Synthesizing Analytical SQL Queries from Computation Demonstration

Xiangyu Zhou
xiangz28@cs.washington.edu
University of Washington
Seattle, WA, USA

Rastislav Bodik
bodik@cs.washington.edu
University of Washington
Seattle, WA, USA

Alvin Cheung
akcheung@cs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Chenglong Wang
chenglong.wang@microsoft.com
Microsoft Research
Redmond, WA, USA

Abstract

Analytical SQL is widely used in modern database applications and data analysis. However, its partitioning and grouping operators are challenging for novice users. Unfortunately, programming by example, shown effective on standard SQL, are less attractive because examples for analytical queries are more laborious to solve by hand.

To make demonstrations easier to create, we designed a new end-user specification, *programming by computation demonstration*, that allows the user to demonstrate the task using a (possibly incomplete) cell-level computation trace. This specification is exploited in a new abstraction-based synthesis algorithm to prove that a partially formed query cannot be completed to satisfy the specification, allowing us to prune the search space.

We implemented our approach in a tool named SICKLE and tested it on 80 real-world analytical SQL tasks. Results show that even from small demonstrations, SICKLE can solve 76 tasks, in 12.8 seconds on average, while the prior approaches can solve only 60 tasks and are on average 22.5× slower. Our user study with 13 participants reveals that our specification increases user efficiency and confidence on challenging tasks.

1 Introduction

While standard SQL is the de facto language for *data retrieval tasks*, the language of choice for database applications, wrangling routines and other *computation tasks* is analytical SQL [6, 17, 18]. While retrieval queries tend to use operators like select, join and filter, analytical queries see more frequent use of group-aggregation, custom arithmetic functions, and subqueries. Analytical SQL is significantly more powerful than SQL, thanks to its support of the critical "`Partition By`" operator, which can express partition-aggregation tasks such as the moving average, window aggregation, and ranking.

The combination of partitioning and aggregation computes values over a group of rows and returns a result for *each row*. This differs from aggregation queries in SQL (constructed from the "`Group By`" operator and aggregation functions), which returns only a single result for *each group of rows*. For example, given the input table T below, the aggregation query q_1 that sums the sales value for each product ID returns the table T_1 with two rows. In contrast, the analytical query q_2 partitions the table in place and calculates aggregated values in a new column. Analytical SQL's increased expressiveness make it harder to program, especially for inexperienced engineers and data scientists [24].

q_1 : **Select** ID, Sum(Sales) **From** T **Group By** ID
q_2 : **Select** ID, Quarter, CumSum(Sales) **Over** (**Partition By** ID) **From** T;

Recently, program synthesizers, especially programming-by-example (PBE) tools, have been successfully adopted to solve similar programming challenges in domains such as string manipulation [15, 28], SQL/Datalog query [29, 31, 36,

39], data wrangling [11, 12, 40]. In such tools, the user provides one or more pairs of small input-output example values (I, O); the tool then synthesizes a program (or a list of ranked programs) p that satisfies $[\![p(I)]\!] = O$. Though promising, PBE is less effective on analytical tasks for two reasons:

- **Increased specification effort:** As analytical functions operate on partitions of rows, a representative example often contains results computed from multiple groups of rows. Providing an example requires users to manually find all the values that belong to a specific group and compute the aggregated value from them. This is costly when groups are created by partitioning and often results in erroneous examples (as reported in [41]).
- **Limited synthesis efficiency:** Because examples capture only output values, the synthesizer needs to "guess" the functions used in the computation. This makes reasoning about analytical queries much harder and makes the synthesizer difficult to scale up, especially when custom arithmetic functions are needed.

In this paper, we propose *programming by computation demonstration* to solve the specification challenge. Our design is motivated by the observation that online users often leverage example formulas to explain tasks. In our new specification:

1. The user provides the input tables and a *partial* output table to demonstrate the task.
2. In the partial output table, the user provides expressions (similar to Excel formulas) to demonstrate how output values are computed from values in the inputs, as opposed to providing only the final values.
3. The user can optionally provide *incomplete expressions* when the expression involves many values to further reduce specification effort.

We conducted a user study with 13 participants on 6 real-world scenarios and compared user experiences using computation demonstration versus classical PBE (Section 5). Our quantitative results show that both approaches have similar user efficiencies on simpler tasks, but our method made users more efficient on harder tasks. Our qualitative results furthermore show that users are more confident using computation demonstration and generally prefer it over examples.

Because a computation demonstration constrains the structure of the desired computation but does not provide the output value, the classical PBE objective "$[\![p(I)]\!] = O$" no longer characterizes our synthesis problem. We develop a new synthesis task formulation based on *provenance consistency*, which is a property of a program whose computation traces generalize the user demonstration. This formulation builds on a new query semantics (Section 3) called the *provenance-tracking query semantics* that keeps track of cell-level data provenance during evaluation.

Our new specification raises new challenges and opportunities for algorithm design. A promising approach for our synthesis task is abstraction-based search approach that has

been successfully adopted in synthesis of list and table transformation programs [11, 22, 39, 40]: the synthesizer iteratively enumerates and prunes partial programs until the correct solution is found. The key in this approach is to leverage an abstract interpreter to reason about the realizability of the synthesis goal given a partial program, and the synthesizer's scalability depends on whether the abstraction can effectively capture inconsistent behavior of incorrect partial programs for early pruning. However, existing abstractions are insufficient for analytical SQL (as shown in our experiments Section 5): abstractions based on high-level type properties of full outputs (row/column/group numbers) [11, 22] are not ideal for partial demonstration whose type information is incomplete [40]; value-based abstractions that reason program properties by tracking concrete value flow [39, 40] are also insufficient, as analytical operators can mutate values – making them intractable by current abstractions; and computation graph based abstractions [3] are inapplicable due to the user's computation demonstration contains partial expressions.

Hence, our second contribution is a new abstract semantics, *abstract data provenance*, developed to better capture the semantics of partial queries with analytical computations. Our key insight is to leverage the fine-grained provenance information presented by computation demonstrations to prune partial queries that cannot be completed. Given a partial query, the abstract analyzer over-approximates the provenances of each output cell and checks their consistency with the user demonstration. This analysis reveals cell-level provenance inconsistency introduced by infeasible queries that other abstractions cannot capture. The new abstraction lets our algorithm on average visit 97.08% less queries.

We implemented our algorithm in a tool called Sɪᴄᴋʟᴇ and evaluated it on 80 tasks: 60 tasks from online posts and 20 from the popular TPC-DS [23] database benchmark. Our experiments show that Sɪᴄᴋʟᴇ can solve 76 of these benchmarks and outperforms prior state-of-the-art techniques [11, 40] that only solves 60 and 51 benchmarks. On benchmarks all techniques can solve, Sɪᴄᴋʟᴇ is on average 22.5× faster.

Contributions. This paper's contributions include:

- We introduce the analytical SQL query synthesis problem and present a new user specification, *computation demonstration*, that makes specifying analytical tasks easier.
- We conducted a user study to compare user experiences in creating computation demonstrations and examples.
- We propose an abstraction-based algorithm with a new language abstraction, *abstract data provenance*, that can better utilize finer-grained computation information to dramatically prune infeasible programs.
- We implemented our approach as a practical tool, Sɪᴄᴋʟᴇ, and evaluated it on 80 real-world benchmarks. Results

show that Sickle can efficiently solve practical analytical SQL tasks with much better performance comparing against prior state-of-the-art algorithms.

2 Overview

In this section, we use an example to illustrate how a user specifies an analytical task using computation demonstration, and how Sickle solves the problem.

The Task. Suppose the user has an input table T (Figure 1) with the number of people enrolled in a health program split by city, quarter, and age group (the `Population` column shows the city population). For each city, the user wants to compute the percentage of total population enrolled in the program at the end of each quarter. This can be done using the analytic SQL query q shown in Figure 2: (1) the subquery q_1 calculates the total enrollment in each city for each quarter (2) the subquery q_2 then calculates the cumulative sum of the enrollment number for each city at the end of each quarter, by partitioning on `City` and applying `Cumsum` on the column `C1` generated by q_1; (3) the subquery q_3 finally calculates the enrollment percentage based on `C2` and `Population`. Figure 1 shows the output t_i of each subquery q_i.

Because this is a challenging task that requires using analytical function together with subqueries, grouping and arithmetic, the user decides to solve it with Sickle.

2.1 The User Specification

To use Sickle, the user creates a *computation demonstration* to demonstrate the task. Instead of providing the full output (t_3 in Figure 1), the user creates a partial output that shows how output cells are derived from the input.

The User Demonstration. Figure 3 shows the user demo $\mathcal{E}$, constructed by expressions and references to input table cells. Here, the user demonstrates how the percentage is computed for quarter 1 and 4 of city A (i.e., the first and fourth row of the output t_3 in Figure 3):

- The percentage of people enrolled at the end of quarter 1 in city A is computed by (1) summing up the number of people enrolled in both age groups $\text{sum}(1667, 1367)$, and (2) dividing the sum with the city population 5668.
- For quarter 4, the percentage requires dividing the number of people from both age groups enrolled throughout quarters 1 to 4 with the population 5668. As it requires collecting many values from T, the user uses an incomplete expression $\text{sum}(1667, 1367, \diamond, 432)$ in the demonstration ($\diamond$ denotes omitted values) to save effort, where the enrollment from rows 3-7 of T are omitted.

