ϕ. For example, $[\![q_1(T)]\!]^{\phi_{10}} \longleftarrow [\![q_{11}(T)]\!]^{\phi_{11}}$ indicates that tuples in $[\![q_{11}(T)]\!]$ satisfying ϕ_{11} determines the multiplicities of tuples in $[\![q_1(T)]\!]$ satisfying ϕ_{10}.

to $[\![q(T)]\!]^{\phi_{10}}$. Thus, we derive from ϕ_{10} and q_1 the predicate $\phi_{11}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{sum} = t_O.\mathsf{sum} \wedge t.\mathsf{job} < 10)$ to describe these tuples in $[\![q_{11}(T)]\!]$; we denote them as $[\![q_{11}(T)]\!]^{\phi_{11}}$.

- *(Step 3).* Similarly, we analyze which tuples in $[\![q_{12}(T)]\!]$ contribute to $[\![q_{11}(T)]\!]^{\phi_{11}}$. Since q_{11} is an aggregation query that groups by the job and dept columns, each tuple t in $[\![q_{11}(T)]\!]^{\phi_{11}}$ is determined by all tuples in $[\![q_{12}(T)]\!]$ with the same job and dept values as t. Since ϕ_{11} states that target entries in $[\![q_{11}(T)]\!]^{\phi_{11}}$ are those belonging to the group $t_O.\mathsf{job}, t_O.\mathsf{dept}$ with $\mathsf{job} < 10$, we derive the predicate $\phi_{12}(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10)$ to describe this group in $[\![q_{12}(T)]\!]$. The result $[\![q_{12}(T)]\!]^{\phi_{12}}$ then contains all tuples contributing to $[\![q_{11}(T)]\!]^{\phi_{11}}$.

- *(Step 4).* Last, we analyze which tuples in T contribute to $[\![q_{12}(T)]\!]^{\phi_{12}}$ from the expression $q_{12} = \mathsf{Select}(T, \mathsf{sal} > 5)$. Similar to step 2, the desired tuples are those satisfying both $\mathsf{sal} > 5$ and ϕ_{12}. We use the predicate $\phi_1(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)$ to describe these target tuples.

Figure 4.5 summarizes the relationship between these predicates: T^{ϕ_1} determines $[\![q_{12}(T)]\!]^{\phi_{12}}$, $[\![q_{12}(T)]\!]^{\phi_{12}}$ determines $[\![q_{11}]\!]^{\phi_{11}}$, $[\![q_{11}]\!]^{\phi_{11}}$ determines $[\![q_1(T)]\!]^{\phi_{10}}$, and $[\![q_1(T)]\!]^{\phi_{10}}$ determines the multiplicity of t_O in the output $[\![q_1(T)]\!]$. It indicates that ϕ_1 specifies the provenance of t_O in T with respect to q_1.

Similarly, we also analyze q_2 using the same output tuple t_O to obtain the predicate $\phi_2(t) = (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)$ describing the provenance of t_O in T with respect to q_2.

4.3.2 Space Refinement

After computing ϕ_1 and ϕ_2, we combine them to find the provenance of t_O with respect to both queries: the provenance tuples are those from T contained by both T^{ϕ_1} and T^{ϕ_2}. We can use the following ϕ to represent them.

$$\begin{aligned}\phi(t) &= \phi_1(t) \vee \phi_2(t) \\ &= (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{dept} = t_O.\mathsf{dept} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5) \\ &\quad \vee (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5) \\ &= (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)\end{aligned}$$

Since t_O used in the analysis is a symbolic tuple, the generated predicate ϕ generalizes to all possible instances of an output tuple. Thus, for any instance of t_O, T^{ϕ} specifies which tuples in T matters for the equivalence checking of q_1 and q_2 with respect to it: if t_O is a distinguishing output tuple for the two queries q_1, q_2 when evaluated on T, then T^{ϕ} is sufficient to replay this difference. In other words, any two input table T_1, T_2 sharing the same fragments that satisfy ϕ would be equivalent with respect to discovering the distinguishing output tuple t_O.

Using ϕ, we construct a new search space $\mathcal{S}'$ from $\mathcal{S}$ such that no two tables share the same provenance tuples for any instances of t_O. This space can be constructed by including only tables T such that $T^{\phi} = T$, so that $T_1^{\phi} = T_2^{\phi} \Rightarrow T_1 = T_2$ for each t_O. The new search space is defined using the following predicate Φ over tables derived from ϕ. (We use $\phi(t, t_O)$ to denote instantiating t, t_O with the provided arguments.)

$$\begin{aligned}\Phi(T) &= \exists t_O. \forall t \in T.\ \phi(t, t_O) \\ &= \exists t_O. \forall t \in T.\ (t.\mathsf{job} = t_O.\mathsf{job} \wedge t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5)\end{aligned}$$

The predicate $\Phi(T)$ specifies that: (1) the table should contain only one job group, and (2) for each tuple t, $t.\mathsf{job} < 10 \wedge t.\mathsf{sal} > 5$. The new search space $\mathcal{S}'$ is then defined as $\{T \mid T \in \mathcal{S} \wedge \Phi(T)\}$. On the other hand, for each table $T \in \mathcal{S}$, T^{ϕ} is contained in $\mathcal{S}'$ for all t_O. Thus, if we can find a counterexample T in $\mathcal{S}$ with distinguishing output tuple t_O, we can also find T^{ϕ} in $\mathcal{S}'$ to reveal the same distinguishing output tuple. Thus, $\mathcal{S}'$ is a smaller yet equivalent space for checking the equivalence between q_1 and q_2.

Figure 4.3 illustrates the relationship between the new search space $\mathcal{S}'$ and $\mathcal{S}$. T_1 and T_2 are both counterexamples for q_1 and q_2 from $\mathcal{S}$, and they both generate the distinguishing

output tuple $t_O = (2, 2, 21)$. Since $\mathcal{S}'$ disallows tuples with multiple groups in the job column, we do not need to consider T_1 in the equivalence checking process. This indicates we can use $\mathcal{S}'$ to speed up the equivalence checking for q_1 and q_2 due to its smaller size.

The refined search space $\mathcal{S}'$ can then be used to encode SMT formulas for bounded verification to determine query equivalence. For instance, encoding $\mathcal{S}'$ along with the queries to SMT formulas and solving them will show that q_1 and q_2 are indeed inequivalent.

4.4 Definitions

In this section we formally define SQL operators and the predicate language we use to describe provenance.

The SQL Language. Figure 4.6 shows the abstract syntax of SQL. A SQL query is formed by compositions of basic operators including Projection, Distinct (for de-duplication), Select (for filtering), Join, Aggr, LeftJoin, Union and Rename on top of base tables. The constructor $\mathsf{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q)$ corresponds to the aggregation query "Select $\bar{c}$, $\overline{\alpha(c_t)}$ From q Group By $\bar{c}$", where $\bar{c}$ are group by keys, $\bar{c}_t$ are columns involved in aggregation, and $\bar{\alpha}$ are aggregation functions used for each column. The constructor $\mathsf{Proj}(\bar{c}, q)$ corresponds to the projection query "Select $\bar{c}$ From q", $\mathsf{Distinct}(q)$ corresponds to "Select Distinct $*$ From q", and others directly corresponds to their concrete forms.

$$
\begin{array}{rcll}
q & ::= & T \mid \mathsf{Proj}(\bar{v}, q) \mid \mathsf{Rename}(q, name, \bar{c}) \mid \mathsf{Select}(q, f) \mid \mathsf{Join}(q_1, q_2) & \text{(Query)} \\
 & \mid & \mathsf{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q) \mid \mathsf{Distinct}(q) \mid \mathsf{Union}(q_1, q_2) \mid \mathsf{LeftJoin}(q_1, q_2, f) & \\
f & ::= & \mathsf{true} \mid \mathsf{false} \mid v \ op \ v \mid f \ \mathsf{And} \ f \mid f \ \mathsf{Or} \ f \mid \mathsf{Not} \ f & \text{(Filters)} \\
v & ::= & c \mid const \mid \mathsf{null} \mid \mathsf{expr}(\bar{v}) & \text{(Values)} \\
\alpha & ::= & \mathsf{max} \mid \mathsf{min} \mid \mathsf{sum} \mid \mathsf{count} \mid \mathsf{count\_distinct} & \text{(Aggregators)} \\
op & ::= & = \mid > \mid < \mid <= \mid >= \mid <> & \text{(Operators)}
\end{array}
$$

Figure 4.6: The abstract syntax of SQL, where c ranges over column names, v over values, q over queries and f over filter predicates. We use expr for arithmetic operations.

Predicates. Figure 4.7 defines the predicate language for describing data provenance. A predicate ϕ is formed from compositions of primitives $v \ op \ v$, where a value v is either a reference of a tuple ($t.i$ represents the i-th element in the tuple t, $t.c$ is the same but refers the tuple entry by name c), an expression composed from operators like $+$, $-$ or function application. In our paper, symbolic tuples in predicates are allowed, and we use the notation $\phi(t_1, t_2)$ to denote substituting symbolic tuples with input tuples t_1, t_2.

$$
\begin{array}{rcl}
\phi & := & \mathsf{true} \mid \mathsf{false} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid v \ op \ v \\
v & := & t.i \mid t.c \mid const \mid \mathsf{expr}(\bar{v}) \\
op & := & = \mid > \mid < \mid <= \mid >= \mid <>
\end{array}
$$

Figure 4.7: The predicate language, where t ranges over tuples and c over column names.

We also use the notation T^{ϕ} to denote filtering a table by keeping only those tuples in T that satisfy $\phi(x)$. Formally, $T^{\phi} = \{\{t \mid t \in T \wedge \phi(t)\}\}$ (we use $\{\{\cdot\}\}$ to represent a bag).

4.5 Algorithm

We next formally describe the space refinement algorithm. First, we introduce the provenance analysis algorithm (Section 4.5.1). Then, we present how we use the analysis result to conduct search space refinement for the query equivalence checking problem (Section 4.5.2). Without loss of generality, we assume queries refer only to one input table T to simplify notation.

4.5.1 Symbolic Provenance Analysis

Given a query q, the provenance analysis algorithm computes the provenance of a symbolic tuple t_O with respect to q. Taking q and t_O as input, the analysis returns a predicate ϕ describing which tuples in input table T contribute to the multiplicity of t_O in the query output.

Definition 3. (Provenance Predicate) ϕ is a provenance predicate for query q if the following property is satisfied:

$$\forall t_O.\forall T.\ [\![q(T)]\!]t_O = [\![q(T^{\phi(t,t_O)})]\!]t_O.$$

In other words, the multiplicity of the tuple t_O in the output of q is unchanged even though input table T is restricted to only tuples t that satisfy property $\phi(t, t_O)$. By definition, a query q has multiple provenance predicates, including the trivial predicate true. To compute a non-trivial predicate, we perform a backward analysis on q by propagating constraints from a tuple t_O in the query output back to the input, as shown in Section 4.3.

The Algorithm.

Key to the backward analysis is constructing a predicate ϕ that describes which tuples in T contribute to the multiplicity of the output tuple $t_O \in [\![q(T)]\!]$. To do so, the backward analysis first constructs a predicate over q to specify which tuples in q determine the multiplicity of t_O in the output and then propagates it to its subexpressions until reaching the input table T (the leaf subexpression).

Provenance Analysis. We compute the provenance predicate of a query q with respect to a symbolic output tuple $t_O \in [\![q(T)]\!]$ in the following three steps:

1. *(Initialization).* Construct the predicate ϕ_0 to be $\phi_0 = \bigwedge_{i=1}^{n}(t.i = t_O.i)$. This initial predicate states that the provenance of t_O in the output table $[\![q(T)]\!]$ is itself.

2. *(Propagation).* For each query $q_1 = \mathsf{op}(q_2)$ (or $q_1 = \mathsf{op}(q_2, q_3)$) where op is a SQL operator defined in Figure 4.6, we propagate the provenance predicate ϕ_1 over q_1 to its subquery q_2. The result is a predicate ϕ_2 over q_2 specifying which tuples in $[\![q_2(T)]\!]$ are sufficient to decide multiplicities of tuples in $[\![q_1(T)]\!]^{\phi_1}$. Propagation rules are shown in Figure 4.8.

   - We use the notation $(q_1 \sim \phi_1)$ to describe that ϕ_1 is the provenance predicate computed at the query q_1 (i.e., $[\![q_1(T)]\!]^{\phi_1}$ is the provenance of the output tuple t_O with respect to $[\![q_1(T)]\!]$), and we use $(q_1 \sim \phi_1) \rightsquigarrow (q_2 \sim \phi_2)$ to describe the propagation of ϕ_1 over q_1 to the predicate ϕ_2 over the subquery q_2.
   - Given a query $q_1 = \mathsf{op}(q_2, q_3)$ and a predicate ϕ_1 over q_1, the rule $(q_1 \sim \phi_1) \rightsquigarrow (q_2 \sim \phi_2) \wedge (q_3 \sim \phi_3)$ produces ϕ_2 and ϕ_3 specifying which tuples in $[\![q_1(T)]\!]$ and $[\![q_2(T)]\!]$ determine multiplicities of tuples in $[\![q_1(T)]\!]^{\phi_1}$. The conjunction states that $[\![q_2(T)]\!]^{\phi_2}$ and $[\![q_3(T)]\!]^{\phi_3}$ together determine $[\![q_1(T)]\!]^{\phi_1}$.

   The propagation process terminates when all queries in the expression are T (i.e., reaching leaf nodes of the AST), since the remaining AST depth of the algorithm decreases over analysis steps. The propagation process can be presented as $(q \sim \phi_0) \rightsquigarrow \cdots \rightsquigarrow \bigwedge_k (T \sim \phi_k)$, where the final state shows which tuples in T determine multiplicities of tuples in $[\![q(T)]\!]^{\phi_0}$. The conjunction results from the fact that table T may appear multiple times in different subqueries of q.

3. *(Merge).* The final step is to resolve the expression $\bigwedge_k (T \sim \phi_k)$ to obtain a provenance predicate ϕ over T s.t. T^{ϕ} determines $[\![q(T)]\!]^{\phi_0}$. According to the merge rule (Figure 4.9), we construct ϕ as $\phi = \bigvee_k \phi_k$. The correctness of ϕ is shown by Lemma 4.