With a PBE tool, the user would need to manually group values from the input and compute all the derived values (t_3 in Figure 1) with considerable effort; but using Sickle, the user constructs the demonstration easily by dragging and

dropping input cells to create a *partial output*. Additionally, the support for incomplete expressions permits users to not exhaustively find all values needed for the sum expression, and such support would be especially beneficial if values needed the expression are not in a continuous region in the input data (e.g., if the user wants to compute the percentage of enrolled Youth participants over all enrolled participants in all cities over all four quarters). Because the user's drag-and-drop actions provide references to input cells (not just their values), Sickle internally stores cell references in $\mathcal{E}$ and uses them in the synthesis process (Figure 3).

Synthesis Consistency Criteria. With our new specification, we formulate the new *computation consistency* criteria for deciding whether a synthesized query q satisfies the user demonstration. We first introduce a new semantics of analytical SQL: *provenance-tracking query semantics*. Under this semantics, query operators are "term rewriters" that transform and simplify cell-level expressions symbolically to keep track of cell-level data provenance.

Figure 4 shows the provenance-tracking evaluation result for the example in Figure 1. For example, the first row of the `City` column in $t_3^\star$ is derived by the **Group By** operator in the subquery q_1; since the **Group By** uses both $T[1, 1]$ and $T[2, 1]$ in the same group, the cell content is $\boxed{\text{group}\{T[1, 1], T[1, 2]\}}$. The percentage in row 4 is obtained by (1) summing up the enrolled number from row 1 to row 8 in T, and (2) applying $\lambda x, y.x/y * 100\%$ to the sum and the city population.

We now define the synthesis objective based on the following provenance consistency criteria: we consider a query consistent with the user demonstration if its provenance tracking evaluation output $t_0^\star$ satisfies the following criteria:

- There exists a subtable $t_s^\star$ of $t^\star$ such that every cell in $t_s^\star$ generalizes each cell in the user demonstration $\mathcal{E}$ based on their provenance information.

In our example, the query q in Section 2 is consistent with $\mathcal{E}$ (Figure 3): the subtable $t_s^\star$ of $t_3^\star$ (provenance tracking evaluation output of q) that witnesses the property is the table consists of only rows 1 and 4 of $t_3^\star$. The generalization is shown by: the term $\boxed{T[1, 1]}$ (from row 1, column c_1 in $\mathcal{E}$) is subsumed by the term $\boxed{\text{group}\{T[1, 1], T[2, 1]\}}$ (row 1, column `City` in $t_3^\star$) [1]; the term $\boxed{\text{sum}\left(\begin{smallmatrix}T[1, 4], T[2, 4]\\ \diamond, T[8, 4]\end{smallmatrix}\right)/T[7, 5] * 100\%}$ in $\mathcal{E}$ is subsumed by $\boxed{\text{sum}\left(\begin{smallmatrix}T[1, 4], T[2, 4], T[3, 4], T[4, 4]\\ T[5, 4], T[6, 4], T[7, 4], T[8, 4]\end{smallmatrix}\right)/\text{group}\{\begin{smallmatrix}T[7, 5]\\ T[8, 5]\end{smallmatrix}\} * 100\%}$ in $t_3^\star$ (because the user's omitted values denoted by "$\diamond$" can match any number of other values).

Now, our goal is to synthesize the computation-consistent query q from the input T and the user demonstration $\mathcal{E}$.

---

[1]Since all of the values in the same group-by column have the same value, (e.g., "A" in this case); the user can use either $T[1, 1]$ or $T[1, 2]$ in the demonstration of the cell (or both), this makes user demonstration flexible.

Figure 1. The running example: given the input table T, the user aims to calculate the percentage of total population that have enrolled in the program at the end of the each quarter (for each city). The solution requires three steps (q in Figure 2).

```
-- q3: calculate percentage
Select City, Quarter, C2 / Population * 100%
From (
  -- q2: calculate cumulative enrolled number
  Select City, Quarter, Population,
         Cumsum(C1) Over (Partition By City) As C2
  From (
    -- q1: calculate the number of people enrolled in
    --     the program for each city / quarter
    Select City, Quarter, Population, Sum(Enrolled) As C1
    From T
    Group By City, Quarter, Population));
```

Figure 2. The desired query q to solve the task in Figure 1.

Figure 3. The user demonstration $\mathcal{E}$ constructed by dragging and dropping input cells to the partial output table. The symbol "◇" denotes that the user omitted some values in the expression. The table in the bottom shows internal representation of the demonstration (based on references of input cells). $T[i, j]$ denotes the reference to the cell at row i and column j in the input T (index starts from 1).

2.2 The Synthesis algorithm

To solve the synthesis problem, our algorithm adopts an abstraction-based enumerative search algorithm. For simplicity, we consider only three query operators here: group

Figure 4. The evaluation result of q (Figure 2) on the input T (from Figure 1) under provenance tracking semantics. The output $t_3^\star$ shows how t_3's cells (Figure 1) are derived from T.

(grouping and aggregation), partition (partition and aggregation), and arithmetic. We also represent queries in the following "instruction" style. The query q in Figure 2 is shown below (the instruction at line i corresponds to the subquery q_i, and t_i is to the output of q_i):

```
t1 <- group(T, [City, Quarter, Population], sum, Enrolled)
t2 <- partition(t1, [City], cumsum, C1)
t3 <- arithmetic(t2, λx, y.x/y * 100%, [C2, Population])
```

Enumerative Search. Figure 5 shows the enumerative search process to solve the running example. From the input table T, SICKLE first enumerates query skeletons formed by compositions of query operators with no instantiated parameters (represented as holes "□") and then iteratively instantiates their parameters (e.g., the query q_C is generated by instantiating the first hole of its parent). The search tree expands with the breadth-first-search strategy.

During the search process, an abstract analyzer analyzes whether a partial query can realize the synthesis goal and prunes infeasible ones. The synthesizer terminates when it finds N (a synthesizer parameter, SICKLE uses $N = 10$) consistent solutions or timeout. The search space can be very
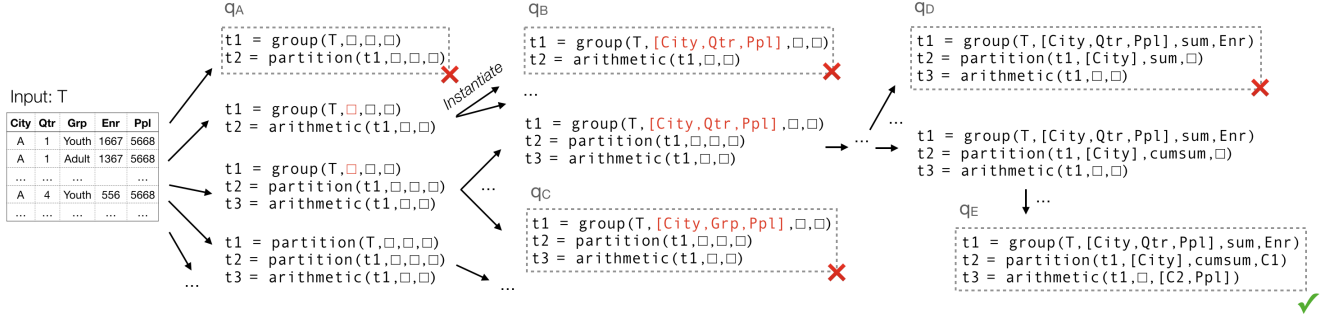
Figure 5. The enumerative search algorithm to solve the running example ($\mathcal{E}$ from Figure 3). Queries in dashed boxes are leaf nodes in the search tree. Infeasible partial programs are pruned based on abstract reasoning.

large in practice (e.g., the search space for the running example contains 1,181,224 queries even only queries up to size 3 are considered), especially because the candidate numbers for many query parameters (e.g., grouping columns) grow exponentially with the input column number. Thus, SICKLE's performance depends on whether the abstract analyzer can prune infeasible partial queries early.

Pruning with Abstract Provenance Analysis. Our abstraction design is based on the following key observation: incorrect analytical queries are often resulted by wrong partitioning or grouping of columns, including (1) values that are supposed to be aggregated into the same group are spread in different groups, and (2) values supposedly to appear in different groups in the demonstration are wrongly aggregated together. Because recognizing such inconsistency requires fine-grained cell level information tracking, existing abstractions (e.g., type abstractions [12, 27], value abstractions [39]) are insufficient for effective pruning.

We introduce a new abstraction, *abstract data provenance*, that over-approximates cell-level data provenance to keep track of how input cells will be partitioned / grouped / aggregated throughout the computation process for more effective pruning. Given a partial query q (with holes), the abstract analyzer returns a table $t^\circ = [[q(T)]]^\circ$ (denotes evaluating q on T using abstract semantics). Each cell $t^\circ[i, j]$ is a set of input cells that contains all possible input values that can possibly flow into the position i, j. This is an over-approximation of data provenance: for any possible instantiations of q, its output cell at $[i, j]$ may only use a subset of input cells from $t^\circ[i, j]$. The analyzer then checks if the provenance information from the user example $\mathcal{E}$ is consistent with t°.

Figure 6 shows how our technique analyzes and prunes the incorrect partial query q_B in Figure 5. q_B is a partial query with only group columns instantiated; other parameters are still "□"s (Figure 6 right). Despite unknown parameters, we can analyze the data provenance abstractly (Figure 6 left).