Figure 4.8 and Figure 4.9 show the analysis rules. We describe these rules below.

- *(Select).* Given a query $q = \mathsf{Select}(q_1, f)$ and a predicate ϕ, the predicate ϕ_1 over q_1 is the conjunction of ϕ with a predicate formed by replacing every column reference c_i in the filter predicate f with $t.i$. The idea is that tuples in $[\![q_1(T)]\!]$ satisfying $f \wedge \phi$ alone are sufficient to determine $[\![q(T)]\!]^{\phi}$. For example, if $q = \mathsf{Select}(q_1, c_1 < 5)$ and

$\phi = (t.2 > 1)$, then the tuples in $[\![q_1(T)]\!]$ satisfying $\phi_1 = (t.2 > 1 \wedge t.1 < 5)$ determine tuples in $[\![q(T)]\!]$ satisfying ϕ.

- *(Projection).* Given a query $q = \mathsf{Proj}(\bar{v}, q_1)$ and a predicate ϕ, we compute the predicate ϕ_1 by substituting column references in ϕ with expressions specified by $\bar{v}$, followed by abstracting column names using tuple expression $t.i$. For example, given a query $q = \mathsf{Proj}(c_3 + 2, c_1,\ q_1)$ (corresponding to Select $c_3 + 2, c_1$ From q_1) and $\phi = (t.1 = 1 \wedge t.2 > 1)$, the output predicate ϕ_1 for q_1 is $\phi_1 = (t.3 + 2 = 1 \wedge t.1 > 1)$. The rationale is that the multiplicity of the tuple (x_1, x_2) in $[\![q(T)]\!]$ is determined by all tuples t in $[\![q_1(T)]\!]$ such that $t.3 + 2 = x_1$ and $t.1 = x_2$.

$\boxed{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1) \wedge ...}$

$$\frac{q = T}{(q \sim \phi) \rightsquigarrow (T \sim \phi)}\ \text{(Table)} \qquad \frac{q = \mathsf{Select}(q_1, f) \quad \mathsf{schema}(q_1) = \bar{c}}{(q \sim \phi) \rightsquigarrow \big(q_1 \sim \phi \wedge \big[c_i \mapsto t.i\big] f\big)}\ \text{(Select)}$$

$$\frac{q = \mathsf{Distinct}(q_1)}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi)}\ \text{(Distinct)} \qquad \frac{q = \mathsf{Proj}(\bar{v}, q_1) \quad \mathsf{schema}(q_1) = \bar{c}}{(q \sim \phi) \rightsquigarrow \Big(q_1 \sim \Big[t.i \mapsto \Big(\big[c_j \mapsto t.j\big] v_i\Big)\Big] \phi\Big)}$$

$$\frac{q = \mathsf{Join}(q_1, q_2) \quad \phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \quad |\mathsf{schema}(q_1)| = n_1 \quad \mathsf{colRef}(\phi_i) \cap \mathsf{schema}(q_i) = \emptyset\ {}_{(i=1,2)}}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1) \wedge (q_2 \sim [t.i \mapsto t.(i - n_1)]\phi_2)}\ \text{(Join)}$$

$$\frac{q = \mathsf{Union}(q_1, q_2)}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi) \wedge (q_2 \sim \phi)}\ \text{(Union)} \qquad \frac{q = \mathsf{Rename}(q_1, name, \bar{c})}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1)}\ \text{(Rename)}$$

$$\frac{q = \mathsf{LeftJoin}(q_1, q_2, f) \quad \phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \quad \mathsf{colRef}(\phi_i) \cap \mathsf{schema}(q_i) = \emptyset\ {}_{(i=1,2)}}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1) \wedge (q_2 \sim \mathsf{true})}\ \text{(LeftJoin)}$$

$$\frac{\begin{array}{c} q = \mathsf{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q_1) \\ \phi = \phi_1 \wedge \phi_2 \quad \mathsf{colRef}(\phi_1) \subseteq \{\bar{c}\} \end{array}}{(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1)}\ \text{(Aggr)} \qquad \frac{(q_i \sim \phi_i) \rightsquigarrow \bigwedge_k (q'_{ik} \sim \phi'_{ik})}{\bigwedge_i (q_i \sim \phi_i) \rightsquigarrow \bigwedge_{i,k} (q'_{ik} \sim \phi'_{ik})}\ \text{(Step)}$$

Figure 4.8: Propogation rules. Each propagation step $(q \sim \phi) \rightsquigarrow (q_1 \sim \phi_1)$ computes the constraint ϕ_1 that specifies which tuples in $[\![q_1(T)]\!]$ are sufficient to determine the multiplicities of tuples in $[\![q(T)]\!]$ that satisfy constraint ϕ. The auxiliary function colRef returns the column a predicate refers to, and the function schema extracts the output schema of a query.

$$\frac{}{\bigwedge_i (T \sim \phi_i) \rightsquigarrow (T \sim \bigvee_i \phi_i)} \text{(Merge)}$$

Figure 4.9: The merge rule. It computes the provenance constraint of the table T by merging contraints obtained from backward analysis.

- *(Join).* To propagate ϕ through $q = \mathsf{Join}(q_1, q_2)$, the rule breaks ϕ into $\phi_1 \wedge \phi_2 \wedge \phi_3$, where ϕ_1 contains terms that involve only columns from q_1, ϕ_2 contains terms that involve only columns from q_2, and ϕ_3 contains terms that that involve both. Then, ϕ_1 is propagated to q_1, ϕ_2 is propagated to q_2, and ϕ_3 is discarded. Discarding ϕ_3 allows the analysis continues independently in different branches, thus reducing the complexity of the resulting predicate. This approximation retains the soundness of the analysis: since $[\![q_1(T)]\!]^{\phi_1} \times [\![q_2(T)]\!]^{\phi_2}$ subsumes $\left([\![q_1(T)]\!]^{\phi_1} \times [\![q_2(T)]\!]^{\phi_2}\right)^{\phi_3}$ and the latter is sufficient to determine $[\![q(T)]\!]^{\phi}$, $[\![q_1(T)]\!]^{\phi_1} \times [\![q_2(T)]\!]^{\phi_2}$ is sufficient to determine $[\![q(T)]\!]^{\phi}$ as well. On the other hand, discarding ϕ_3 weakens the provenance predicate we obtain in the final result, which could limit the pruning power of the final provenance predicate. We leave a detailed discussion of the trade-off between analysis performance and pruning power to the end of this section. For example, the propagation of $\phi = (t.1 = a_1 \wedge t.2 > t.3 \wedge r.3 < 5)$ through query $q = \mathsf{Join}(q_1, q_2)$ is computed by first constructing $\phi_1 = (t.1 = a_1)$ for q_1, $\phi_2 = (t.1 < 5)$ for q_2, and discarding $\phi_3 = (t.2 > t.3)$ (since it refers to columns from both subqueries).

- *(Aggregate).* To propagate a predicate ϕ over an aggregation query $q = \mathsf{Aggr}(\bar{c}, \bar{\alpha}, \bar{c}_t, q_1)$ to its subquery q_1, we first split ϕ into $\phi_1 \wedge \phi_2$, where column references in ϕ_1 are limited to grouping columns $\bar{c}$, and then set ϕ_1 as the target predicate for q_1. Similar to the rule for Join, this is an approximation of the aggregation semantics, it guarantees that all tuples in each group satisfy ϕ_1 are retained so that $[\![q_1(T)]\!]^{\phi_1}$ is sufficient to compute aggregation results in $[\![q(T)]\!]^{\phi}$. We discard ϕ_2 to retain analysis efficiency.

- *(Union, Rename).* The predicates ϕ_1, ϕ_2 for the subqueries are the same as ϕ.

- *(LeftJoin).* Unlike for Join, the predicate for the subquery q_2 in LeftJoin is set to true instead of ϕ_2. This difference is introduced by the non-monotonicity of Left Join, i.e., given $T = [\![\mathsf{LeftJoin}(T_1, T_2)]\!]$, if we remove a tuple from T_2 (denoted as T_2^-), the result $[\![\mathsf{LeftJoin}(T_1, T_2^-)]\!]$ is not subsumed by T. The enforcement of using the predicate true for q_2 in our rule design is a conservative way to preserved all tuples contributing to tuples in $[\![q(T)]\!]$.

Properties of the Analysis Algorithm

We now demonstrate properties of the provenance analysis algorithm. Lemma 2 states that each step of the analysis correctly propagates the provenance predicate to its subquery (subqueries). Lemma 3 states that weakening any provenance predicate generated by the algorithm still results in a provenance predicate. The weakening property allows us to generate not necessarily the strongest but sound constraints, and it guarantees the correctness of the merge rule, where predicates ϕ_k's are weakened to their disjunction $\bigvee_k \phi_k$. Finally, Lemma 4 shows that the provenance analysis algorithm returns a provenance predicate over T for the symbolic output tuple t_O.

Lemma 2. For each propagation rule $(q \sim \phi) \rightsquigarrow \bigwedge_k (q_k \sim \phi_k)$, the following property holds:

$$\forall T, T'.\ \left(\bigwedge_k \left([\![q_k(T)]\!]^{\phi_k} = [\![q_k(T')]\!]^{\phi_k}\right)\right) \Longrightarrow \left([\![q(T)]\!]^{\phi} = [\![q(T')]\!]^{\phi}\right)$$

Proof Sketch. By design (details are shown in the rule explanations in Section 4.5.1), every rule guarantees that tuples excluded from $[\![q_1(T)]\!]$ by ϕ_i do not contribute to multiplicities of tuples in $[\![q(T)]\!]^{\phi}$. □

Lemma 3 (Weakening)**.** For each propagation rule $(q \sim \phi) \rightsquigarrow \bigwedge_k (q_k \sim \phi_k)$, if $\forall t.(\phi_k \Rightarrow \phi'_k)$ holds for all k, then the following property holds:

$$\forall T, T'.\ \left(\bigwedge_k \left([\![q_k(T)]\!]^{\phi'_k} = [\![q_k(T')]\!]^{\phi'_k}\right)\right) \Longrightarrow \left([\![q(T)]\!]^{\phi} = [\![q(T')]\!]^{\phi}\right)$$

Proof Sketch. The weakening property for monotonic queries operators (Select, Proj, Join) is obvious, since appending extra rows not satisfying the original constraint do not affect $[\![q(T)]\!]^{\phi}$. For non-monotonic operators Aggr, the rule ensures that the whole a whole group would either all be included or none get included for each Group-by group; therefore, relaxing the constraint ϕ_i does not introduce new entries in the same group. For LeftJoin, the rule disallows propagation of the predicate to the right hand side query q_2 (the predicate is true, which can no longer be weakened); therefore, no new tuples with null placeholders will be introduced in the result. □

Lemma 4 (Soundness)**.** For a given query q whose output schema size is n, let $\phi_0 = \bigwedge_{i=1}^{n}(t.i = t_O.i)$, assume $(q \sim \phi_0) \rightsquigarrow \cdots \rightsquigarrow \bigwedge_k (T \sim \phi_k)$ where t_O is a symbolic output tuple; then, $\phi = \bigvee_k \phi_k$ is a provenance predicate over T with respect to the tuple t_O in $[\![q(T)]\!]$.

Proof Sketch. By induction on the propagation rules, we can apply Lemma 2 to show that $T^{\phi_1}, \ldots, T^{\phi_n}$ together determine the multiplicity of t_O in $[\![q(T)]\!]$. Then by the weakening property, we can weaken every ϕ_i to $\bigvee_k \phi_k$ while retaining soundness. Thus, $\bigvee_k \phi_k$ is a provenance predicate over T for $[\![q(T)]\!]^{\phi_0}$ (i.e., the tuple t_O in $[\![q(T)]\!]$). □

The Effect of Analysis Approximation.

Note that our inference algorithm provides a sound provenance predicate but *does not* produce the strongest provenance predicate at each analysis step, i.e., the property in Lemma 2 does not hold if we flip the "$\Longrightarrow$" into "$\Longleftarrow$". This is because the propagation process does not use the full semantics of SQL in the analysis process for the purpose of improving analysis efficiency. For example, the Join propagation rule does consider join predicates that refer to columns in both tables (by discarding ϕ_3); as a result, $[\![q_1(T)^{\phi_1}]\!]$ and $[\![q_2(T)^{\phi_2}]\!]$ can include tuples that have no matching tuples in the other table. Similarly, the Aggr propagation rule does not consider the semantics of aggregation functions, and it conservatively keeps all tuples in $[\![q_1(T)]\!]$ that are in the same groups as tuples in $[\![q(T)]\!]^{\phi}$. This design decision trades-off between the cost of provenance analysis and the effectiveness of the generated search space: we could ask a solver to find the strongest provenance predicate for the tuple t_O in $[\![q(T)]\!]$, but its computation time would be the same as directly asking the solver to solve the equivalence problem, making the analysis pointless. As we will show in Section 4.6, although the generated provenance predicate is not the strongest, it is highly effective in speeding up various symbolic SQL reasoning tasks.

4.5.2 Space Refinement

We now introduce how to refine the search space for query equivalence checking between two queries q_1 and q_2 using their provenance predicates ϕ_1 and ϕ_2 over input table T.

Merge Predicates. We first merge ϕ_1 and ϕ_2 to form a provenance predicate that is sound for both tables using the MergeRule in Figure 4.9. Since ϕ_1 specifies which tuples in T decide the multiplicity of t_O in $[\![q_1(T)]\!]$, and ϕ_2 specifies which tuples in T decide the multiplicity of t_O in $[\![q_1(T)]\!]$, combining them yields a predicate $\phi = \phi_1 \vee \phi_2$ that is sufficient to determine the multiplicity of t_O in both outputs of q_1 and q_2, as guaranteed by the weakening property (Lemma 3). Formally, the merged predicate ϕ holds the following property (recall that ϕ is shorthand for $\phi(t, t_O)$, which is a function over t_O).

$$\forall T, t_O. \left([\![q_1(T)]\!]t_O = [\![q_1(T^{\phi})]\!]t_O\right) \wedge \left([\![q_2(T)]\!]t_O = [\![q_2(T^{\phi})]\!]t_O\right)$$

Space Refinement. Given a table T and an output tuple t_O, the provenance predicate ϕ is a predicate over tuples that determines which tuples in T are sufficient to decide the multiplicity of t_O in both query outputs. Next, we lift ϕ from a predicate over tuples to a predicate over tables to refine the search space. We define a *table predicate* as:

$$\Phi(T) = \exists t_O . \forall t \in T \; . \; \phi(t, t_O)$$

Given a table search space $\mathcal{S}$, we construct a new search space $\mathcal{S}'$ using Φ as follows:

- If the space $\mathcal{S}$ satisfies the property that $\forall T, T' \; . \; T' \in \mathcal{S} \wedge T \subseteq T' \Rightarrow T \in \mathcal{S}$ (i.e., $\mathcal{S}$ is closed under containment):
$$\mathcal{S}' = \{T \in \mathcal{S} \mid \Phi(T)\}$$

- Otherwise, we first construct the closure $\mathcal{S}^* = \{T \mid \exists T' \in \mathcal{S} \wedge T \subseteq T'\}$ and then apply the above with $\mathcal{S} = \mathcal{S}^*$.

Intuitively, the new search space $\mathcal{S}'$ contains only one representative from each equivalence class of tables with respect to input queries: i.e., for each instantiation of output tuple t_O, if $T_1^\phi = T_2^\phi$ (ϕ also instantiated with t_O), only one table remains in the new search space $\mathcal{S}'$. For the second case, we construct the closure of $\mathcal{S}^*$ since the fragment contributing to an output tuple may not be contained in $\mathcal{S}$. For example, if $\mathcal{S}$ contains only tables with exactly 2 distinct tuples (and not containing those with 1 tuple), the table fragment identified by a provenance predicate might consist of only one tuple and not be contained in $\mathcal{S}$.

As shown in Section 4.6, we use Φ to identify equivalent tables in the search space $\mathcal{S}$ to speed up query equivalence checking. We formally state the relationship between $\mathcal{S}$ and $\mathcal{S}'$ below.

Lemma 5 (Removing Redundancy)**.** Given q_1, q_2, a search space $\mathcal{S}$, and $\mathcal{S}'$ is the refined search space constructed from $\mathcal{S}$ using a table constraint. If there exists a table $T \in \mathcal{S}$ such that $[\![q_1(T)]\!] \neq [\![q_2(T)]\!]$, then there also exists a table $T' \in \mathcal{S}'$ such that $[\![q_1(T')]\!] \neq [\![q_2(T')]\!]$.

Proof Sketch. Assume that $T \in \mathcal{S}$ is a counterexample with distinguishing output t_O for q_1 and q_2 (with multiplicities m_1, m_2, respectively). Then, $T^{\phi(t,t_O)}$ is a table from $\mathcal{S}'$. According to Def. 3, applying q_1, q_2 on $T^{\phi(t,t_O)}$ also results in m_1, m_2 as the multiplicities of t_O. This shows that $T^{\phi(t,t_O)}$ is also a counterexample for q_1, q_2. □

This property guarantees that if we fail to find a counterexample in $\mathcal{S}'$ for two queries q_1 and q_2, then q_1 and q_2 are guaranteed to be equivalent in $\mathcal{S}$. We show how to encode the refined search space in Section 4.8.

4.6 Evaluation

We evaluate the effectiveness of our space refinement algorithm on three tools that reason about tables and queries for different purposes: (1) bounded verification [34], (2) test data generation (in the context of mutation testing[24], auto-grading [74]) and unit test data generation [189], and (3) concolic testing [179]. In each scenario, we run the tool on benchmarks extracted from the original paper with and without space refinement, and compare performance differences. In addition, we run each experiment twice with Cosette encoding [34] and Qex encoding [189] to demonstrate that our space refinement algorithm is general to different underlying solver implementations.

4.6.1 Experiment 1: Bounded Verification

We first study the space refinement algorithm on bounded verification. To do so, we use 46 benchmarks collected from the 232 test cases for the SQL rewrite rules in the Apache Calcite project,[3] an open source query optimization framework used by many database systems. Cases excluded from the benchmark are either those testing non-SQL feature or those containing features that are currently not support by Cosette and Qex (e.g., Partition, Order-By, Case and In). These 46 benchmarks contain non-trivial use of SQL operators: 29 cases contain queries with more than 5 subqueries, and 41 cases involve tables with more than 9 columns.

For each benchmark and each encoding method (i.e., Cosette and Qex), we chose the verification bound as the size of the maximum search space that the solver can completely explore within 600 seconds without space refinement. We next re-run the solver on the same bound but with space refinement. We measure the search space based on the number of symbolic values used in the encoding and report the relative performance with and without space refinement for each encoding approach.

This experiment helps us answer how the refinement algorithm affects the bounded verification process of different types of query, different search space size and different underlying solver choices.

Conclusion 1. *Queries with aggregations benefit most from space refinement.*

As shown in Table 4.1, 19 out of the 21 cases with aggregations show significant speedup in the bounded verification task, using both Qex and Cosette encodings. The medium is a $48\times$ speedup for Cosette and a $58\times$ speedup for Qex. The speedup mainly comes from the reduction of the number of groups that the solver needs to consider while using the refined search space. In such cases, the refinement algorithm determines it is sufficient to encode

[3] https://calcite.apache.org

only a small number of groups to prove the equivalence between the given queries. The two cases that didn't benefit from the refinement algorithm are *boolean queries*, i.e., queries of the form "Select 1 From . . . ". In these cases, query inputs only affect how many "1"s are returned, and the provenance analysis algorithm can not propagate the initial predicate to its subqueries in a non-trivial way. As a result, the space refinement algorithm determines that *all* tuples in the input table are necessary to determine the output tuple (the tuple (1)) multiplicity.

For the other 25 cases that do not involve aggregation, only 6 cases display significant speedup. The other 19 cases are either unaffected or slowed down (minor slow-down with less than 10% time difference). These 25 cases are unions of conjunctive queries (UCQs) constructed from Select, Join, and Union operators. In these cases, since the query semantics does not introduce interactions among tuples, the underlying symbolic evaluator and SMT solver can also exploit the independence among different tuples when solving generated constraints. As a result, such queries benefit less from space refinement. The speedup achieved in the 6 non-aggregation cases comes from backward constant propagation, i.e., the propagation constants appeared in query predicates to input tables; this propagation allows us to preassign values to certain parts of the symbolic table, which reduces the number of symbolic values need to encode the search space. For example, given the query "Select . . . Where $c = 10$", the analysis algorithm propagates the constant 10 from the predicate to the input table through a provenance predicate $t.1 = 10$, which frees us from using symbolic values for the column c in the input table.

Conclusion 2. *The benefits of space refinement generalize to different encoding methods and different query sizes.*

While the speedup varies for different encoding methods, i.e., Cosette v.s. Qex, whether a pair of queries benefits from space refinement is not affected. All cases in Table 4.1 and Table 4.2 with over $2\times$ speedups display under Cosette encoding also display a noticeable improvement under Qex encoding. Also, compared to query structural differences (e.g., whether the target query uses aggregation or contains constants), the differences in query sizes have little influence on the amount of speedup gained from space refinement.

Illustrative Examples. We present two examples below to demonstrate the strengths and limitations of the space refinement algorithm. Both examples run on the following tables:

```
Dept(deptno:int, name:str)
Emp(empno:int, ename:str, job:int, mgr:int, hiredate:int,
    comm:int, sal:int, deptno:int, slacker:int)
```

- *(PushFilterPastAggGroupSets2).* In this example, the refinement algorithm produces the provenance predicate $\phi(t, t_O) = (t.\mathsf{name} = \text{“}Charlie\text{”} \wedge t.\mathsf{name} = t_O.1 \wedge t.\mathsf{deptno} = t_O.2)$.

Calcite (with aggregates)	#sq	Qex Encoding				Cosette Encoding			
		#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup
PushFilterPastAgg	3	14	33.44	0.26	130.0	15	107.61	0.26	410.0
PushFilterPastAggTwo	3	16	195.55	0.69	280.0	19	154.43	0.56	280.0
AggConstKeyRule3	3	63	115.1	0.72	160.0	59	92.89	0.47	200.0
PullFilterThroughAgg	3	63	58.92	0.69	86.0	68	119.08	0.7	170.0
PushFilterPastAggGroupSets2	3	16	58.67	0.73	81.0	21	121.18	0.74	160.0
PullFilterThroughAggGroupSets	3	54	49.59	0.58	86.0	59	59.97	0.57	100.0
PullAggThroughUnion	6	45	61.83	1.06	58.0	50	92.28	1.15	81.0
AggProjectPullUpConsts	2	45	84.72	1.47	58.0	28	36.55	0.61	60.0
PushAvgThroughUnion	6	63	86.58	3.05	28.0	68	188.93	3.79	50.0
PushAggThroughJoin1	6	33	221.71	5.65	39.0	38	263.84	5.5	48.0
PushFilterPastAggGroupSets1	2	24	163.09	1.97	83.0	27	60.4	1.73	35.0
AggConstKeyRule2	2	108	69.82	2.68	26.0	113	76.72	2.63	29.0
PushFilterPastAggThree	2	117	128.88	3.75	34.0	113	87.62	3.07	29.0
PushAvgGroupSetsThroughUnion	6	63	215.38	3.47	62.0	59	54.38	2.05	27.0
PushAggThroughJoin3	5	35	32.14	3.19	10.0	38	28.32	2.47	11.0
AggProjectMerge	2	99	149.11	2.55	58.0	82	120.19	12.21	9.8
AggGroupSetsProjectMerge	2	99	148.58	2.58	58.0	82	120.35	12.26	9.8
PushAggThroughJoinDistinct	6	35	146.25	11.69	13.0	29	57.0	6.29	9.1
AggConstKeyRule	2	54	97.12	3.8	26.0	50	12.89	1.95	6.6
TransitiveInferAgg	6	63	69.06	69.56	0.99	59	42.0	40.44	1.0
TransitiveInferJoin3wayAgg	9	45	106.76	107.36	0.99	59	35.56	37.64	0.94

Table 4.1: The evaluation result for bounded verification on Calcite benchmarks (cases with aggregations). Column #sq refers to the number of subqueries in the target query for measuring complexity, #SV refers to the number of symbolic values used in encoding the search space, and $t_{\mathcal{S}}$, $t_{\mathcal{S}'}$ refer to the time spent by the solver without and with space refinement.

The predicate determines that (1) we need to consider only the group with name '*Charlie*', and (2) we only need to consider one department group. In this way, the solver no longer needs to consider the all possible ways to group the name and deptno columns (which is exponential to the input table size). This result in a speedup of 160×, as shown in Table 4.1.

```
-- q1
Select name, deptno, Count(*)
From Dept
Group By name, deptno
Having name = 'Charlie';
```

```
-- q2
Select t2.name, t2.deptno, Count(*)
From   (Select name, deptno
        From Dept) As t2
Where  t2.name = 'Charlie'
Group By t2.name, t2.deptno;
```

- *(TransitiveInferenceJoin3Way).* This example shows a boolean query that does not benefit from space refinement. Since both these query outputs are tables consisting of tuples with content 1, our algorithm generates only a trivial provenance predicate $\phi(t, t_O) = (t.\text{deptno} > 7 \wedge t_O.1 = 1)$ that cannot effectively reduce complexity of encoding the search space.

Calcite (without aggregates)	#sq	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup
PullConstIntoProject	2	126	98.26	0.52	190.0	122	71.05	0.5	140.0
PullConstIntoFilter	3	171	45.45	0.88	52.0	176	54.96	0.89	62.0
RemoveSemiJoinFilter	5	79	55.14	0.85	65.0	38	0.38	0.04	9.7
RemoveSemiJoinRightFilter	7	46	57.31	1.36	42.0	26	0.35	0.04	7.8
MergeJoinFilter	5	46	54.54	0.44	120.0	37	1.12	0.15	7.4
MergeFilter	3	290	5.73	6.88	0.83	283	10.65	7.03	1.5
TransitiveInferUnion3way	13	54	106.9	108.8	0.98	32	0.95	0.96	1.0
PushJoinThroughUnionOnRight	10	72	62.92	63.3	0.99	59	7.73	7.72	1.0
PullConstThroughUnion3	6	1188	7.15	7.07	1.0	1157	7.52	7.36	1.0
TransitiveInferJoin	6	90	140.83	142.33	0.99	68	47.35	46.93	1.0
PushJoinCondDownToProject	6	222	37.77	38.37	0.98	227	38.45	37.28	1.0
TransitiveInferUnion	9	63	79.72	78.27	1.0	41	11.99	11.93	1.0
TransitiveInferJoin3way	9	81	195.8	200.27	0.98	41	0.59	0.59	1.0
TransitiveInferUnionAlwaysTrue	10	45	113.56	113.5	1.0	32	0.91	0.91	1.0
TransitiveInferProject	6	90	145.19	145.35	1.0	68	52.94	52.16	1.0
TransitiveInferComplexPredicate	7	63	100.72	101.25	0.99	32	172.81	170.94	1.0
RemoveSemiJoinRight	6	101	69.02	68.49	1.0	103	71.41	70.37	1.0
TransitiveInferConjInPullUp	6	72	69.42	69.3	1.0	41	7.08	7.04	1.0
PushJoinThroughUnionOnLeft	10	72	78.67	78.44	1.0	59	10.11	10.05	1.0
ExtractJoinFilterRule	4	418	18.12	18.52	0.98	423	17.48	17.25	1.0
TransitiveInferPullUpThruAlias	6	45	8.99	9.18	0.98	41	107.75	105.92	1.0
SemiJoinReduceConsts	8	234	44.52	43.97	1.0	239	44.57	44.88	0.99
PushProjectPastSetOp	6	972	8.17	8.51	0.96	959	8.14	8.44	0.96
PullConstThroughUnion	6	918	8.39	8.96	0.94	910	8.55	8.98	0.95
PullConstThroughUnion2	5	909	8.43	8.68	0.97	905	10.15	11.2	0.91

Table 4.2: The evaluation result for bounded verification on Calcite benchmarks (cases without aggregations). Column #sq refers to the number of subqueries in the target query, #SV refers to the number of symbolic values, and $t_{\mathcal{S}}$, $t_{\mathcal{S}'}$ refer to the time spent by the solver without and with space refinement.

```
-- q1
Select 1
From (Select * From emp
      Where emp.deptno > 7) As t
Join emp As EMP0
On t.deptno = EMP0.deptno
Join emp As EMP1
On EMP0.deptno = EMP1.deptno;
```

```
-- q2
Select 1
From (Select * From emp
      Where  deptno > 7) As t1
Join (Select * From emp
      Where deptno > 7) As t2
On t1.deptno = t2.deptno
Join (Select * From emp
      Where deptno > 7) As t3
On t2.deptno = t3.deptno;
```

In sum, the search space refinement algorithm effectively speeds up bounded verification of an important fraction of complex queries.