First, for the subquery q_{B1} `t1 <- group(T, [City, Quarter, Population], □, □)`:

- Because T is grouped by `City`, `Quarter` and `Population`, the first two rows in T are grouped together. Thus, $t_1^\circ[1, 1]$ (the

cell at row 1 column 1 from the output of q_{B1}) comes from $T[1, 1], T[2, 1]$, thus its provenance is $\{T[1, 1], T[2, 1]\}$; the provenances for $t_1^\circ[1, 2]$ and $t_1^\circ[1, 3]$ are derived similarly (shown by gray arrows).

- Because the aggregation function and column are unknown, we *over-approximate* the provenance of cells in C1 (the new column generated by aggregation). For example, for $t_1^\circ[1, 4]$, because the aggregation function can use any columns outside of the grouping columns, its abstract provenance is $S = \{T[1, 3], T[2, 3], T[1, 4], T[2, 4]\}$. This shows only values from the set S will flow from T to $t_1[1, 4]$ regardless of how q_{B1} is instantiated.

The analysis generates t_1°, the abstract output of q_{B1}. Then, for the subquery q_{B2} `t2 <- arithmetic(t1, □, □)`:

- Because arithmetic does not modify existing columns, columns 1,2,3,4 of t_2° are directly inherited from columns 1,2,3,4 of t_1° (e.g., the provenance of $t_2^\circ[1, 1]$ is $\{t_1^\circ[1, 1]\}$).
- Due to the unknown arithmetic function and its parameters, we need to over-approximate the provenance for cells in the new column. For $t_2^\circ[1, 5]$, because any cells from row 1 of t_1° can potentially be used in the arithmetic computation, its provenance is $\{t_1^\circ[1, 1], t_1^\circ[1, 2], t_1^\circ[1, 3], t_1^\circ[1, 4]\}$ (shown by the orange box in Figure 6).

This analysis generates the abstract output t_2°. By combining the provenance information in t_1° and t_2°, the final abstract output of q_B is shown as t_2° in Figure 6 right.

Given t_2°, we check its consistency with the user demonstration $\mathcal{E}$: can we extract a subtable t° of t_2° such that there exists a one-to-one mapping between cells in $\mathcal{E}$ and t°? This is a necessary condition q_B need to satisfy to achieve the consistency goal from Section 2.1.

However, we cannot find such a consistent mapping between them, because the cell $\mathcal{E}[2, 3]$ (colored grey) has no matching cell in t_2°. The cell $\mathcal{E}[2, 3]$ requires using at least values $\{T[1, 4], T[2, 4], T[8, 4]\}$, but there is no cell in t_2° that contains these values, meaning that there is no way for these values in the input T to flow into the same cell in the output t_2°. Thus, no matter how q_B is instantiated, it cannot realize our goal, and we can prune q_B. The same reasoning can
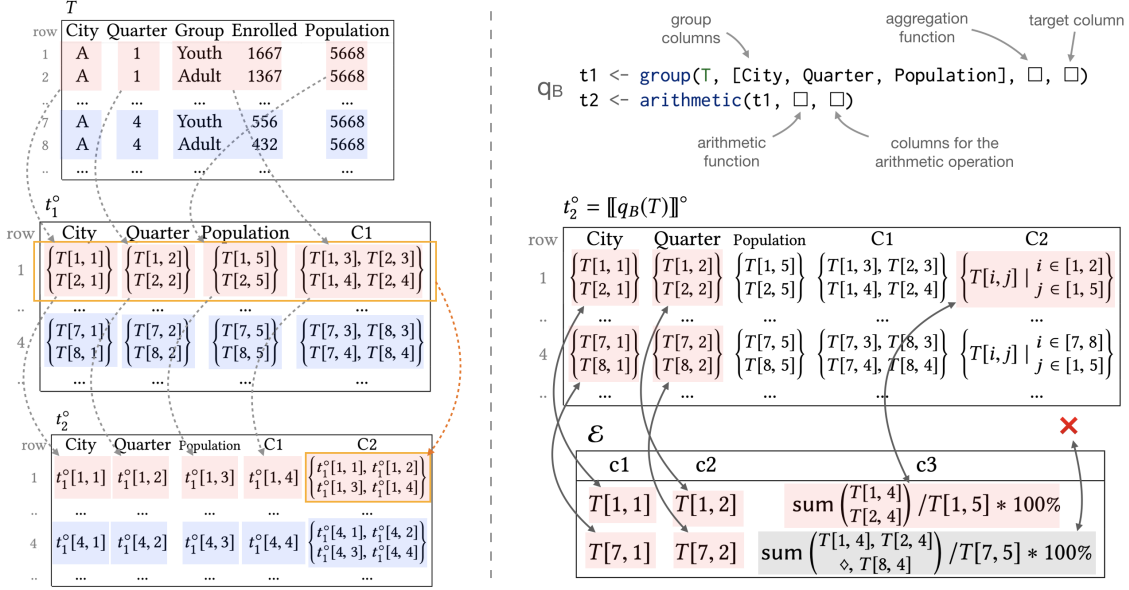
Figure 6. Pruning the partial query q_B using abstract provenance analysis. Right: q_B can be pruned because its abstract output t_2° is inconsistent with the user demonstration $\mathcal{E}$. Left: how t_2° is derived from the input T using abstract provenance analysis.

prune many other infeasible queries in Figure 5: the synthesizer visited only 1,453 (partial and concrete) programs before finding the solution q_E within only 6 seconds.

Note that this abstract query q_B cannot be pruned if we use value-based abstract semantics as in prior work [39, 40] *even if the user example contains full output and without incomplete expressions*: as shown in the alternative table t_2^v below, we lack information about aggregation/arithmetic functions and their parameters, hence we cannot derive a concrete value to approximate the values in columns C1 and C2 (thus represented as unknown). These unknown values can match any value in $\mathcal{E}$, even with a full output example, the query q_B cannot be pruned.

t_2^v (derived from $q_B(T)$ using value abstraction)

row	City	Quarter	Population	C1	C2
1	A	1	5668	unknown	unknown
..	...	...	...	...	...
4	A	4	5668	unknown	unknown
..	...	...	...	...	...

In summary, the user solves the analytical task using a computation demonstration. The benefit of our approach is that complex queries can be easily demonstrated in individual groups: the complexity of an analytical query often lies in how to join and partition the table properly, but in the demonstration, the user only needs to interact with values from *one group*, thus avoiding the need to consider complex joining/grouping/partitioning behind the scenes. SICKLE makes this new synthesis problem tractable by leveraging abstract provenance analysis for dramatic pruning.

We next formally present the synthesis specification (Section 3), the abstraction-based pruning algorithm (Section 4)

and demonstrate the practical use of SICKLE through experiments on real world analytical SQL tasks (Section 5).

3 The Synthesis Problem

In this section, we formally describe analytical SQL and the program-by-computation-demonstration problem.

In the following, we use q for queries, T and t for tables, and c for table columns. We use the bar notation $\bar{x}$ to denote a list of x (e.g., $\bar{T}$ stands for a list of tables $T_1, \ldots, T_n$ and $\bar{c}$ for columns $c_1, \ldots, c_n$), and $\bar{f}(\bar{x})$ for a list of function applications $[f_1(x_1), \ldots, f_n(x_n)]$.

3.1 Analytical SQL

Table. A table is an ordered bag of tuples, and each tuple is a list of values (strings or numbers). A table T with n columns and m rows is represented as $\{[v_{11}, ..., v_{1n}], ..., [v_{m1}, ..., v_{mn}]\}$.

The choice of using ordered bag is to support sorting in intermediate computation steps, which is essential for order-dependent aggregation functions like rank and cumsum. The ordered bag representation also permits the use of set operators as in normal bags, and allows us to refer to table rows based on their index (like lists). For simplicity, we omit table column names in our formulation, and table columns are referred based on column indexes. We use $T[i, j]$ to refer to the cell at row i and column j, $T[i, *]$ for the i-th row in T, and $T[\bar{c}]$ for the result of projecting T on columns $\bar{c}$. We also use $T_1 \subseteq T_2$ to denote table containment, and $T_1 \times T_2$ for table crossproduct. Two tables are equivalent $(T_1 = T_2)$ if they contain each other regardless of orders $(T_1 \subseteq T_2 \land T_2 \subseteq T_1)$ (because row order only matters when used with order-dependent aggregation functions).

$$q \leftarrow T \mid \mathsf{filter}(q, p) \mid \mathsf{join}(q_1, q_2) \mid \mathsf{proj}(q, \bar{c})$$
$$\mid \; \mathsf{sort}(q, \bar{c}, op) \mid \mathsf{left\_join}(q_1, q_2, p) \mid \mathsf{arithmetic}(q, \gamma(\bar{c}))$$
$$\mid \; \mathsf{group}(q, \bar{c}, \alpha(c)) \mid \mathsf{partition}(q, c, \alpha'(c))$$
$$p \leftarrow p_1 \; \mathsf{and} \; p_2 \mid \mathsf{true} \mid \mathsf{false} \mid c_1 \; op \; c_2$$
$$\alpha \leftarrow \mathsf{sum} \mid \mathsf{avg} \mid \mathsf{max} \mid \mathsf{min} \mid \mathsf{count}$$
$$\alpha' \leftarrow \alpha \mid \mathsf{dense\_rank} \mid \mathsf{rank} \mid \mathsf{cumsum}$$
$$op \leftarrow \; < \; \mid \; \le \; \mid \; == \; \mid \; > \; \mid \; \ge$$