4.6.2 Experiment 2: Test Data Generation

Our second experiment studies how the space refinement algorithm can be used to improve test data generation tasks, including: (1) generating inputs to disambiguate non-equivalent

query pairs (for mutation testing and auto-grading) and (2) generating unit test inputs for the given query such that the query returns a non-empty output [189].

In this experiment, we use 13 query disambiguation benchmarks and 2 unit test generation benchmarks from prior work. In the benchmark collection phase, we exclude cases whose distinguishing input tables are those with only 1 tuple, as all such cases can be solved within 0.2 seconds by current solvers and do not present scalability challenges in terms of search space size. For each benchmark, we measure the time each solver it takes (Cosette, Qex) to find the first desirable model with and without search space refinement.

Conclusion 3. *The benefit of space refinement is limited when the target model size is small.*

As shown in Table 4.3, the refined space results in speedups for 7 cases under Qex encoding and 6 cases under Cosette encoding. These cases are harder cases whose solutions require more tuples. Other cases are either unaffected or show an insignificant (< 0.5 seconds) slowdown. The speedup is limited in these benchmarks because most cases requires tables only with 2 distinct tuples to disambiguate. As a result, the benefit of space size reduction does not compensate for the overhead of encoding the refinement predicate.

Case	#Q	Qex Encoding				Cosette Encoding			
		#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup	#SV	$t_{\mathcal{S}}$(s)	$t_{\mathcal{S}'}$(s)	Speedup
mutant-1	8	16	0.28	0.34	0.83	18	0.49	0.08	6.0
mutant-2	4	15	2.06	0.98	2.1	17	2.65	1.83	1.4
mutant-3	7	36	12.46	8.3	1.5	43	37.74	27.8	1.4
mutant-4	7	30	7.26	6.93	1.0	37	7.16	6.69	1.1
mutant-5	8	38	0.33	0.33	1.0	45	0.65	0.65	1.0
mutant-6	7	18	1.59	1.61	0.99	25	4.3	4.27	1.0
hw-1	5	8	0.46	0.47	0.97	12	1.08	1.07	1.0
mutant-7	13	6	9.12	9.46	0.96	5	0.56	0.56	1.0
mutant-8	5	8	0.3	0.29	1.0	15	0.27	0.27	1.0
hw-2	5	8	0.31	0.31	1.0	12	0.24	0.24	0.99
mutant-9	5	16	0.15	0.14	1.1	23	0.19	0.2	0.95
hw-3	4	21	53.51	17.99	3.0	18	2.2	2.31	0.95
hw-4	4	16	1.69	0.58	2.9	18	6.85	7.58	0.9
unit-test-1	4	13	0.12	0.1	1.1	13	0.22	0.16	1.3
unit-test-2	4	14	3.55	1.98	1.8	8	0.46	0.39	1.2

Table 4.3: The evaluation result for test data generation benchmarks. Column #sq refers to the number of subqueries in the target query for measuring complexity, #SV refers to the number of symbolic values used in encoding the search space, and $t_{\mathcal{S}}$, $t_{\mathcal{S}'}$ refer to the time spent by the solver without and with space refinement.

4.6.3 Experiment 3: Concolic Testing

Last, we demonstrate that the space refinement algorithm can speed up cocolic testing of relational queries. In this experiment, we manually translate the following two pairs of representative SQL queries (q_1, q_2 and q_3, q_4) into Java programs and call the CATG concolic testing engine to test whether the two queries are equal.

```
-- q1, q2
Select id, Sum(val) From Flight Where year > 2010 Group By fid;
Select fid, year From Flight Where year > 2010 Group By fid, year;

-- q3, q4
Select fid, year From Flight Where year > 2010 And fid = carrier;
Select fid, year From Flight Where year > 2015 And fid = carrier;
```

Given a pair of queries, we create a Java snippet shown below, where the two queries take as input a randomly initialized concolic table. Then, we run the CATG concolic testing engine on the Java snippet and log the time spent by the concolic tester to run 20 test iterations. We set the size of the symbolic input table (the number of tuples in the table) as a variable and study the performance of the concolic tester.

```
... // input and query definition
if (tableEqual(q1.execute(), q2.execute()) {
    System.out.print("Reach EQ Branch");
} else {
    System.out.print("Reach NEQ Branch");
}
```

Conclusion 4. *The benefit of space refinement in concolic testing increases as input space size increases.*

In both examples, the concolic test engine successfully found input tables to cover both branches in the Java snippet. As shown in Figure 4.10, both examples indicate that the refined search space enables the concolic test engine to run faster in the testing process. In the first example (q_1, q_2), the provenance predicate $\phi(t, t_O) = (t.\mathsf{fid} = t_O.1 \wedge t.\mathsf{year} > 2010)$ restricts the choices of the grouping key fid to be the same across the table. In the second example, the provenance predicate determines that it is sufficient to make both fid and year to be the same. Thus, the concolic test engine benefits from the smaller size of the refined search space to generate inputs faster.

4.7 Related Work

SQL Equivalence. SQL query equivalence is a problem that has been extensively study in the database theory community [149, 39, 40, 28]. In general, query equivalence is undecidable, and subsequent study aims to identify decidable subsets of SQL and build decision

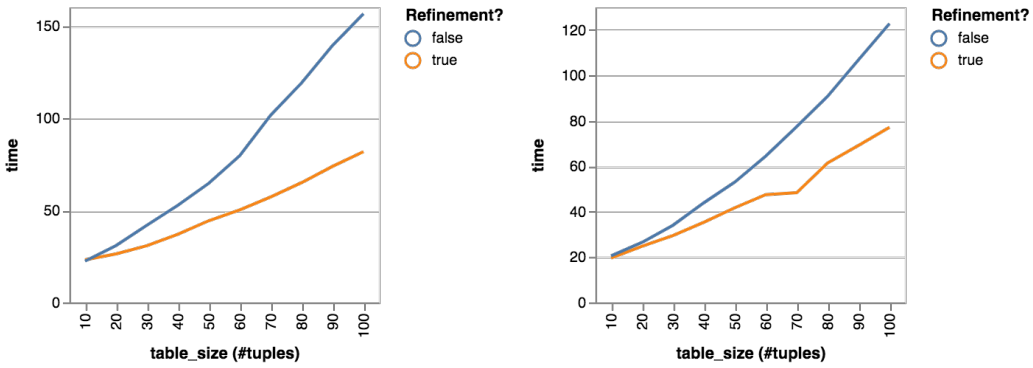


Figure 4.10: Results for checking query equivalence between q_1, q_2 (left) and q_3, q_4 (right).

procedures for them. Known decidable SQL subsets include conjunctive queries (CQ) [28], CQ with union (UCQ) [149], CQ with linear arithmetic [39], and CQ with *one* aggregate in the outer-most layer [40].

Tools for reasoning about query equivalence include HottSQL [35], a proof environment built on top of Coq for interactive SQL proof that targets a different verification method, and Cosette [34], an SMT-based verifier for bounded verification. As shown in our case study, our space refinement algorithm can be used to improve symbolic reasoning of SQL equivalence.

[156] describes a type inference algorithm that maps datalog queries into type programs formulated in (decidable) monadic datalog to check Datalog query containment. These types are abstractions of the program semantics, and they can be used to optimize query execution as well as query containment checking. In particular, containment checking within the type language is sound but incomplete. Their abstraction is a sound approximation of all possible input tables. We also map programs (i.e., queries) into constraints for equivalence checking but with different goals: we are interested in producing symbolic constraints that can be used to generate concrete test cases or counterexamples in addition to proving query equivalence. As a result, our approach uses the provenance of query outputs to speed up symbolic reasoning from existing solvers, rather than approximating programs semantics.

Test Data Generation. Tools for generating test data for SQL queries include Qex [189, 188], XData [74] and Tesma [179]. Qex is a SMT-based tool for generating unit tests from a given query and test assertion, where the goal is to construct unit tests from the data, query and assertion. XData is a mutation testing tool for SQL: given an input query, XData generates mutations of the query and then asks the underlying SMT solver to construct a distinguishing input to kill the mutant. It then adds the generated distinguishing input to the test suite for

database testing or grading [13]. Tesma is a concolic test engine that allows testing database applications under the concolic test framework. As shown in our evaluation, our space refinement algorithm can be applied to speedup test data generation.

Symmetry Breaking. Both Qex and Cosette include symmetry breaking modules for compiling into SMT formulas, similar to traditional symmetry breaking techniques [42, 49] used in constraint solving. Our approach instead utilizes high-level program semantics and breaks symmetry by identifying table equivalence given the semantics of the input queries. Also, our approach is specific to queries but generalizable to underlying encoding methods, which allows the opportunity to combine it with other lower-level symmetry breaking techniques [181].

Provenance Analysis. Provenance analysis has been studied in both scientific computation [17] and the database community [43, 21, 22]. Data provenance has been applied to incremental view updates and data filtering [68] to improve database performance. Our symbolic provenance analysis resembles these approaches, but our algorithm generates predicates to describe the provenance relation for *symbolic output tuples*, and we extend provenance reasoning to multiple queries simultaneously to solve the query equivalence problem. Our symbolic provenance analysis is a backward abstract analysis that feeds its result to improve the forward concrete analysis (e.g., compiling queries to SMT formulas, generating test cases). Previous work used backward analysis to summarize information that is not readily available to forward analysis, making the latter more efficient [81, 54, 25].

Semantics Abstraction. Semantics abstraction [61, 192, 139, 58] is also used in program synthesis to speed up (program) search space traversal, where program synthesizers compute space refinement constraints using abstract semantics of the target language to perform search space pruning. In program synthesis, such space refinement constraints are computed from (1) user specifications of the target program (logic formulas [139] or input-output examples [192, 58, 61]) and (2) currently synthesized partial programs. The refinement constraints capture properties of partial programs and allow the synthesizer to partition and prune the search space before reaching complete programs. In symbolic reasoning tasks, provenance predicates are computed from the verification condition and the target programs, which are then used to break semantic symmetry in the (table) search space. While different, studying the relationship between the two types of refinement constraints in these scenarios offers an interesting future work.

Verification of Database Applications. Mediator [201] is a tool for reasoning about database applications with updates. The SparkLite verifier [69] is an SMT-based tool for checking MapReduce program equivalence. Both approaches check program equivalences by in-

ferring program invariants. Our space refinement algorithm currently can only speedup the type of assertions defined in Section 4.2 but not general invariants. Generalizing our symmetry breaking algorithm to richer assertions is an interesting future work.

4.8 Summary

In this chapter, we introduced a space refinement algorithm for symbolic SQL reasoning. At the algorithm's core are: a symbolic provenance analysis module that analyzes which tuples in the input table contribute to the multiplicities of the target tuples in query outputs, and a space refinement module that refines the search space with the provenance information. Our experiments on bounded SQL verification, test data generation and concolic testing show that the refined search space effectively speeds up the symbolic reasoning process.

Discussion: Search Space Encoding

We discuss how Qex and Cosette represent the search space and how we encode the refined search space in these tools.

Representation of $\mathcal{S}$. Qex encodes the search space based on the exact number of tuples allowed in the table, and the search space of all tables containing k tuples is encoded as a list of tuples where each tuple is a list of symbolic values. For example, given the schema Bonus(Job:int, Dept:int, Sal:int), the space of all tables with 3 tuples is encoded as the symbolic table $\texttt{Bonus}_Q$ in Figure 4.11, where each value s_{ij} in the table is a symbolic integer. To encode the search space consisting of all tables with at most k tuples, Qex would iteratively increase the number of tuples from 0 to k and check the verification condition separately. Since the order of tuples in a table does not matter, Qex adopts a set of encoding constraints to break the encoding symmetry by asserting a canonical order of the tuples in the table. These constraints avoid the solver to encode the same table multiple times with a different order of tuples in the content.

Cosette differs from Qex by explicitly encoding the multiplicities of the tuples in the symbolic table, and a symbolic table with k entries in Cosette represents the search space of all tables with at most k distinct tuples. The symbolic table $\texttt{Bonus}_C$ in Figure 4.11 shows how Cosette encodes all tables with the schema `Bonus` containing at most 3 different tuples, and Cosette adopts a similar encoding constraints as Qex to reduce encoding symmetry. Compared to Qex, Cosette's encoding approach has the benefit of being able to compress the encoding for tables containing multiple identical tuples, e.g., a table with 100 identical tuples is represented with only one symbolic tuple with multiplicity 100. On the other hand, the use of multiplicity also makes representing tables without repeating tuples less compressed.

Bonus$_Q$

job	dept	sal
s_{11}	s_{12}	s_{13}
s_{21}	s_{22}	s_{23}
s_{31}	s_{32}	s_{33}

encoding constraints:
$(s_{21}, s_{22}, s_{23}) \geq (s_{11}, s_{12}, s_{13})$
$\wedge\, (s_{31}, s_{32}, s_{33}) \geq (s_{21}, s_{22}, s_{23})$

Bonus$_C$

job	dept	sal	mult
s_{11}	s_{12}	s_{13}	m_1
s_{21}	s_{22}	s_{23}	m_2
s_{31}	s_{32}	s_{33}	m_3

encoding constraints:
$(m_1 \geq 0) \ \wedge\ (m_2 \geq 0) \ \wedge\ (m_3 \geq 0)$
$\wedge\, (s_{21}, s_{22}, s_{23}) \geq (s_{11}, s_{12}, s_{13})$
$\wedge\, (s_{31}, s_{32}, s_{33}) \geq (s_{21}, s_{22}, s_{23})$

Figure 4.11: The encoding of the search space of all tables containing 3 tuples by Qex (Bonus$_Q$), and the encoding of the search space of all tables containing at most 3 tuples in Cosette (Bonus$_C$). Both encodings adopt a set of encoding constraints to reduce the encoding symmetry.

Representation of $\mathcal{S}'$. The reduced search space generated from $\mathcal{S}$ is encoded similarly. We encode the refined search space $\mathcal{S}'$ by adding additional assertions to the encoding constraints.

We first introduce a new symbolic tuple $t_O = (a_1, \ldots, a_n)$, where n is the number of columns in the output of q_1 and q_2, and then assert the constraint $\phi(t, t_O)$ for each t_O that encodes $\mathcal{S}$.

For example, assume the provenance of a query whose output has two columns (denoted as $t_O.1$ and $t_O.2$) is $\phi(t, t_O) = (t.\texttt{Job} = t_O.1 \wedge t.\texttt{Sal} > 5)$ (for the Bonus table above). To encode the refined search space $\mathcal{S}'$ using ϕ, we first introduce two new symbolic values (a_1, a_2) to model t_O, and add the following assertion in addition to the encoding constraint:

$$(s_{11} = a_1 \wedge s_{12} > 5) \wedge\ (s_{21} = a_1 \wedge s_{22} > 5) \wedge\ (s_{31} = a_1 \wedge s_{32} > 5)$$

Alternatively, we can also simplify the encoding by eliminating redundant symbolic values, e.g., replacing all s_{i1} with a_1 in the example above.

Part III

PROGRAM SYNTHESIS WITH VALUE PRESERVING ABSTRACTION

Chapter 5

The Kopis Framework: Synthesizing Relational Programs with Value Preserving Abstraction

In this chapter, we summarize the value-preserving abstraction program synthesis framework, Kopis, for relational query synthesis. The Kopis framework is designed for fast prototyping efficient synthesizers that synthesize relational queries from examples, but it could also support synthesizing programs that manipulate complex data structures in other domains.

In the relational query synthesis task, given input tables $\bar{T}_{in}$, and a partial output table T_{out}, we want to synthesize a set of programs S_p such that $\forall p \in S_p . T_{out} \subseteq [\![p(\bar{T}_{in})]\!]$.

Comparing to synthesis problems in other languages like assembly code [138], string manipulation programs [71], web extraction programs [26] or list programs [211], the relational query synthesis problem has the following unique challenges:

- *(Highly Parametric)* While a practical relational query often consists of only a small number of operators, every operator is highly parametric and a relational query can be instantiated in an exponential number of ways (the parameter space is exponential to input table column numbers). This indicates that the search tree often has smaller depth but with larger width, and the synthesizer needs to explore a smaller number of layers but a large number of candidates at each layer. This differentiates relational queries from assembly programs [138] that are often longer but less parametric.

- *(High Evaluation Cost)* Evaluating a relational query is much more costly than evaluating a string program or an assembly program, as the evaluation costs of many operators (e.g., `join`, `pivot`) are polynomial to input table size. Thus, a practical synthesizer should limit the number of invocations to the program evaluators for efficiency.

- *(High Memoization Overhead)* Value memoization is a technique that represents explored search space using behaviors of the programs (output of the programs) [186, 138]. This way, programs that behave equivalently on the given input example are identified, and only one representative program is used in further program search to avoid redundant search (since these programs are indistinguishable with the user example), which speeds up the synthesis process. However, because tables are com-

pound data with low compression rate (i.e., there are less programs with the same behavior compared to other domains), its cost overshadows its benefit.

- *(Non-decomposable)* Unlike string or web extraction programs that are often backwardly decomposable [140, 102, 71], relational queries can not be easily tailored to be both expressive for practical tasks and decomposable for efficient synthesis. This makes version space algebra-based approaches prohibitively expensive, as there exist exponentially many programs that can be inferred from the output example.

The Kopis framework is designed to address these issues. The key insight in the framework is to design *value-preserving abstractions* of the relational query language to allow the synthesizer to reason about realizability of the synthesis objective given a partially instantiated program. Such abstractions allow the synthesizer to prune families of infeasible programs (a collection of programs with the same structure but different parameters) at a time to dramatically reduce the search space size.

In the following sections, we formalize the relational query language and the synthesis problem and describe the value-preserving abstraction synthesis framework to solve it.

5.1 The Synthesis Task

In this section, we formally describe table and the relational query language $\mathcal{L}_Q$, and formulates the synthesis problem.

Notations. In the following, we use p to refer to programs in $\mathcal{L}_Q$, T and t to refer to tables, and c for table columns. We use the bar notation $\bar{x}$ to refer to a list of x (e.g., we use $\bar{T}$ for a list of tables $T_1, \ldots, T_n$ and $\bar{c}$ for a list of columns $c_1, \ldots, c_n$), and $\bar{f}(\bar{x})$ for a list of function applications $[f_1(x_1), \ldots, f_n(x_n)]$.

5.1.1 Relational Table

A table is formed by schema and content (Figure 5.1). The schema of a table is a list of strings representing column names, and the contents are a list of rows. Values in each row are scalars of types string, int, float, or datetime or null. Here, as we adopt *bag-semantics* [130] for the relational query language $\mathcal{L}_Q$, duplicate rows are allowed in tables. We define the following set of primitive table operations:

- *(Table containment).* $T_1 \subseteq T_2$, if all rows in T_1 present in T_2, and the multiplicity of each row in T_1 is smaller than or equal to its multiplicity in T_2.
- *(Table equivalent).* $T_1 = T_2$, if they mutually contain each other ($T_1 \subseteq T_2$ and $T_2 \subseteq T_1$).
- *(Table schema reference).* $\mathsf{schema}(T)$, returns the schema of the table.

- *(Table column reference).* $T[\bar{c}]$, returns a subtable of T that contains only columns $\bar{c}$. Special case: $T[c]$ returns a table with one column, we use it interchangeably as a list that contains only values from column c of T. We use $T[-\bar{c}]$ to refer to selecting all columns not included in $\bar{c}$.
- *(Table inclusion).* $T_1 \overset{\diamond}{\subseteq} T_2$, if there exists $\bar{c} \subseteq \mathsf{schema}(T_2)$ such that $T_1 \subseteq T_2[\bar{c}]$.
- *(Table union).* $T_1 \cup T_2$, union of contents in T_1 and T_2 (when their schema type are compatible); the schema of $T_1 \cup T_2$ is the same as T_1.

$$
\begin{array}{rcll}
T & ::= & \mathsf{Table}(schema, content) & \text{(Table)} \\
schema & ::= & [c_1, \ldots, c_m] & \text{(Schema)} \\
content & ::= & [r_1, \ldots, r_n] & \text{(Content)} \\
r & ::= & [v_1, \ldots, v_m] & \text{(Row)}
\end{array}
$$

Figure 5.1: The definition of relational tables, where c refers to column names and v refers cell values. Cell values are scalars of types string, int, float, and datetime.

5.1.2 The Language of Relational Queries $\mathcal{L}_Q$

Figure 5.2 shows the grammar of $\mathcal{L}_Q$. $\mathcal{L}_Q$ contains both operators from standard SQL [192], analytical SQL, and tidyverse [194] and supports both database querying and data transformation tasks. We choose this highly expressive $\mathcal{L}_Q$ as the synthesis target language for the purpose of illustrating how the general value-preserving abstraction synthesis framework works. In practice, only a subset of $\mathcal{L}_Q$ is needed for problems in a particular domain; $\mathcal{L}_Q$ can also be extended when necessary.

Programs p in $\mathcal{L}_Q$ are constructed from the following set of operators:

- Relational algebra: projection, dedup (select distinct), filter (selection), join and union;
- Extended relational algebra: aggregate (group and aggregate), left_join (left outer join);
- Basic arithmetic and string manipulation: separate (split a column into two via string splitting), unite (combine two columns via string concatenation), arithmetic (column-wise arithmetic computation);
- Pivot operators: pivot_longer (pivot wide to long), pivot_wider (pivot long to wide).

$$
\begin{aligned}
p &\leftarrow T \mid \mathsf{projection}(p, \bar{c}) \mid \mathsf{dedup}(p) \mid \mathsf{filter}(p, pred) \\
&\mid \mathsf{join}(p_1, p_2, pred) \mid \mathsf{union}(p_1, p_2) \\
&\mid \mathsf{aggregate}(p, \bar{c}, \alpha, c_t) \mid \mathsf{left\_join}(p_1, p_2, \bar{c}_1 = \bar{c}_2) \\
&\mid \mathsf{separate}(p, c) \mid \mathsf{unite}(p, c_1, c_2) \mid \mathsf{arithmetic}(p, f) \\
&\mid \mathsf{pivot\_longer}(p, \bar{c}) \mid \mathsf{pivot\_wider}(p, c_{key}, c_{val}) \\
pred &\leftarrow \mathsf{true} \mid v\ binop\ v \mid \mathsf{is\_null}\ c \mid pred\ \mathsf{and}\ pred \mid pred\ \mathsf{or}\ pred \mid \mathsf{not}\ pred \\
v &\leftarrow c \mid const \mid \mathsf{null} \\
\alpha &\leftarrow \mathsf{max} \mid \mathsf{min} \mid \mathsf{average} \mid \mathsf{count} \mid \mathsf{sum} \mid \mathsf{count\_distinct} \mid \mathsf{concat} \\
op &\leftarrow = \mid > \mid < \mid \leq \mid \geq \mid \neq
\end{aligned}
$$

Figure 5.2: The definition of $\mathcal{L}_Q$, where T refers to tables, c refers to column names, $const$ refers to constant values.

5.1.3 The Synthesis Problem

Definition 4 formulates the example-based relational query synthesis problem. Compared to traditional synthesis tasks where the goal is to synthesize a program p from input example I and output example O such that $[\![p(I)]\!] = O$, our synthesis task expects the user to provide only a *partial output table* (a subset of the actual output O). The rationale behind the design is to reduce the user effort when working with large input data: without this, the user either needs to provide the full output (which requires considerable effort due to its large size) or craft a small representative input data to reduce the output size (which could make the synthesis task more ambiguous).

Definition 4. (The Synthesis Task) Given input tables $\bar{T}_{in}$, a partial output example T_{out} and the number of desired programs N, the synthesis objective is to find a set of N programs S_p from $\mathcal{L}_Q$ such that each $p \in S_p$ satisfies the property $T_{out} \subseteq [\![p(\bar{T}_{in})]\!]$.

5.2 The Synthesis Algorithm

Algorithm 8 shows the top level synthesis algorithm: an abstraction-based enumerative search algorithm that iteratively enumerates and prunes abstract programs until specification-consistent programs are found (an abstract program p is a program with unfilled parameters, unfilled parameters are represented as holes "□"). The Synthesize procedure contains two main components: (1) a coarse search component that enumerates program skeletons – pro-

Algorithm 8 Top-level synthesis algorithm.

1: **procedure** SYNTHESIZE($\bar{T}_{in}, T_{out}, depth, N$)
2: **input:** input tables $\bar{T}_{in}$, partial output example T_{out}, search depth $depth$, the number of desired programs N.
3: **output:** a set of programs that are consistent with $\bar{T}_{in}$ and T_{out}
4: $W_0 \leftarrow \{T_{in_1}, \ldots, T_{in_n}\}$;
5: $k \leftarrow 0$;
6: **while** $k < depth$ **do**
7: **for** $\gamma \in \left\{ \begin{array}{l} \mathsf{projection}, \mathsf{dedup}, \mathsf{filter}, \mathsf{aggregate}, \mathsf{separate}, \\ \mathsf{unite}, \mathsf{arithmetic}, \mathsf{pivot\_longer}, \mathsf{pivot\_wider} \end{array} \right\}$ **do**
8: $W_k \leftarrow \{\gamma(p, \bar{\square}) \mid p \in W_{k-1}\}$;
9: **for** $\gamma \in \{\mathsf{join}, \mathsf{left\_join}, \mathsf{union}\}$ **do**
10: $W_k \leftarrow W_k \cup \{\gamma(p_1, p_2, \square) \mid p_1 \in W_i, p_2 \in W_j, i + j = k - 1\}$;
11: $W \leftarrow \bigcup_{i \in [0, depth]} W_i$;
12: $R \leftarrow \emptyset$;
13: **while** $\neg W.\mathsf{isEmpty}()$ **do**
14: $p \leftarrow W.\mathsf{next}()$;
15: **if** $\mathsf{isConcrete}(p)$ **then**
16: **if** $T_{out} \subseteq [\![p(\bar{T}_{in})]\!]$ **then**
17: $R \leftarrow R \cup \{p\}$;
18: **if** $|R| \geq N$ **then return** R
19: **else continue**;
20: $\phi \leftarrow \mathsf{abstractReasoning}(p, \bar{T}_{in}, T_{out})$;
21: **if** $\mathsf{UNSAT}(\phi)$ **then continue**;
22: $\square_i \leftarrow \mathsf{chooseNextHole}(p)$;
23: $W \leftarrow W \cup \left\{ [\square_i \mapsto v]p \;\middle|\; v \in \mathsf{inferDomain}(\square_i, p, \bar{T}) \right\}$
return R

grams with no instantiated parameters, and (2) a fine search phase that fills parameters of feasible program skeletons until programs consistent with the user specification are found.

In the coarse search phase, starting from input tables $\bar{T}_{in}$, and at each iteration k, the algorithm constructs program skeletons of size k by applying unary constructors on skeletons of size $k-1$ (line 7-8) or applying binary constructors on skeletons whose sizes sum to $k-1$ (line 9-10). This process generates the set of all program skeletons W up to size $depth$ (an algorithm parameter).

In the fine search phase, the algorithm takes one abstract program p out of the work list W at each time. If the program p is already concrete (i.e., no hole left to be instantiated), the algorithm checks if it is consistent with the output example T_{out} and adds it to the result

set R if so (line 15-18). Otherwise, p is an abstract program, and the algorithm conducts an abstract analysis to check the whether the current abstract program p can be instantiated to satisfy the output example T_{out} (we call this a *realizability check*). If the realizability check fails, the abstract program will be pruned (as no instantiation of it could satisfy T_{out}) (line 20-21). If the check succeeds, the synthesizer expands the search space by (1) choosing the next hole $\square_i$ in p to be instantiated, (2) inferring the domain of the hole $\square_i$, and (3) replacing the hole $\square_i$ with each candidate value v and add them to the work list for further search (lines 22, 23).