Figure 7. The analytical SQL language, where c denotes a column index, γ refers to an arithmetic function.

$$
\begin{array}{llll}
T^\star & \leftarrow & [r^\star, \ldots, r_n^\star] & \\
r^\star & \leftarrow & [e_1^\star, \ldots, e_m^\star] & \\
e^\star & \leftarrow & const & \\
 & \mid & T_k[i, j] & \\
 & \mid & f(e_1^\star, \ldots, e_l^\star) & \\
 & \mid & \mathsf{group}\{e_1^\star, \ldots, e_l^\star\} & \\
\end{array}
\qquad
\begin{array}{lll}
\mathcal{E} & \leftarrow & [r_1, \ldots, r_n] \\
r & \leftarrow & [e_1, \ldots, e_m] \\
e & \leftarrow & const \\
 & \mid & T_k[i, j] \\
 & \mid & f(e_1, \ldots, e_l) \\
 & \mid & f^\diamond(e_1, \ldots, e_l) \\
\end{array}
$$

Figure 8. Definitions of the provenance-embedded table $T^\star$ and the user demonstration $\mathcal{E}$. Metavariable f refers to an aggregator or an arithmetic function, $T_k[i, j]$ denotes the reference to cell i, j from the input table T_k, and "$f^\diamond$" stands for a function f with some parameters omitted by the user in their demonstration.

Analytical SQL. Figure 7 defines the analytical SQL language $\mathcal{L}_{SQL}$ used in our paper. A query q is formed by compositions of basic constructors proj (projection), filter, join, group, partition, sort (sort rows), aggregate, and arithmetic constructors. Comparing to prior work [39], the key extensions are portioning, window functions (cumsum, rank) and arithmetic functions. Note that aggregation functions α can be used as analytical functions α' (but not otherwise). The semantics of analytical SQL follows the standard in modern databases. We use the notation $[\![q(\bar{T})]\!]$ to denote running q on input tables $\bar{T}$, and the output is a table.

Provenance-Tracking Query Semantics. In addition to standard query evaluation rules, we introduce provenance-tracking semantics for analytical SQL. Under this semantics, each operator is a term rewriter that builds and transforms references to input table cells to keep track of the data provenance. Evaluating a query q on inputs $\bar{T}$ under provenance-tracking semantics produces a *provenance-embedded table* (denoted as $T^\star$). As shown in Figure 8 left, each cell in $T^\star$ is an expression $e^\star$ that stores how the cell is derived from the inputs. A cell $e^\star$ is composed by f (representing aggregation or arithmetic results), $\mathsf{group}\{\ldots\}$ (results from the group operator), reference $T_k[i, j]$ and constants. A provenance embedded table $T^\star$ can be further evaluated into a table T by evaluating the expressions in every table cell (denoted by $T = [\![T^\star]\!]$). We use $[\![q(\bar{T})]\!]^\star$ to denote the provenance-tracking evaluation process. Rules for key SQL operators are shown in Figure 9. For example, for the

$$[\![T]\!]^\star = \left\{ \left[\text{``}T[i, c]\text{''} \mid c \in \mathsf{columns}(T^\star) \right] \middle| i \in [1, \mathsf{rowNum}(T^\star)] \right\}$$

$$[\![\mathsf{filter}(q, p)(\bar{T})]\!]^\star = \; let \; T^\star = [\![q(\bar{T})]\!]^\star \; in$$
$$\left\{ T^\star[i, *] \middle| i \in [1, \mathsf{rowNum}(T^\star)], \; [\![p(T^\star[i, *])]\!] = \mathsf{true} \right\}$$

$$[\![\mathsf{proj}(q, \bar{c})(\bar{T})]\!]^\star = [\![q(\bar{T})]\!]^\star[\bar{c}]$$

$$[\![\mathsf{join}(q_1, q_2)(\bar{T})]\!]^\star = [\![q_1(\bar{T})]\!]^\star \times [\![q_2(\bar{T})]\!]^\star$$

$$[\![\mathsf{left\_join}(q_1, q_2, p)(\bar{T})]\!]^\star = \; let \; \begin{cases} T_1^\star = [\![q_1(\bar{T})]\!]^\star \\ T_2^\star = [\![q_2(\bar{T})]\!]^\star \end{cases} in$$
$$\left\{ T_1^\star[i, *] + T_2^\star[j, *] \middle| \begin{array}{l} i \in [1, \mathsf{rowNum}(T_1^\star)], \\ j \in [1, \mathsf{rowNum}(T_2^\star)] \\ [\![p(T_1^\star[i, *] + T_2^\star[j, *])]\!] = \mathsf{true} \end{array} \right\}$$
$$\cup \left\{ T_1^\star[i, *] + \bar{\emptyset} \middle| \begin{array}{l} i \in [1, \mathsf{rowNum}(T_1^\star)], \; j \in [1, \mathsf{rowNum}(T_2^\star)] \\ [\![p(T_1^\star[i, *] + T_2^\star[j, *])]\!] = \mathsf{false} \end{array} \right\}$$

$$[\![\mathsf{sort}(q, \bar{c}, op)(\bar{T})]\!]^\star = \; let \; T^\star = [\![q(\bar{T})]\!]^\star \; in$$
$$\left\{ T^\star[k, *] \mid T^\star[k, \bar{c}] \; op \; T^\star[k+1, \bar{c}], k \in [1, \mathsf{rowNum}(T^\star)] \right\}$$

$$[\![\mathsf{group}(q, \bar{c}, \alpha(c_t))(\bar{T})]\!]^\star =$$
$$let \; T^\star = [\![q(\bar{T})]\!]^\star, \; G = \mathsf{extractGroups}([\![T^\star[\bar{c}]]\!]) \; in$$
$$\left\{ \begin{array}{l} [\mathsf{group} \; \{T^\star[k, c] \mid k \in g\} \mid c \in \bar{c}] \\ + [\alpha(T^\star[k, c_t] \mid k \in g)] \end{array} \middle| g \in G \right\}$$

$$[\![\mathsf{partition}(q, \bar{c}, \alpha'(c_t))(\bar{T})]\!]^\star =$$
$$let \; T^\star = [\![q(\bar{T})]\!]^\star, \; G = \mathsf{extractGroups}([\![T^\star[\bar{c}]]\!]) \; in$$
$$\left\{ \begin{array}{l} [T^\star[i, j] \mid j \in [1, \mathsf{colNum}(T^\star)]] \\ + [\alpha'(T^\star[k, c_t] \mid k, i \in g \in G)] \end{array} \middle| i \in [1, \mathsf{rowNum}(T^\star)] \right\}$$

$$[\![\mathsf{arithmetic}(q, \gamma, \bar{c})(\bar{T})]\!]^\star = \; let \; T^\star = [\![q(\bar{T})]\!]^\star \; in$$
$$\left\{ \begin{array}{l} [T^\star[i, j] \mid j \in [1, \mathsf{colNum}(T^\star)]] \\ + [\gamma(T[i, c] \mid c \in \bar{c})] \end{array} \middle| i \in [1, \mathsf{rowNum}(T^\star)] \right\}$$

$$\mathsf{extractGroup}(T) = \left\{ S \middle| \begin{array}{l} S \subseteq [1, \mathsf{rowNum}(T^\star)] \\ \forall i, j \in S, T[i, *] = T[j, *] \\ \forall k \notin S. \nexists j \in S.T[k, *] = T[j, *] \end{array} \right\}$$

Figure 9. Provenance tracking semantics of analytical SQL operators. The operator '$l_1 + l_2$' concatenates two lists; the function extractGroup partitions the row indexes of table T into disjoint equivalence sets (rows from the same set are equal, and rows from different sets are not equal).

partition$(q, \bar{c}, \alpha'(c_t))$ operator, the rule first partitions the input $T^\star$ based on its partitioning columns $\bar{c}$ (using the auxiliary function extractGroups). Then, a new aggregated value is generated for each row i: the rule finds the group g that row i belongs to it and constructs an expression $\alpha'(\ldots)$ that includes all values in column c_t from the group g. Also, in the evaluation process, SICKLE simplifies consecutive applications of aggregation functions like $f(f(a, b), c)$ into $f(a, b, c)$ for $f \in \{\mathsf{sum}, \mathsf{max}, \mathsf{min}\}$. This let SICKLE better reason about semantically equivalent aggregations (we do not consider general term equivalence checking as we focus on analytical

SQL; in future, our approach can also work with advanced equivalence checker [45] when applied to new domains).

3.2 The Synthesis Task

We now define the specification language in which the user provides demonstrations and formulates the synthesis task.

The User Specification. Besides input tables $\bar{T}$, the user provides a demonstration $\mathcal{E}$ (which is a table) consists of *partial expressions* showing how each output cell is derived from inputs. As shown in Figure 8, each cell in $\mathcal{E}$ is an expression e, constructed from either a constant, a reference to the cell i, j in input table T_k ($T_k[i, j]$), an expression, or a partial expression with some omitted values. A function with omitted values is denoted as $f^{\diamond}(e_1, \ldots, e_l)$, meaning that the user only provides values $e_1, \ldots, e_l$ and leaves out some others: the omitted values can appear at either beginning, middle or end of the function (e.g., in Figure 3, omitted values are in the middle of the sum expression). Comparing to $T^{\star}$, cells in $\mathcal{E}$ are not expected to be constructed from group$\{\ldots\}$: the design rationale is that all cells in the same group have the same value; thus, the user only needs to use any one of them in the demonstration (as shown in Figure 3).