The synthesis framework has three submodules that can be customized to make the synthesis algorithm more efficient in certain domains:

- W.next(): this function chooses the next sketch from the worklist W to instantiate, and its choice decides the search strategy. Scythe [192] and Falx [195] adopt a breadth-first search strategy to visit programs with smaller sizes first before examining more complex ones, and the order in which operators are decided with a predefined heuristics (e.g., Falx explores programs with pivot_longer before pivot_wider as they are more commonly used in data processing for visualizations).

- chooseNextHole(p): this function decides the next hole in the abstract program p to expand, and a good choice of the function allows the synthesizer to prune infeasible programs earlier. Scythe and Falx adopt the breadth-first-search strategy by always choosing holes from operators closer to AST leaf nodes, as this makes partial evaluation possible and makes the reasoning more precise.

- abstractReasoning(p): this function analyzes whether an abstract program p can realize the synthesis objective based on abstract semantics of the language. Prior work like Morpheus [58] adopts high-level table type information (e.g., column, row number) to decide realizability. But Scythe and Falx adopt value-preservation abstraction for the abstract reasoning, as it allows more precise reasoning and can dramatically prune the search space. We will describe the details in the following sections.

In general, the Synthesize procedure provides a sound and complete way to explore the program space within the given depth. but its practical performance relies on whether the abstract reasoning subroutine abstractReasoning can effectively prune infeasible abstract programs early to avoid search explosion. In the next section, we formally describe the design of *value-preserving abstraction* that reasoning about realizability of the synthesis objective given an abstract program p.

5.3 Value-preserving Abstraction for Program Reasoning

Given an abstract program p, the abstract reasoning process aims to decide (as precisely as possible but without affording excessive computation time) if p can be instantiated into a concrete program p' (i.e., $p' \in \mathsf{instantiate}(p)$, where $\mathsf{instantiate}(p)$ denotes the set of all programs that can be instantiated from p by substituting holes in p with concrete parameters) such that $T_{out} \subseteq [\![p'(\bar{T}_{in})]\!]$.

Definition 5. (The Analysis Process) Given input tables $\bar{T}_{in}$, a partial output example T_{out} and an abstract program p of form $p = p_2(p_1(\dots), \dots)$ (p_1 is a prefix of p, and p_2 the a suffix of p given p_1; both of them can be abstract), the analyzer computes:

- ϕ_1^+, which over-approximation of the outputs that can be derived from programs instantiated from p_1. ϕ^+ satisfies: $\forall p_1' \in \mathsf{instantiate}(p_1).[t \mapsto [\![p_1'(\bar{T}_{in})]\!]]\phi_1^+$.
- ϕ_2^-, which is a necessary condition the input to p_2 needs to satisfy if there exists an instantiation of p_2 that can return T_{out}. ϕ_2^- satisfies the following property: $\forall p_2' \in \mathsf{instantiate}(p_2).\forall T.(T_{out} \subseteq [\![p_2'(T)]\!]) \implies [t \mapsto T]\phi_2^-)$.

In this way, the analyzer checks whether the whole program p can realize the synthesis objective by checking if $\mathsf{UNSAT}(\phi_1^+ \wedge \phi_2^-)$, and prunes p if the check passes.

In this analysis process, there are two special cases of how p can be decomposed into $p = p_2(p_1(\dots), \dots)$:

- $p = p_1(\dots)$. In this case, p_2 is the identity function, and we only need to derive ϕ_1^+ and check $\mathsf{UNSAT}([t \mapsto T_{out}]\phi_1^+)$ to decide whether to prune p. This is an approach adopted in Scythe [192].
- $p = p_2(\dots)$. In this case, $p_1 = \bar{T}_{in}$, and we only need to derive ϕ_2^- and check $\mathsf{UNSAT}([t_k \mapsto T_{in_k}]\phi_2^-)$ to decide whether p can be pruned or not.

In general, the first analysis in Definition 5 is a *forward analysis* that calculates an upper bound of output of p_1 given input $\bar{T}_{in}$ and the second analysis is a *backward analysis* that calculates a lower bound of the input to p_2 given the output T_{out}. In practice, given a program p, instead of only trying one decomposition of p into prefix and suffix, Kopis decomposes p in all possible ways to exploit the pruning opportunity (the total number of decomposition is linear to the program size, which is tractable).

In the following, we introduce the abstract semantics to derive ϕ^+ and ϕ^- from input and output examples.

Forward analysis: $p \Downarrow \phi^+$

$$\frac{}{T \Downarrow t \subseteq T} \qquad \frac{p \Downarrow t \subseteq^* T}{\mathsf{projection}(p, \_) \Downarrow t \subseteq^* T} \qquad \frac{p \Downarrow t \subseteq^* T \quad T' = [\![\mathsf{dedup}(T)]\!]}{\mathsf{dedup}(p) \Downarrow t \subseteq^* T'}$$

$$\frac{p_1 \Downarrow t \subseteq^* T_1 \quad p_2 \Downarrow t \subseteq^* T_2}{\mathsf{join}(p_1, p_2, \_) \Downarrow t \subseteq^* T_1 \times T_2} \qquad \frac{p_1 \Downarrow t \subseteq T_1 \quad p_2 \Downarrow t \subseteq T_2}{\mathsf{union}(p_1, p_2) \Downarrow t \subseteq^* T_1 \cup T_2}$$

$$\frac{p \Downarrow t \subseteq^* T}{\mathsf{filter}(p, \_) \Downarrow t \subseteq^* T} \qquad \frac{p \Downarrow t \subseteq^* T \quad T' = [\![\mathsf{arithmetic}(T, f)]\!]}{\mathsf{arithmetic}(p, f) \Downarrow t \subseteq^* T'}$$

$$\frac{p_1 \Downarrow t \subseteq^* T_1 \quad p_2 \Downarrow t \subseteq^* T_2 \quad T_3 = [\![\mathsf{left\_join}(T_1, \mathsf{filter}(T_2, \mathsf{false}))]\!]}{\mathsf{left\_join}(p_1, p_2, \_) \Downarrow t \subseteq^* (T_1 \times T_2) \cup T_3}$$

$$\frac{p \Downarrow t \subseteq^* T \quad T' = [\![\mathsf{separate}(T, c)]\!]}{\mathsf{separate}(p, c) \Downarrow t \subseteq^* T'} \qquad \frac{p \Downarrow t \subseteq^* T \quad T' = [\![\mathsf{unite}(T, c_1, c_2)]\!]}{\mathsf{unite}(p, c_1, c_2) \Downarrow t \subseteq^* T'}$$

$$\frac{p \Downarrow t \subseteq^* T \quad T' = [\![\mathsf{pivot\_longer}(T, \bar{c})]\!]}{\mathsf{pivot\_longer}(p, \bar{c}) \Downarrow t \subseteq^* T'} \qquad \frac{p \Downarrow t \subseteq^* T \quad T' = [\![\mathsf{pivot\_wider}(T, c_{key}, c_{val})]\!]}{\mathsf{pivot\_wider}(p, c_{key}, c_{val}) \Downarrow t \subseteq^* T'}$$

$$\frac{\mathsf{isConcrete}(p)}{\gamma(p, \_) \Downarrow t \subseteq [\![\gamma(p, \_)]\!]}\ \text{(F-Strong)} \qquad \frac{}{\gamma(p, \_) \Downarrow \top}\ \text{(F-Weak)}$$

Figure 5.3: Forward analysis that computes a condition ϕ^+ that outputs of p needs to satisfy. Rule F-Strong is applied whenever possible (when p is concrete) to derive a tight overapproximation, and F-Weak is applied when non other rules is applicable. Notation "_" refers to either a hole "□" or a concrete parameter, and $\subseteq^*$ refers to either $\subseteq$ or $\subseteq^\diamond$.

5.3.1 Forward Analysis

Figure 5.3 shows the forward analysis process. Given a program p, we derive a constraint $\phi^+ = t \in T$ (if $\phi^+ \neq \top$), where t is a symbolic variable and T is a concrete table that represents the "upper bound" of the outputs of p. This derivation process is denoted as $p \Downarrow \phi^+$. As shown in Definition 5, ϕ^+ overapproximates the behavior of all possible outputs that can come out of p.

The analysis is recursively defined, for example, in the analysis of $\mathsf{filter}(p, \_)$, we first derive the overapproximation of p based on $p \Downarrow t \subseteq^* T$, and then overapproximates the output from filter: while we don't know anything about the actual filter predicate, we know that all of its output must be contained by T, which gives us the constraint $t \subseteq^* T$. The analysis

Backward analysis: $\langle T, p\rangle \Uparrow \phi^-$

$$\frac{}{\langle T, t\rangle \Uparrow T \overset{\diamond}{\subseteq} t} \qquad \frac{\langle T, p\rangle \Uparrow \phi^-}{\langle T, \mathsf{filter}(p, \_)\rangle \Uparrow \phi^-} \qquad \frac{\langle T, p\rangle \Uparrow \phi^-}{\langle t, \mathsf{projection}(p, \_)\rangle \Uparrow \phi^-}$$

$$\frac{\langle T, p\rangle \Uparrow \phi^-}{\langle t, \mathsf{dedup}(p, \_)\rangle \Uparrow \phi^-} \qquad \frac{\langle T[-c_i], p\rangle \Uparrow \phi_i^-}{\langle T, \mathsf{arithmetic}(p, f)\rangle \Uparrow \bigvee_{i\in[1,|\mathsf{schema}(T)|]} \phi_i^-}$$

$$\frac{\langle [\![\mathsf{separate}(T, c_i)]\!], p\rangle \Uparrow \phi_i^-}{\langle T, \mathsf{unite}(p, \_)\rangle \Uparrow \bigvee_{i\in[1,|\mathsf{schema}(T)|]} \phi_i^-} \qquad \frac{\langle [\![\mathsf{unite}(T, c_i, c_j)]\!], p\rangle \Uparrow \phi_{ij}^-}{\langle T, \mathsf{separate}(p, \_)\rangle \Uparrow \bigvee_{i,j\in[1,|\mathsf{schema}(T)|]} \phi_{ij}^-}$$

$$\frac{\langle [\![\mathsf{dedup}(T[-c_i, -c_j])]\!], p\rangle \Uparrow \phi_{ij}^-}{\langle T, \mathsf{pivot\_longer}(p, \_)\rangle \Uparrow \bigvee_{i,j\in[1,|\mathsf{schema}(T)|]} \phi_{ij}^-} \qquad \frac{\langle [\![\mathsf{pivot\_longer}(T, c_i, c_j)]\!], p\rangle \Uparrow \phi_{ij}^-}{\langle T, \mathsf{pivot\_wider}(p, \_)\rangle \Uparrow \bigvee_{i,j\in[1,|\mathsf{schema}(T)|]} \phi_{ij}^-}$$

$$\frac{\langle T[-c_i], p\rangle \Uparrow \phi_i^-}{\langle T, \mathsf{aggregate}(p, \_, \_, \_)\rangle \Uparrow \bigvee_{i\in[1,|\mathsf{schema}(T)|]} \phi_i^-} \qquad \frac{}{\langle T, \gamma(p, \_)\rangle \Uparrow \top}\ \text{(B-Weak)}$$

Figure 5.4: Backward analysis. Derives a condition ϕ^- that the inputs to p need to satisfy given output $\langle T, p\rangle$. B-Weak is applied when none of the other rules are applicable.

is similar for binary operators like join, union and left_join. For example, in join, the output constraint $t \subseteq^* T_1 \times T_2$ is an over-approximation of the output for all possible join predicates.

Note that there are two special rules in the forward analysis process: F-Weak and F-Strong. F-Strong is applied whenever possible to make the analysis output more accurate: i.e., whenever we encounter a fully instantiated sub program p, we use its output $[\![p]\!]$ as its over-approximation (this is a partial evaluation strategy). F-Weak is then used when non of the other analysis rule can be applied, which effectively "gives up" the analysis (as the output constraint $\phi^+ = \top$ means that ϕ^+ is always satisfiable). For example, given programs like $\mathsf{arithmetic}(p, \square)$ or $\mathsf{aggregate}(p, \bar{\square})$, while it is not impossible to compute an output table to summarize its output behavior, its computation would be exponentially expensive, and our analysis rule simply gives up forward analysis here. While it may looks pessimistic here, in many practical cases, the reasoning of such a program can be achieved based on the backward analysis process, and otherwise, we only need to continue enumerating a few more key parameters to enable forward analysis.

Backward Analysis

Figure 5.4 shows the backward analysis process. Given a program p and its output T, we derive a constraint ϕ^- of the format $\phi^+ = \bigvee_i T_i \overset{\diamond}{\subseteq} t$ (if $\phi^- \neq \top$), where t is a symbolic variable and T_i is a concrete table that represents the "lower bound" to inputs of p. This derivation process is denoted as $\langle T, p\rangle \Uparrow \phi^-$. As shown in Definition 5, ϕ^- is a necessary condition the input table to p needs to satisfy given that we want the output T to be contained by the output of p.

In the backward analysis process, the analysis rules for unite, separate, pivot_longer, pivot_wider would involve a lightweight enumeration process. The purpose here is to "guess" which columns in the output belong to the ones generated by the current operator, and continue propagating other parts of the table to its child program. For example, in the analysis of $\langle T, \mathsf{unite}(p, \_)\rangle$, the rule considers all possible c_i that might be the new column generated by unite, and then for each c_i, the analyzer continue analyzing $\langle [\![\mathsf{separate}(T, c_i)]\!], p\rangle$ that derive ϕ_i^- by considering that $[\![\mathsf{separate}(T, c_i)]\!]$ is the table generated by p. The analyzer finally combines all ϕ_i^- together and form the property $\bigvee_{i\in[1,|\mathsf{schema}(T)|]} \phi_i^-$ that the inputs to the program needs to satisfy. This process may look very expensive. However, several practical reasons make the analysis not just effective at pruning but also easy to compute: (1) the number of operators in a practical program is often small, thus the number of branches will not grow dramatically, and (2) the analyzer often don't need to explore all possible c_i because of type restriction (e.g., the target column for unite much be a string column that contains connector).