Consistency with the User Specification. To define the synthesis task, we first need to define what properties a provenance-embedded tables $T^{\star}$ needs to satisfy to be considered consistent with the user demonstration $\mathcal{E}$.

We first define cell-level consistency criteria: we say an expression $e^{\star}$ (from a provenance-embedded table) is consistent with an expression e (from the user demonstration $\mathcal{E}$) if e can be derived from $e^{\star}$ using rules from Figure 10 (denoted as $e \prec e^{\star}$). Concretely, $e^{\star}$ is consistent with e if (1) $e^{\star}$ and e are the same, (2) $e^{\star}$ is of form group$\{\bar{e}^{\star}\}$ and a member $e_i^{\star}$ is consistent with e, or (3) $e^{\star}$ and e are both expressions $f(\ldots)$ and arguments in e are subsumed by those in $e^{\star}$. In case (3), parameter order is not enforced if f is commutative (e.g., aggregation function sum(); arithmetic function $\lambda x, y.x * y$).

Table-level consistency rules are defined in Definition 1. Intuitively, a provenance-embedded table $T^{\star}$ is consistent with the user demonstration $\mathcal{E}$ if there exists a subtable $T_1^{\star} \subseteq T^{\star}$ satisfying the property that each cell $T_1^{\star}[i, j]$ generalizes the cell $\mathcal{E}[i, j]$ in the user demonstration.

Definition 1. *(Provenance Consistency)* Given a provenance-embedded table $T^{\star}$ and the user demonstration $\mathcal{E}$ with m rows and n columns, $T^{\star}$ is provenance-consistent with $\mathcal{E}$ if:
$$\exists T_1^{\star} \subseteq T^{\star}. \forall_{j \in [1, m]}^{i \in [1, n]}. \mathcal{E}[i, j] \prec T_1^{\star}[i, j].$$

Finally, we define the task: synthesize queries whose outputs are provenance consistent with the user demonstration.

Definition 2. *(The Synthesis Task)* Given input tables $\bar{T}$ and user demonstration $\mathcal{E}$, the synthesis task is to find analytical SQL queries, such that each q satisfies $\mathcal{E} \prec [[q(\bar{T})]]^{\star}$.

$$\frac{}{const \prec const} \qquad \frac{}{T[i, j] \prec T[i, j]} \qquad \frac{\exists e_i^{\star} \in \{\bar{e}^{\star}\}. e \prec e_i^{\star}}{e \prec \text{group}\{\bar{e}^{\star}\}}$$

$$\frac{\text{isCommutative}(f) \quad \exists. \{e_{k_1}^{\star} \ldots e_{k_m}^{\star}\} \subseteq \{e_1^{\star} \ldots e_n^{\star}\}. \forall i \in [1, m].e_i \prec e_{k_i}^{\star}}{f^{\diamond}(e_1, \ldots, e_m) \prec f(e_1^{\star}, \ldots, e_n^{\star})}$$

$$\frac{\neg\text{isCommutative}(f)}{\forall i \in [1, m].e_i \prec e_i^{\star}} \qquad \frac{f^{\diamond}(e_1, \ldots, e_n) \prec f(e_1^{\star}, \ldots, e_n^{\star})}{f(e_1, \ldots, e_n) \prec f(e_1^{\star}, \ldots, e_n^{\star})}$$
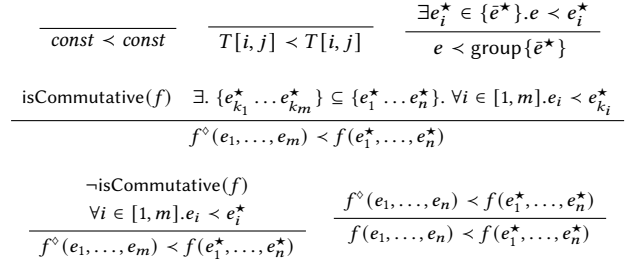
Figure 10. Rules that determine whether an expression $e^{\star}$ from a provenance-embedded table generalizes an expression e from the user demonstration ($e \prec e^{\star}$).

Algorithm 1 Top-level analytical SQL synthesis algorithm.

1: **procedure** SYNTHESIZE($\bar{T}, \mathcal{E}, depth, N$)
2:　　**input:** input tables $\bar{T}$, user demonstration $\mathcal{E}$, search depth *depth*, query number limit N.
3:　　**output:** a set of queries consistent with $\bar{T}$ and $\mathcal{E}$

4:　　$W \leftarrow \text{constructSkeletons}(\bar{T}, depth)$;
5:　　$R \leftarrow \emptyset$;
6:　　**while** $\neg W.\text{isEmpty}()$ **do**
7:　　　　$q \leftarrow W.\text{next}()$;
8:　　　　**if** IsConcrete(q) **then**
9:　　　　　　**if** $\mathcal{E} \prec [[q(\bar{T})]]^{\star}$ **then**
10:　　　　　　　　$R \leftarrow R \cup \{q\}$;
11:　　　　　　　　**if** $|R| \geq N$ **then return** R
12:　　　　**else continue**;
13:　　　　$\phi \leftarrow \text{AbstractReasoning}(q, \bar{T}, \mathcal{E})$;
14:　　　　**if** UNSAT(ϕ) **then continue**;
15:　　　　$\square_i \leftarrow \text{chooseNextHole}(q)$;
16:　　　　$D \leftarrow \text{inferDomain}(\square_i, q, \bar{T})$;
17:　　　　$W \leftarrow W \cup \{[\square_i \mapsto v]q \mid v \in D\}$
　　　　return R

Remarks. Since the user demonstration is an incomplete specification of the actual task, it is inherently ambiguous, and there could be multiple queries consistent with the user specification. As we will present more in detail in Section 5.1, our algorithm addresses this problem by synthesizing a set of queries consistent with the demonstration and present them to the user; it can work with existing program disambiguation framework to interactively resolve ambiguity (which is not the focus of our paper).

4 The Synthesis Algorithm

In this section, we introduce our synthesis algorithm: given input tables $\bar{T}$ and the user demonstration $\mathcal{E}$, our algorithm aims to find a set of queries consistent with $\mathcal{E}$.

Algorithm 1 shows our top-level synthesis algorithm: an abstraction-based enumerative search algorithm that iteratively enumerates and prunes partial programs until up-to-N programs consistent with the user specification are found.

SYNTHESIZE starts out by constructing query skeletons from input tables $\bar{T}$ up to size *depth* (line 4). A query skeleton is composed from top-level query operators in Figure 7 but with all arguments unfilled (represented as holes "□"). Then, the algorithm takes one partial query q out of the work list W at a time. If the query q is already concrete (i.e., no hole left to be instantiated), the algorithm checks if it is consistent with the user demonstration $\mathcal{E}$ and add it to the result set R if so (line 8-12). If q is partial, the algorithm conducts an abstract analysis to check if q can potentially realize the user demonstration $\mathcal{E}$. If the check fails, the partial query is pruned (line 14). Otherwise, the synthesizer expands the search space by (1) choosing the next hole $\square_i$ in q 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 it to W for further search (lines 15-17).

While the SYNTHESIZE procedure is a sound and complete way to explore the query space within the given depth, its efficiency relies on whether the abstract reasoning subroutine AbstractReasoning can effectively prune infeasible partial programs early to avoid search explosion. As discussed in Section 1, our key insight is to abstractly reason about cell-level provenance of the output table to check whether required values can flow from the input data to it in a way consistent with the user specification $\mathcal{E}$.

The Abstract Semantics. Given a partial query q and input tables $\bar{T}$, the abstract provenance analyzer returns an abstract output table T°, where each cell $T^\circ[i, j]$ stores a set of references to input table cells that *over-approximates* of all possible input cells that can flow to $T^\circ[i, j]$. Figure 11 shows the abstract semantics of key operators.