Similar to the forward analysis rule F-Weak, the rule B-Weak here is applied when non of the other rules is applicable, and derive a constraint $\top$ that the input can always satisfy. Thus the program cannot be pruned with this condition.

Bidirectional Abstract Analysis

As noted in (in Section 5.3) a program p will be sliced into $p = p_2(p_1(\dots), \dots)$ to explore all possible ways to prune p. Here, given the slicing, we compute $p_1(\dots) \Downarrow \phi_1^+$, and $\langle T_{out}, p_2(t, \_)\rangle \Uparrow \phi_2^-$. Then we check $\mathsf{UNSAT}(\phi_1^+ \wedge \phi_2^-)$ (here t is the only variable) to decide if p can be pruned — this pruning process only requires running "$\overset{\diamond}{\subseteq}$" checking up to k times, where k is the number of disjunctions in ϕ_2^-. Note that a slicing with a smaller sized p_2 is more likely to derive a stronger pruning condition ϕ_2^- and a weaker ϕ_1^+, and a slicing with bigger sized p_2 will have weaker ϕ_2^- and a stronger ϕ_1^+. Whether p can be pruned given a slicing p_1, p_2 depends on operators are in p. Here are some examples:

- If $p = \mathsf{pivot\_longer}(\mathsf{join}(\dots), \dots)$, p is more likely to be pruned with a slicing $p_1 = \mathsf{join}(\dots)$ and $p_2 = \mathsf{pivot\_longer}(\dots)$;

- If $p = \mathsf{pivot\_longer}(\mathsf{pivot\_wider}(\dots), \dots)$, p is more likely to be pruned with a slicing $p_1 = \mathsf{id}$ and $p_2 = \mathsf{pivot\_longer}(\mathsf{pivot\_wider}(\dots), \dots)$ unless all parameters in pivot_wider are instantiated (note that the forward analysis rules can only handle pivoting when the they are fully instantiated);

- If $p = \mathsf{join}(\mathsf{filter}(\dots), \dots)$, $p_1 = \mathsf{join}(\mathsf{filter}(\dots), \dots)$ and $p_2 = \mathsf{id}$ is ideal for pruning (note that no rule in backward analysis can effectively analyze join, but forward analysis rules for both join and filter are available);

- If $p = \mathsf{join}(\mathsf{pivot\_longer}(\dots), \dots)$, p is more unlikely to be pruned with any slicing unless p is relatively concrete. Such cases are in "the blind spot" of the pruning algorithm. In practice, in many cases, a semantically equivalent program with different syntactical structure exists and the synthesizer does not need to search for such programs. For example, $p = \mathsf{join}(\mathsf{unite}(\dots), \dots)$ is a challenging to analyze program, but an equivalent program $p' = \mathsf{unite}(\mathsf{join}(\dots), \dots)$ exists and thus p does not need to be explored by the synthesizer.

In general, value-preserving abstraction is a relatively expensive (compared to type-based analysis [58, 139]), but it provides much stronger pruning power.

The abstract analysis satisfies the following property that ensures pruning soundness:

Property 9. Given inputs $\bar{T}_{in}$ and partial output T_{out} and a program $p = p_2(p_1(\dots), \dots)$, let $p_1(\dots) \Downarrow \phi_1^+$ and $\langle T_{out}, p_2(t, \_)\rangle \Uparrow \phi_2^-$, if $\mathsf{UNSAT}(\phi_1^+ \wedge \phi_2^-)$, then there is no instantiation of p that would satisfy $T_{out} \subseteq [\![p(\bar{T}_{in})]\!]$.

This property can be derived from Definition 5, as ϕ_1^+ constrains the property of all possible outputs that p_1 can return (given input T_{in}), and ϕ_2^- constrains the property the input to p_2 needs to satisfy (given output T_{out}). If the constraint cannot be satisfied, the two programs p_1 and p_2 cannot work together to solve the synthesis task (Definition 4). Note that because the analysis is under abstract semantics, there is still no guarantee that p can be instantiated into a correct solution if $\phi_1^+ \wedge \phi_2^-$ can be satisfied. In such cases, the synthesizer will continue instantiating and check when more information is known, or until a fully instantiated program that satisfies the user specification is found (Algorithm 8).

5.4 Summary

The Kopis framework summarizes the key synthesis algorithm behind Falx and Scythe. The high level algorithm in Kopis is an enumerative search algorithm (where the search policy can be customized) that leverages abstract reasoning to prune infeasible abstract programs

(Section 5.2). The key here is to use value-preserving abstractions to enable more accurate reasoning of the program semantics and prune infeasible ones (Section 5.3). We envision KOPIS can be effectively applied to other domains where programs manipulate complex data structures (e.g., trees, sequence, graphs), where value-preserving abstraction can effectively leverage structure and value information in the data to enable effective pruning.

Conclusion

In this thesis, we build synthesis-powered data analysis tools to bridge the *programmability gap* in data analysis that inexperienced data scientists often struggle with advanced data analysis tasks due to lack of programming experience. Concretely, we build tools that let inexperienced users perform advanced data manipulation and data visualization tasks using examples, by synthesizing programs that generalize the examples.

We introduced Falx, a visualization-by-example tool that let the user create expressive visualizations using a demonstration of how a few data points from the dataset would be mapped to the canvas, and Scythe, a SQL query synthesizer that lets the user author advanced SQL queries using input-output examples.

- *(Interaction Model)* From the human computer interaction perspective, our system design transforms the traditional full specification task (specifying all steps needed to solve a data analysis problem) into a partial specification-recognition task (providing a partial task specification that demonstrate the task and then exploring synthesized programs to find the desired solution).

  In Falx, the user specifies the visualization by demonstrating how a few data points in the input data are mapped to the canvas. This lets users directly specify visualization on "untidy" data, and it reduces the users' efforts in learning and performing data transformation in visualization authoring. Falx then takes advantage of the "visual nature" of the domain and lets the user discover the desired visualization by exploring candidate solutions using a design exploration interface (Chapter 1).

  In Scythe, the user demonstrates a database querying task using input-output example table pairs, and Scythe finds queries that are consistent with the specified input-output behavior. Scythe then leverages a symbolic reasoning engine to compute distinguishing inputs that can disambiguate similar complex queries. This allows the user to distinguish queries based on their output behaviors on the distinguishing input as opposed to directly inspecting the queries (Chapter 4). These designs let the user specify a complex task easily and select the desired solution confidently.

- *(Scalability)* From the programming language research perspective, our key design is to leverage *value-preserving abstractions* to achieve early pruning of the search space (Chapter 5). Using value-preserving abstractions, the synthesizer analyzes behaviors

of abstract programs using over-approximations and decides if the user example is realizable. Though relatively expensive, the analysis accuracy can discover more pruning opportunities than other approaches, which is the key to let Scythe (Chapter 3) and Falx (Chapter 2) scale much better in the context of synthesizing expressive programs that manipulate relational tables.

We believe this thesis is the prelude of a new generation of data analysis tools. We envision the following future designs to make synthesis-powered analysis tools more prevalent and better at reducing data scientists' effort.

- *Abstraction Learning.* Good abstractions of language semantics are key to scaling up synthesis algorithms since they let synthesizers reason about realizability of user specifications in different parts of search space and prune infeasible ones. To build an efficient synthesizer for a new domain, we need to carefully design the semantic abstraction of the language that supports both effective pruning and compositional reasoning. This makes current practices of manual abstraction design insufficient.

  Thus, we envision the design of an abstraction learning approach to allow automatic discoveries of semantic abstractions that adapt to *different languages* and *different application contexts*. One possibility is to build *a reinforcement learning based abstraction learning framework* with the following components: (1) a symbolic compiler that can enumerate and propose sound abstractions for given operators, (2) a synthesis engine that assesses quality of abstractions by running synthesis task simulations, and (3) a continuous optimizer that optimizes and guides the discovery of new abstractions.

- *Multi-modal Program Synthesis.* In addition to better search algorithms, allowing users to provide richer task information can also greatly benefit the synthesizer performance. Richer information means that the synthesizer will be better at pruning infeasible search space and better at program disambiguation.

  Examples of such multi-modal interface includes: (1) an interface that lets users demonstrate computation processes (e.g., using example formulas) besides concrete values to solve analytical problems that often involve complex computations; and (2) an interface that combines natural specification (e.g., sketching, dialog systems) with logic specification (e.g., input-output examples, demonstrations, properties). These interfaces will allow the user to provide richer task information without much extra effort.

- *Interactive Synthesis Environment.* For more complex data analysis tasks, it is no longer ideal for the user to specify from scratch every time, because this increases the user's specification effort, synthesizer scalability, and disambiguation difficulty.

  We envision that future program synthesizers would work in the form of "program synthesis environments" similar to how data scientists use programming notebooks to solve complex analysis tasks. First, we envision the program synthesis environment to support *incremental synthesis* that let the user to develop complex analysis tasks on top of results synthesized from the previous iterations. Second, we envision the synthesis environment to support *opportunistic adaptation* that allows the user to copy, paste and adapt existing ad hoc code snippets (e.g., existing visualization snippets from online galleries) to solve *new* tasks. For example, the data scientist could demonstrate how they intend to adapt an existing visualization design to work on their new datasets by manipulating visual objects. The system will then propagate these changes to the program parameter space to adapt the visualization program. Finally, we envision the environments to have the ability to learn user preferences over time to make the synthesis algorithm search and disambiguate better. Learning signals can come from the user's acceptance and rejections of certain designs and the order in which the user explores synthesis outputs. In general, interactive synthesis environments would let data scientists and synthesizers work together to tackle more challenging tasks.

All of these possibilities suggest a promising future for empowering data analysis with synthesis-based techniques. We hope that the thesis provides one exemplar of adapting core techniques in synthesis into powerful interactive tools that empower human creativity.

Bibliography

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. In *Proceedings of the Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pages 67–83, 2016.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. pages 934–950. Springer, 2013.

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[4] Louie Andre. 20 best data visualization software solutions of 2019. https://financesonline.com/data-visualization/, 2019.

[5] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[6] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[7] Shaon Barman, Sarah Chasins, Rastislav Bodík, and Sumit Gulwani. Ringer: web automation by demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 748–764, 2016.

[8] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices*, 50(6):218–228, 2015.

[9] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[10] Lyn Bartram, Michael Correll, and Melanie Tory. Untidy data: The unreasonable effectiveness of tables. *arXiv preprint arXiv:2106.15005*, 2021.

[11] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo-Lado, and Yannis Velegrakis. Quest: a keyword search system for relational data based on semantic and machine learning techniques. *Proceedings of the VLDB Endowment*, 6(12):1222–1225, 2013.

[12] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, pages 59–6, 2010.

[13] Amol Bhangdiya, Bikash Chandra, Biplab Kar, Bharath Radhakrishnan, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. The xda-ta system for automated grading of sql query assignments. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1468–1471. IEEE, 2015.

[14] Pavol Bielik, Marc Fischer, and Martin Vechev. Robust relational layout synthesis from examples for android. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[15] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[16] Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013.

[17] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[18] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics*, 15(6):1121–1128, 2009.

[19] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2301–2309, 2011.

[20] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[21] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 539–550, 2006.

[22] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings.*, pages 316–330, 2001.

[23] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90, 1977.

[24] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. Data generation for testing and grading SQL queries. *VLDB J.*, 24(6):731–755, 2015.

[25] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 363–374, 2009.

[26] Sarah Chasins and Rastislav Bodík. Skip blocks: reusing execution history to accelerate web scripts. *Proc. ACM Program. Lang.*, 1(OOPSLA):51:1–51:28, 2017.

[27] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43, 1998.

[28] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 59–70, 1993.

[29] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 587–610. Springer, 2020.

[30] Alvin Cheung and Armando Solar-Lezama. Computer-assisted query formulation. *Foundations and Trends in Programming Languages*, 3(1):1–94, June 2016.

[31] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 135–145, 2011.

[32] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1732–1736, 2012.

[33] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, 2013.

[34] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[35] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 510–524, 2017.

[36] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 341–354. ACM, 2016.

[37] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[38] Sara Cohen. Equivalence of queries combining set and bag-set semantics. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pages 70–79, 2006.

[39] Sara Cohen. Equivalence of queries that are sensitive to multiplicities. *VLDB J.*, 18(3):765–785, 2009.

[40] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2):5, 2007.

[41] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[42] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996.*, pages 148–159, 1996.

[43] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[44] Hanjun Dai, Yujia Li, Chenglong Wang, Rishabh Singh, Po-Sen Huang, and Pushmeet Kohli. Learning transferable graph exploration. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 2514–2525, 2019.

[45] Jonathan Danaparamita and Wolfgang Gatterbauer. Queryviz: helping users understand sql queries and their patterns. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 558–561. ACM, 2011.

[46] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 57–68, 2002.

[47] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[48] Luc De Raedt. Inductive logic programming. In *Encyclopedia of Machine Learning*, pages 529–537. Springer, 2011.

[49] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *Automated Deduction - CADE-23 - 23rd Inter-*

national Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings, pages 222–236, 2011.

[50] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 990–998, 2017.

[51] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-sensitive dataflow analysis with iterative refinement. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pages 425–442, 2006.

[52] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[53] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics (TOG)*, 37(6):1–16, 2018.

[54] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 37–48, 1995.

[55] Purna Duggirala. *Chandoo.org Website*, 2019.

[56] E90E50. *E90E50 Website*, 2019.

[57] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435, 2018.

[58] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[59] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. Component-based synthesis for complex apis. pages 599–612. ACM, 2017.

[60] Alberto Ferrari and Marco Russo. *Introducing Microsoft Power BI*. Microsoft Press, 2016.

[61] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[62] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 802–815, New York, NY, USA, 2016. ACM.

[63] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[64] Richard A Ganski and Harry KT Wong. Optimization of nested sql queries revisited. In *ACM SIGMOD Record*, volume 16, pages 23–33. ACM, 1987.

[65] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. pages 175–188. Springer, 2004.

[66] Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. Look for the proof to find the program: Decorated-component-based program synthesis. pages 86–103. Springer, 2017.