In the base case, given an input table T_k, each cell in the abstract output symbolically stores the table id and its row/-column indexes ("$T_k[i, j]$"). Because filter, proj, join, left_join and sort do not derive new cells, they simply propagate abstract analysis results of their subqueries to top-level operators. For core analytical operators group, partition and arithmetic, different levels of abstraction are designed based on concreteness of operator to accommodate analysis for queries in different stages in the search process — the abstraction is stronger when more parameters are instantiated. For example, the abstraction of partition is defined as follows:

- *Weak abstraction:* When the query is fully abstract (i.e., partition($q, \square, \square(\square)$)), the analyzer can only confirm that existing columns in T° will be preserved, and every value in the newly generated column can use *any* cell from T°. While it is weak, it can propagate abstraction from T°.
- *Medium-precision abstraction:* When the partition columns $\bar{c}$ are known (partition($q, \bar{c}, \square(\square)$)) but $T^\circ[\bar{c}]$ is abstract, the analyzer can narrow down aggregation columns (to those outside $\bar{c}$). Thus, the new value in row i can refer to any values from any rows in T° that is not in columns $\bar{c}$.

$$[[T_k]]^\circ = \{ [\{``T_k[i, c]"\} \mid c \in \text{columns}(T_k)] \mid i \in [1, \text{rowNum}(T_k)] \}$$

$$[[\text{filter}(q, \square)(\bar{T})]]^\circ = [[q(\bar{T})]]^\circ \qquad [[\text{proj}(q, \square)(\bar{T})]]^\circ = [[q(\bar{T})]]^\circ$$

$$[[\text{join}(q_1, q_2)(\bar{T})]]^\circ = [[q_1(\bar{T})]]^\circ \times [[q_2(\bar{T})]]^\circ$$

$$[[\text{left\_join}(q_1, q_2, \square)(\bar{T})]]^\circ = let \; T_1^\circ = [[q_1(\bar{T})]], \; T_2^\circ = [[q_2(\bar{T})]]^\circ \; in$$
$$(T_1^\circ \times T_2^\circ) \cup \{T_1^\circ[i, *] + \bar{\emptyset} \mid i \in [1, \text{rowNum}(T_1^\circ)]\}$$

$$[[\text{sort}(q, \bar{c}, op)(\bar{T})]]^\circ = \; let \; T^\circ = [[q(\bar{T})]]^\circ \; in$$
$$\{T^\circ[k, *] \mid T^\circ[k, \bar{c}] \; op \; T^\circ[k+1, \bar{c}], k \in [1, \text{rowNum}(T^\circ)]\}$$
$$(when \; T^\circ[\bar{c}] \; is \; concrete)$$

········· *Weak abstraction (no instantiated parameters)* ········

$$[[\text{arithmetic}(q, \square, \square)(\bar{T})]]^\circ = \; let \; T^\circ = [[q(\bar{T})]]^\circ \; in$$
$$\left\{ \begin{array}{l} [T^\circ[i, c] \mid c \in \text{columns}(T^\circ)] \\ + [\bigcup \{T^\circ[i, c] \mid c \in \text{columns}(T^\circ)\}] \end{array} \; \middle| \; i \in [1, \text{rowNum}(T^\circ)] \right\}$$

$$[[\text{partition}(q, \square, \square(\square))(\bar{T})]]^\circ = \; let \; \left\{ \begin{array}{l} T^\circ = [[q(\bar{T})]]^\circ \\ N = \text{rowNum}(T^\circ) \end{array} \right. \; in$$
$$\left\{ \begin{array}{l} [\{T^\circ[i, c]\} \mid c \in \text{columns}(T^\circ)] \\ + [\bigcup \{T^\circ[k, c] \mid k \in [1, N], c \in \text{columns}(T^\circ)\}] \end{array} \; \middle| \; i \in [1, N] \right\}$$

$$[[\text{group}(q, \square, \square(\square))(\bar{T})]]^\circ = \; let \; T^\circ = [[q(\bar{T})]]^\circ, \; N = \text{rowNum}(T^\circ) \; in$$
$$\left\{ \begin{array}{l} [\bigcup\{T^\circ[k, c] \mid k \in [1, N]\} \mid c \in \text{columns}(T^\circ)] \\ + [\bigcup \{T^\circ[k, c] \mid k \in [1, N], c \in \text{columns}(T^\circ)\}] \end{array} \; \middle| \; i \in [1, N] \right\}$$

···· *Medium-precision abstraction (some known parameters)* ····

$$[[\text{arithmetic}(q, \square, \bar{c})(\bar{T})]]^\circ = let \; T^\circ = [[q(\bar{T})]]^\circ, \; in$$
$$\left\{ \begin{array}{l} [T^\circ[i, c] \mid c \in \text{columns}(T^\circ)] \\ + [\bigcup \{T^\circ[i, c] \mid c \in \bar{c}\}] \end{array} \; \middle| \; i \in [1, \text{rowNum}(T^\circ)] \right\}$$

$$[[\text{partition}(q, \bar{c}, \square(\square))(\bar{T})]]^\circ = \; let \; \left\{ \begin{array}{l} T^\circ = [[q(\bar{T})]]^\circ \\ N = \text{rowNum}(T^\circ) \end{array} \right. \; in$$
$$\left\{ \begin{array}{l} [T^\circ[i, c] \mid c \in \text{columns}(T^\circ)] \\ + [\bigcup \{T^\circ[k, c'] \mid k \in [1, N], c' \notin \bar{c}\}] \end{array} \; \middle| \; i \in [1, N] \right\}$$

$$[[\text{group}(q, \bar{c}, \square(\square))(\bar{T})]]^\circ = \; let \; T^\circ = [[q(\bar{T})]]^\circ, \; N = \text{rowNum}(T^\circ) \; in$$
$$\left\{ \begin{array}{l} [\bigcup\{T^\circ[k, c] \mid k \in [1, N]\} \mid c \in \bar{c}] \\ + [\bigcup \{T^\circ[k, c'] \mid k \in [1, N], c' \notin \bar{c}\}] \end{array} \; \middle| \; i \in [1, N] \right\}$$

·············· *Strong abstraction ($T^\circ[\bar{c}]$ is concrete)* ··············

$$[[\text{partition}(q, \bar{c}, \square(\square))(\bar{T})]]^\circ = \; let \; \left\{ \begin{array}{l} T^\circ = [[q(\bar{T})]]^\circ \\ G = \text{extractGroups}([[T^\circ[\bar{c}]]]) \end{array} \right. \; in$$
$$\left\{ \begin{array}{l} [T^\circ[i, c] \mid c \in \text{columns}(T^\circ)] \\ + [\bigcup \{T^\circ[k, c'] \mid i, k \in g \in G, c' \notin \bar{c}\}] \end{array} \; \middle| \; i \in [1, \text{rowNum}(T^\circ)] \right\}$$

$$[[\text{group}(q, \bar{c}, \square(\square))(\bar{T})]]^\circ = \; let \; \left\{ \begin{array}{l} T^\circ = [[q(\bar{T})]]^\circ \\ G = \text{extractGroups}([[T^\circ[\bar{c}]]]) \end{array} \right. \; in$$
$$\left\{ \begin{array}{l} [\bigcup \{T^\circ[k, c] \mid k \in g\} \mid c \in \bar{c}] \\ + [\bigcup \{T^\circ[k, c'] \mid k \in g, c' \notin \bar{c}\}] \end{array} \; \middle| \; g \in G \right\}$$

Figure 11. The abstract semantics for analytical SQL operators that over-approximates cell provenance. The analysis is more precise when more parameters are instantiated.

- *Strong abstraction:* On top of the last case, the subquery $T^\circ[\bar{c}]$ is now concrete, and this lets the analyzer know

how to partition the table T°. This refines the provenance of the newly generated value at row i: it can only include values from rows that are in the group as row i and are outside of columns $\bar{c}$. This is a strong abstraction.

The abstract semantics for arithmetic and group follows the same principle: when the analyzer encounters a concrete subquery q (free of holes), the analyzer will evaluate q using provenance-tracking semantics and pass the concrete output for further abstract reasoning to achieve stronger analysis; otherwise, the rule propagates and builds abstract provenance on top of abstract analysis results from its subqueries.

When an abstract query q cannot pattern match any of the abstract analysis rules, the analysis process stops and returns SAT to the main synthesis algorithm (line 13 in Algorithm 1): in such case q is too abstract to derive a strong analysis results for pruning, and it is the main synthesis loop's responsibility to further instantiate the query.

Provenance Consistency. Given an abstract query q, we use the following criteria to check whether the query is consistent with the input $\bar{T}$ and the user demonstration $\mathcal{E}$:

Definition 3. *(Abstract Provenance Consistency)* Given a table T° (generated by the abstract analyzer) and the user demonstration $\mathcal{E}$ with m rows and n columns, T° is consistent with $\mathcal{E}$ if the following predicate is true (denoted as $\mathcal{E} \lhd T^\circ$):

$$\exists r_1 \ldots r_m, c_1 \ldots c_n. \forall_{j \in [1,m]}^{i \in [1,n]}. \; \text{ref}(\mathcal{E}[i,j]) \subseteq T^\circ[r_i, c_j]$$

Here, the ref function extracts all table references in a cell e (ref also works for $e^\star$ defined in Figure 8):

$$\text{ref}(T_k[i,j]) = \{T_k[i,j]\} \quad \text{ref}(f^\diamond(e_1, \ldots, e_l)) = \bigcup_{i \in [1,l]}\{\text{ref}(e_i)\}$$
$$\text{ref}(const) = \emptyset \quad \text{ref}(f(e_1, \ldots, e_l)) = \bigcup_{i \in [1,l]}\{\text{ref}(e_i)\}$$
$$\text{ref}(group\{e_1, \ldots, e_l\}) = \bigcup_{i \in [1,l]}\{\text{ref}(e_i)\}$$

We next show how the checking criteria can be used to prune infeasible partial programs.

Property 1. *(The relation between $[[\;]]^\star$ and $[[\;]]^\circ$)* Given an abstract query q and input tables $\bar{T}$, let S_q be the set of all possible queries that can be instantiated from q, then:

$$\forall q' \in S_q. \; \exists r_1 \ldots r_m, c_1 \ldots c_n. \; \forall i \in [1,m], j \in [1,n].$$
$$\text{ref}([[q'(\bar{T})]]^\star[i,j]) \subseteq [[q(\bar{T})]]^\circ[r_i, c_j]$$

where $m = \text{rowNum}([[q'(\bar{T})]]^\star), n = \text{colNum}([[q'(\bar{T})]]^\star)$.