[67] Malu AC Gatto. Making research useful: Current challenges and good practices in data visualisation. 2015.

[68] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 675–686, 2007.

[69] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of spark programs. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 282–300, 2017.

[70] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2010.

[71] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011.

[72] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. pages 62–73. ACM, 2011.

[73] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 803–814. ACM, 2014.

[74] Bhanu Pratap Gupta, Devang Vira, and S. Sudarshan. X-data: Generating test data for killing SQL mutants. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 876–879, 2010.

[75] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. pages 27–38. ACM, 2013.

[76] Pat Hanrahan. Vizql: a language for query, analysis and visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 721, 2006.

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

[78] Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Trans. Vis. Comput. Graph.*, 16(6):1149–1156, 2010.

[79] Brian Hempel and Ravi Chugh. Semi-automated SVG programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*, pages 379–390, 2016.

[80] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-sketch: Output-directed programming for SVG. In François Guimbretière, Michael Bernstein, and Katharina Reinecke, editors, *Proceedings of the 32nd Annual ACM Symposium on User Interface*

Software and Technology, UIST 2019, New Orleans, LA, USA, October 20-23, 2019, pages 281–292. ACM, 2019.

[81] Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*, pages 104–115, 1995.

[82] Kevin Hu, Snehalkumar'Neil'S Gaikwad, Madelon Hulsebos, Michiel A Bakker, Emanuel Zgraggen, César Hidalgo, Tim Kraska, Guoliang Li, Arvind Satyanarayan, and Çağatay Demiralp. Viznet: Towards a large-scale visualization learning and benchmarking repository. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[83] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95, 2007.

[84] Jeevana Priya Inala and Rishabh Singh. Webrelate: integrating web data with spreadsheets using examples. *PACMPL*, 2(POPL):2:1–2:28, 2018.

[85] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. The containment problem for REAL conjunctive queries with inequalities. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pages 80–89, 2006.

[86] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. Meminsight: platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 345–356, 2015.

[87] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. pages 215–224. ACM/IEEE, 2010.

[88] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 711–726, 2016.

[89] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank Van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011.

[90] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey 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.

[91] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. In *IEEE Visual Analytics Science & Technology (VAST)*, 2012.

[92] Stephen Kasica, Charles Berret, and Tamara Munzner. Table scraps: An actionable framework for multi-table data wrangling from an artifact study of computational journalism. *IEEE Trans. Vis. Comput. Graph.*, 27(2):957–966, 2021.

[93] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid moves between code and graphical work in computational notebooks. In Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller, editors, *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, pages 140–151. ACM, 2020.

[94] Meraj Ahmed Khan, Larry Xu, Arnab Nandi, and Joseph M. Hellerstein. Data tweening: Incremental visualization of data transforms. *Proc. VLDB Endow.*, 10(6):661–672, 2017.

[95] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. Hampi: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 105–116. ACM, 2009.

[96] Nam Wook Kim, Eston Schweickart, Zhicheng Liu, Mira Dontcheva, Wilmot Li, Jovan Popovic, and Hanspeter Pfister. Data-driven guides: Supporting expressive design for information graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):491–500, 2017.

[97] Younghoon Kim, Kanit Wongsuphasawat, Jessica Hullman, and Jeffrey Heer. Graphscape: A model for automated reasoning about visualization similarity and sequencing. In *ACM Human Factors in Computing Systems (CHI)*, 2017.

[98] Dileep Kini and Sumit Gulwani. Flashnormalize: Programming by examples for text normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 776–783. AAAI Press, 2015.

[99] Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Mag.*, 30(4):65–67, 2009.

[100] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[101] Tessa A. Lau, Pedro M. Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 527–534, 2000.

[102] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 542–553, 2014.

[103] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 70–80. ACM, 2016.

[104] Fei Li and Hosagrahar V Jagadish. NaLIR: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 709–712. ACM, 2014.

[105] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[106] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In François Guimbretière, Michael Bernstein, and Katharina Reinecke, editors, *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST 2019, New Orleans, LA, USA, October 20-23, 2019*, pages 577–589. ACM, 2019.

[107] Halden Lin, Dominik Moritz, and Jeffrey Heer. Dziban: Balancing agency & automation in visualization design via anchored recommendations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[108] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John T. Stasko. Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox, editors, *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, page 123. ACM, 2018.

[109] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 958–968, 2018.

[110] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 355–368, 2016.

[111] Edward Lu and Ras Bodik. Quicksilver: Automatic synthesis of relational queries. Master's thesis, EECS Department, University of California, Berkeley, May 2013.

[112] Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. Show me: Automatic presentation for visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1137–1144, 2007.

[113] Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing quotient lenses. *PACMPL*, 2(ICFP):80:1–80:29, 2018.

[114] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. pages 48–61. ACM, 2005.

[115] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[116] Joao Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[117] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. https://github.com/fredfeng/Trinity/, 2019.

[118] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. Bidirectional evaluation with direct manipulation. *PACMPL*, 2(OOPSLA):127:1–127:28, 2018.

[119] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In Celine Latulipe, Bjoern Hartmann, and Tovi Grossman, editors, *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, pages 291–301. ACM, 2015.

[120] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[121] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .net framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706, 2006.

[122] Gonzalo Gabriel Méndez, Miguel A. Nacenta, and Sebastien Vandenheste. ivolver: Interactive visual language for visualization extraction and reconstruction. In Jofish Kaye, Allison Druin, Cliff Lampe, Dan Morris, and Juan Pablo Hourcade, editors, *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, pages 4073–4085. ACM, 2016.

[123] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proc. International Conference on Machine Learning*, pages 187–195. Proceedings of Machine Learning Research, 2013.

[124] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *PACMPL*, 2(POPL):1:1–1:30, 2018.

[125] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Trans. Vis. Comput. Graph.*, 25(1):438–448, 2019.

[126] Stephen Muggleton, Ramon Otero, and Alireza Tamaddoni-Nezhad. *Inductive Logic Programming*, volume 38. Springer, 1992.

[127] Norman Murray, Norman Paton, and Carole Goble. Kaleidoquery: a visual query language for object databases. In *Proceedings of the working conference on Advanced visual interfaces*, pages 247–257. ACM, 1998.

[128] Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. OpenReview, 2017.

[129] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. OpenReview, 2016.

[130] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.

[131] Norman H Nie, Dale H Bent, and C Hadlai Hull. *SPSS: Statistical package for the social sciences*, volume 227. McGraw-Hill New York, 1975.

[132] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 619–630, New York, NY, USA, 2015. ACM, ACM.

[133] Carlos Pacheco et al. *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, 2009.

[134] Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1114–1124, 2018.

[135] Jon Peltier. *Peltiertech Website*, 2019.

[136] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Notices*, volume 49, pages 396–407. ACM, 2014.

[137] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Greenthumb: Superoptimizer construction framework. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 261–262. ACM, 2016.

[138] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 297–310, 2016.

[139] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538, 2016.

[140] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems,*

Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, pages 107–126. ACM, 2015.

[141] Vijayshankar Raman and Joseph M Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

[142] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. pages 761–774. ACM, 2016.

[143] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *POPL*, volume 50, pages 111–124. ACM, 2015.

[144] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. volume 49, pages 419–428. ACM, 2014.

[145] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 792–800, 2015.

[146] Donghao Ren, Bongshin Lee, and Matthew Brehmer. Charticulator: Interactive construction of bespoke chart layouts. *IEEE Trans. Vis. Comput. Graph.*, 25(1):789–799, 2019.

[147] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.

[148] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. Codemend: Assisting interactive programming with bimodal embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 247–258, 2016.

[149] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[150] Bahador Saket, Lei Jiang, Charles Perin, and Alex Endert. Liger: Combining interaction paradigms for visual analysis. *CoRR*, abs/1907.08345, 2019.

[151] Bahador Saket, Hannah Kim, Eli T. Brown, and Alex Endert. Visualization by demonstration: An interaction paradigm for visual data exploration. *IEEE Trans. Vis. Comput. Graph.*, 23(1):331–340, 2017.

[152] Arvind Satyanarayan and Jeffrey Heer. Lyra: An interactive visualization design environment. *Comput. Graph. Forum*, 33(3):351–360, 2014.

[153] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John T. Stasko, John Thompson, Matthew Brehmer, and Zhicheng Liu. Critical reflections on visualization authoring systems. *IEEE Trans. Vis. Comput. Graph.*, 26(1):461–471, 2020.

[154] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):341–350, 2017.

[155] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Vis. Comput. Graph.*, 22(1):659–668, 2016.

[156] Max Schäfer and Oege de Moor. Type inference for datalog with complex type hierarchies. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 145–156, 2010.

[157] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGPLAN Notices*, 48(4):305–316, 2013.

[158] David Schroeder and Daniel F. Keefe. Visualization-by-sketching: An artist's interface for creating multivariate time-varying data visualizations. *IEEE Trans. Vis. Comput. Graph.*, 22(1):877–885, 2016.

[159] Arijit Sengupta and Andrew Dillon. Query by templates: A generalized approach for visual query formulation for text dominated databases. In *Digital Libraries, 1997. ADL'97. Proceedings., IEEE International Forum on Research and Technology Advances in*, pages 36–47. IEEE, 1997.

[160] Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. Generating test data for killing SQL mutants: A constraint-based approach. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1175–1186, 2011.

[161] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 515–527, 2018.

[162] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

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

[164] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*, pages 634–651. Springer, 2012.

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

[166] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. *ACM SIGPLAN Notices*, 51(1):343–356, 2016.

[167] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–340. ACM, 2016.

[168] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. pages 326–340. ACM, 2016.

[169] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[170] Armando Solar-Lezama. The sketching approach to program synthesis. In *Proc. Asian Symposium on Programming Languages and Systems*, pages 4–13. Springer, 2009.

[171] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. pages 281–294. ACM, 2005.

[172] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. pages 404–415. ACM, 2006.

[173] Andreas Stolcke et al. Srilm-an extensible language modeling toolkit. In *Interspeech*, volume 2002, 2002.

[174] Chris Stolte, Diane Tang, and Pat Hanrahan. Query, analysis, and visualization of hierarchically structured data using polaris. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 112–122. ACM, 2002.

[175] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.

[176] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[177] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[178] Danielle Albers Szafir, Steve Haroz, Michael Gleicher, and Steven Franconeri. Four types of ensemble coding in data visualizations. *Journal of vision*, 16(5):11–11, 2016.

[179] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. Tesma and catg: automated test generation tools for models of enterprise applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 717–720. IEEE Press, 2015.

[180] Ashish Tiwari, Adria Gascón, and Bruno Dutertre. Program synthesis using dual interpretation. In *Proc. International Conference on Automated Deduction*, pages 482–497. Springer, 2015.

[181] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.

[182] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 535–548, 2009.

[183] Theophanis Tsandilas. Structgraphics: Flexible visualization design through data-agnostic and reusable graphical structures. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[184] John W. Tukey. The future of data analysis. *The Annals of Mathematical Statistics*, 33(1):1–67, 1962.

[185] John W Tukey et al. *Exploratory data analysis*, volume 2. Reading, Mass., 1977.

[186] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6):287–296, June 2013.

[187] Jacob VanderPlas, Brian E Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. Altair: interactive statistical visualizations for python. *Journal of open source software*, 3(32):1057, 2018.

[188] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, pages 49–68, 2009.

[189] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, pages 425–446, 2010.

[190] Vega-Lite. *Vega-Lite Examples*, 2019.

[191] Ruben Verborgh and Max De Wilde. *Using OpenRefine*. Packt Publishing Ltd, 2013.

[192] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 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, June 18-23, 2017*, pages 452–466, 2017.

[193] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Speeding up symbolic reasoning for relational queries. *PACMPL*, 2(OOPSLA):157:1–157:25, 2018.

[194] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. Visualization by example. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[195] Chenglong Wang, Yu Feng, Rastislav Bodík, Isil Dillig, Alvin Cheung, and Amy J. Ko. Falx: Synthesis-powered visualization authoring. In Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, and Steven Mark Drucker, editors, *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, pages 106:1–106:15. ACM, 2021.

[196] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *PACMPL*, 1(OOPSLA):62:1–62:26, 2017.

[197] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. pages 62:1–62:26. ACM, 2017.

[198] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. pages 63:1–63:30. ACM, 2018.

[199] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. Fidex: Filtering spreadsheet data using examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 195–213, New York, NY, USA, 2016. ACM.

[200] Y. Richard Wang and Stuart E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 519–538, 1990.

[201] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL):56:1–56:29, 2018.

[202] Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

[203] Hadley Wickham. ggplot2. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(2):180–185, 2011.

[204] Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, et al. Welcome to the tidyverse. *Journal of open source software*, 4(43):1686, 2019.

[205] Hadley Wickham et al. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014.

[206] Leland Wilkinson. The grammar of graphics. In *Handbook of Computational Statistics*, pages 375–414. Springer, 2012.

[207] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. Goals, process, and challenges of exploratory data analysis: An interview study. *arXiv preprint arXiv:1911.00568*, 2019.

[208] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. Towards a general-purpose query language for visualization

recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, page 4, 2016.

[209] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE Trans. Vis. Comput. Graph.*, 22(1):649–658, 2016.

[210] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. Voyager 2: Augmenting visual analysis with partial view specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017.*, pages 2648–2659, 2017.

[211] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 508–521. ACM, 2016.

[212] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. pages 63:1–63:26. ACM, 2017.

[213] Bowen Yu and Cláudio T Silva. Flowsense: A natural language interface for visual data exploration within a dataflow system. *IEEE transactions on visualization and computer graphics*, 26(1):1–11, 2019.

[214] Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2019.

[215] Jian Zhang, Chen Xu, and S. C. Cheung. Automatic generation of database instances for white-box testing. In *25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, 8-12 October 2001, Chicago, IL, USA*, pages 161–165, 2001.

[216] Jian Zhang and Hantao Zhang. Sem: a system for enumerating models. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence-Volume 1*, pages 298–303. Morgan Kaufmann Publishers Inc., 1995.

[217] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. of International Conference on Computer-Aided Design*, pages 279–285. IEEE Computer Society, 2001.

[218] Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 224–234, 2013.

[219] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. Interactive program synthesis by augmented examples.

[220] Moshé M Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438. ACM, 1975.