This property is witnessed by the comparison between provenance tracking evaluation rules in Figure 9 and abstract evaluation rules in Figure 11: for each row i in $T^\star$, there exists a row i' in T° such that each cell $T^\star[i,j]$ can map to a cell $T^\star[i',k]$ that contains a superset of input cells used by $T^\star[i,j]$. Note that $[[\;]]^\circ$ can produce more columns and rows than $[[\;]]^\star$ because it does not filter or project any newly generated column or rows.

Property 2. *(Pruning Foundation)* Given an abstract query q and input tables $\bar{T}$, let S_q be the set of all possible queries

that can be instantiated from q, then:

$$\mathcal{E} \not\lhd [[q(\bar{T})]]^\circ \implies \nexists q' \in S_q. \mathcal{E} \lhd [[q'(\bar{T})]]^\star$$

Thus, any abstract query q that does not conforms $\mathcal{E} \lhd [[q(\bar{T})]]^\circ$ can be safely pruned.

This property immediately follows Definition 3 and Property 1. Intuitively, if there exist cells in $\mathcal{E}$ that contains references from the inputs $\bar{T}$ but cannot be propagated through the abstract query q with *over-approximation*, then no instantiation of q can enable such propagation. This property builds the foundation of the pruning algorithm.

In this way, the abstract provenance analysis returns the pruning information to the main enumerative search algorithm (Algorithm 1) to direct the search process.

5 Experiments

To evaluate our approach, we implemented the proposed technique as a tool named Sickle and tested in on 80 real world benchmarks consists of end-user analytical SQL questions and industrial database testing benchmarks. We aim to examine the following hypotheses:

- Sickle can solve practical analytical SQL query synthesis tasks from small demonstrations.
- Sickle's new abstraction prunes the search space better than abstractions used in prior systems [11, 39, 40].

We also conducted a user study to compare users' experiences with computation demonstration and I/O examples for analytical task specification.

5.1 Experiment Setup

Sickle is implemented in Python, the user demonstration is provided via spreadsheet; however, with some additional engineering effort, Sickle can work with a drag-and-drop interface [41] (as illustrated in Section 2) for smoother experience. In the synthesis process, Sickle enumerates join predicates (used in join and left_join) based on primary and foreign keys of the table to avoid generating unnatural predicates like "T1 **Join** T2 **On** T1.id < T2.age"; for the same reason, Sickle does not invent new constants for filter and considers only those provided by the user. Sickle ranks synthesized queries based on query size.

Benchmarks. We evaluate Sickle on 80 benchmarks, including 60 collected from analytical SQL online tutorials/forums, and 20 tasks from TPC-DS decision support benchmark [23] — an industry standard benchmark suite for testing commercial database systems' analytical SQL features:

- From online forums and tutorials, we explored hundreds of posts and select analytical tasks with input data available. We obtained 43 easier tasks that require 2-3 operators from Figure 7 to solve and 17 tasks that require 4-7 operators.

- When collecting benchmarks from TPC-DS, we extract tasks by (1) isolating table view definitions (many TPC-DS tests are long sequences of tasks with one building on top of the result of another), and (2) decomposing big union queries into smaller tasks (some queries are unions of many sub-tasks). This gives us 20 benchmarks each focus on one analytical task; all requires 4-7 operators.

For each benchmark with input $\bar{T}_{raw}$ and the ground truth query q_{gt}, we programmatically generate small computation demonstrations using the following procedure: (1) for input with > 20 rows, we sample 20 rows from the input table and use the sampled data $\bar{T}$ as the synthesis task input, (2) evaluate $T^\star = [[q_{gt}(\bar{T})]]^\star$; (3) randomly sample 2 rows from $T^\star$ as $T^\star_{partial}$ and permute orders of arguments in commutative functions; (4) for expressions in $T^\star_{partial}$ with more than four values, replace it with an incomplete expression that contains at most four values together with a $\diamond$ (for omitted parameters). This generates $\mathcal{E}$, a partial output table with incomplete expressions, to simulate user demonstrations that is suitable for systematic algorithm performance testing. Each benchmark is a tuple $(\bar{T}, \mathcal{E}, q_{gt})$. In total, our benchmark set contains 43 easier tasks (1-3 operators) and 37 harder tasks (4-6 operators); among them, 24 requires join , 51 tasks requires partition-aggregation and 32 requires group-aggregation. Example benchmarks can be found in the appendix.

Baselines. To evaluate our algorithm, especially the provenance abstraction design, we compare Sickle with two state-of-arts abstraction-based pruning techniques successfully used in relational query synthesis:

- *Morpheus's type abstraction [12]:* This abstraction tracks high-level table shape information (row/column/group numbers). This technique is widely adopted in synthesizing list and table transformation programs.
- *Scythe's value abstraction [39]:* This abstraction keeps track of concrete values flow through a partial program to over-approximate its behavior. This technique is used in the state-of-the-art SQL query and visualization synthesis.

Because neither of the baselines supports analytical operators, we reimplement them by extending their abstract semantics: (1) in type abstraction rules, we extend the abstract semantics to infer the most precise table shape and group number for partition and aggregation rules, and (2) in value abstraction rules, we keep track of all known values (e.g., values from the grouping columns) for analytical operators but ignore unknown values (e.g., values from the aggregation column). Other abstraction rules for both baselines are the same as those in the original paper. We then run the evaluation using the same enumerative search framework Sickle used, which keeps the search order the same for all abstractions.
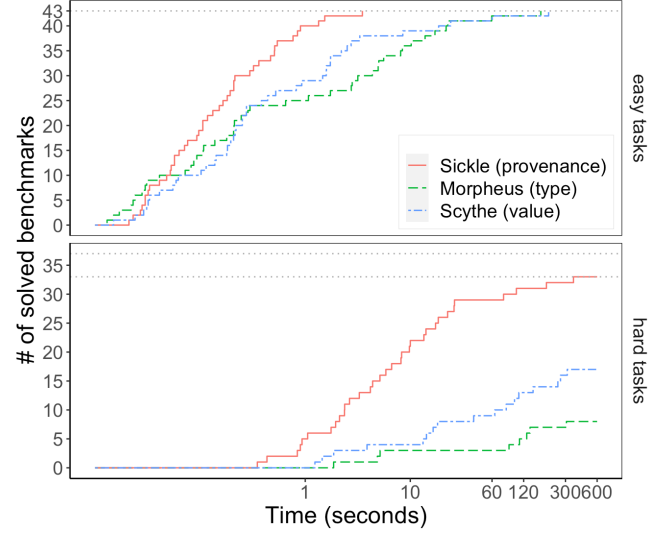

Figure 12. The number of benchmarks each technique can solve within given time limit for each benchmark.

5.2 Study 1: Synthesis Efficiency

Efficiency Comparison. We first test Sickle's performance on practical analytical SQL tasks compared to baseline techniques. For each benchmark $(\bar{T}, \mathcal{E}, q_{gt})$, we run Sickle and two baselines with a timeout of 600 seconds. The synthesizer runs until the correct query q_{gt} is found. We record (1) time each technique takes to solve the tasks, and (2) the number of consistent queries encountered by Sickle (including the correct one q_{gt}). Sickle ranks consistent queries based on the order in which their are found in the search process. Figure 13 shows the results: each plot shows the number of benchmarks (y-axis) that can be solved within the given time limits (x-axis) by each baseline; results for easier and harder benchmarks are plotted separately.

Observation #1: Sickle can efficiently solve practical analytical SQL tasks from small demonstrations. Sickle has significant advantage over baselines on harder benchmarks.

As shown in Figure 12, Sickle in total solved 76 out of 80 benchmarks with a mean solving time of 12.8 seconds (Sickle solved all 43 easier tasks and 33 out of 37 harder tasks). In these benchmarks, the average user demonstration size is 9 cells (the number would be 50 if full output examples are required from the user). In comparison:

- On the easy benchmark suite, while both baselines can also solve all 43 tasks, Sickle is significantly faster: Sickle's average solving time is 0.4 seconds, but type and value abstraction can take up to 8.0 and 8.6 seconds,
- Out of the 37 harder benchmarks, Sickle solved most (33 benchmarks), and type and value abstractions solved only 8 and 17 benchmarks. Sickle is also faster on tasks all techniques can solve. Sickle is on average 22.3× faster than value abstraction and 69.7× faster than type abstraction.
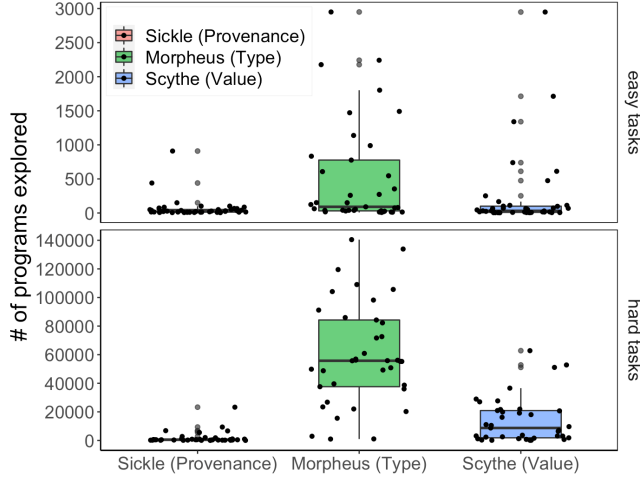
Figure 13. Distributions of the number of queries explored by different techniques on easy (top) and hard (bottom) tasks.

The 4 cases Sickle cannot solve are hard tasks from the TPC-DS. Upon closer inspection, cases that Sickle cannot solve or are slow have the following traits: (1) the input data has many columns, or the task require `join` (they dramatically increase the parameter space), or (2) the solution contains many *nested* operators (this makes analysis less precise).

Pruning Analysis. To understand how the provenance abstraction helps Sickle to efficiently solve these practical tasks, we log and compare the number of queries (both partial and concrete queries) Sickle and both baselines explored during the synthesis process. Figure 13 shows the results (top for easy tasks and bottom for hard ones). For each suite, the box plot shows the distribution of the number of queries each technique encountered when solving these benchmarks (or before timeout). We made the following observation:

Observation #2: Provenance abstraction has small advantages over type and value abstractions on easier tasks because of relatively small query space to explore; but its advantage over others is significant on harder tasks with large search space.

On easier tasks (Figure 13 top), despite Sickle's provenance abstraction prunes the search space better, its advantage is small because these easier benchmarks require only exploring a small search space with a small number of programs inside. However, on harder tasks, the synthesizer needs to find the solution from a large search space with $100,000 - 2,000,000$ queries, and this makes provenance abstraction's pruning power shine. As shown in Figure 13, Sickle explores only 917 queries on average before finding the correct query; this is much less than type and value abstractions, which need to explore 31,371 and 6,837 queries on average (also with higher variance).

In general, while provenance abstraction has higher reasoning overhead compared to type or value abstractions, its stronger pruning power makes it significantly outperform

other techniques on challenging benchmarks as much less programs in the search space are visited. The key strength of Sickle is its ability to track fine-grained provenance information through partial programs, and we envision computation demonstration and provenance abstraction to be applied to synthesis tasks in other domains rich of computation.

We also examined the ranking of queries synthesized by Sickle to examine computation demonstration's ambiguity issues. There are 71 benchmarks with the correct query ranked at top 1, 4 cases ranked between 2 to 9, and 1 case ranked at >10. We noticed that computation traces and cell references are often less ambiguous than concrete value despite the smaller demonstration size; and for the case where the correct query is ranked > 10, we found the issue is that the input data contains only one group of data — this makes the synthesizer unable to easily generalize to multiple groups as required in the solution. This motivates that future disambiguation model could focus on seeking for more representative inputs from the user (rather than asking for more concrete/complete output demonstrations).

5.3 Study 2: User Experience

To understand users' experiences working with computation demonstration, we conducted a within-group user study with 13 participants on 6 analytical tasks. In the study, we focus on specification creation: we compare *user efficiency* and *user experience* using computation demonstration and output-examples for analytical task illustration. We only require the participant to understand basic computation terminologies (e.g., cumsum, rank). The 6 analysis tasks come from our benchmarks, and their complexity and input table size vary with inputs containing 10-30 rows and require 1-3 analytical operators to solve, and they are split into groups based on query complexity and table size: 3 easy tasks (1-2 analytical steps, small table), and 3 hard tasks (2-3 analytical steps, big table). Each task contains (1) a description section of the input table and the task, (2) input table, and (3) output table schema (with no information about the output content).

For each task, we assign a demonstration method that the participant needs to use: (1) output example with concrete values, (2) full computation expressions, and (3) computation demonstration with partial expressions. The participant needs to fill out three rows in the output table using the assigned specification type in a spreadsheet environment. We counterbalance task order and specification order with the Latin square design. Each study session contains 10 minutes tutorial time, 30 minutes for solving the 6 tasks, and 20 minutes for interviews. We recorded the time each participant spent on creating the demonstration and analyzed results with Wilcoxon rank sum test; qualitative results are analyzed via inductive content analysis approach. Our study design intentionally hides the complexity of disambiguation and query understanding tasks that are orthogonal to our

main contribution. This let us allocate more time to observe user behaviors on more tasks. More study details are in the supplementary material.

Quantitative analysis. In the user study, 72 out of 78 participants' solutions are correct. On the three easy tasks (1-2 analytical steps, inputs with < 15 rows), participants are more efficient using examples (on average 122s) compared to expressions (on average 170s) or partial expressions (on average 168s), but the difference is insignificant ($p > 0.1$). On two of the three hard tasks, users are significantly more efficient ($p < 0.05$) using expressions (on average 226s) over examples (on average 300s). Surprisingly, on one hard task — ranking countries based on their average sales using the Rank function — the participants are significantly more efficient using examples (on average 175s) over expressions (on average 286s) or partial expressions (on average 209s) with $p < 0.05$, as we explain in our qualitative analysis.

Qualitative analysis. We noticed that most participants prefer expressions over examples because they enhance user confidence: participants find examples *"tedious and time consuming"* (P6) and *"not show how you derive the data"* (P9) but find computation demonstration make them *"see what exactly I am doing"* (P1) and thus *"more confident"* (P3, P7, P10). The participants also noted that they sometimes spend more time on creating expressions because they need to typeset the answers, and we envision this can be resolved with a drag-and-drop interface design in future. Second, participants mentioned that partial expressions are vital when the input is big (>20 rows). For example, *"smaller expressions can be used more easily than compute the final value, and it can show the pattern"* (P5), and *"I do not need to look many data or do computation on my own"* (P10). However, some participants think partial expression makes them *"less confident (than full expressions), as it depends on the user's experience level to make right omission"* (P13). Finally, for the ranking task where participants perform significantly worse using expressions, participants prefer manually counting over *"using an expression and go over all the data"*, because they (1) cannot easily sort the data by groups in the spreadsheet to create the full expression and (2) find omitting parameters tricky for rank. Currently, this is a limitation of computation demonstration. We envision a mixed (example and computation demonstration) approach to make analytical tasks easy.

6 Related Work

Relational Query Synthesis. Due to the importance of relational queries in the database industry and the programming challenges associated with them, many relational query synthesizers have been developed [29, 31, 36, 37, 39, 47]. Among them, Scythe [39] supports standard SQL queries with subqueries and aggregations; EGS [36], ProSynth [29] and Alps [31] are datalog query synthesizers that synthesizes

datalog queries using syntax and provenance guided search from input-output examples; Morpheus [12] and Viser [40] are table transformation synthesizers for data wrangling.

Sickle's focus is the computation-rich *analytical SQL*. Analytical SQL are not expressible in logic-based synthesizers [36] that requires relational queries to be "pure" (free of computations); algorithms for standard SQL [39, 40] are insufficient due to lack of support for analytical operators (Section 5). As Sickle focuses on analytical computations, Sickle is not specially designed for data wrangling [12] or knowledge base querying tasks [29]; there is an opportunity to combine Sickle with Morpheus [13] or ProSynth [29] for tasks requiring all types of relational query features. Sickle can also benefit from PATSQL's schema inference technique [35] to better scale up to larger input table by prioritizing search for columns that are type-consistent with the user demonstration.

Programming-by-demonstration Tools. Programming-by-example/demonstration (PBE/PBD) tools have been successfully adopted in many end user programming scenarios [3, 5, 12, 15, 20, 27, 28, 32, 40, 42]. PBD tools like Helena [4, 5] and FlashExtract [20]) generalize users' data extraction actions into scripts. Sickle's specification, computation demonstration, is also a type of PBD specification, but it differs from others: existing PBD tools synthesize scripts by "loopifying" the user demonstration, but our approach synthesizes a *declarative query* from the demonstration. For example, the user's expression sketch in DACE [43] can be directly used as a component of the target program, while a partial expression in Sickle is only a cell-level expression generated by evaluating the target analytical query. Thus, this domain challenge makes Sickle unable to directly reuse the user demonstration as program components [7, 43]. Gauss [3] and Sickle have similar specification design that allows the user to provide computation expressions as part of the output demonstration. Because Sickle is designed to solve the analytical SQL synthesis problems where nested computation are more common (Gauss focuses on data wrangling problems), Sickle makes the following different design choices to improve user experience: (1) Sickle permits omitted parameters in expressions, and (2) Sickle does not enforce the users' expression syntactically same as the expression appear the output from the correct query. As a result, Gauss's computation graph abstraction is too strict to apply, and Sickle's choice of data provenance abstraction is more general and effectively prunes inconsistent queries during the search process. We also envision our specification used in other computation-rich domains [1, 33].

Besides, many new interaction models [14, 16, 21, 25, 41] are proposed make PBE and PBD tools easier-to-use. As our paper focuses on the synthesis specification and algorithm design, its interaction model is basic. As suggested by our user study, Sickle could work with a drag-and-drop

based interface [41] for easier demonstration creation dialog-based [16, 21] interface for intent clarification.

Abstraction-based Synthesis Algorithms. Abstraction-based program synthesis algorithms have been widely used in synthesis of structurally rich programs (e.g., loopy programs [19], list programs [27, 46], tables programs [11, 39, 40], regex [44]). The core technique is to design a language abstraction [8] for reasoning behaviors of partial programs and pruning ones unable to reach the goal. Sᴀᴄᴋʟᴇ contributes to this line of work by introducing provenance abstraction that can capture fine-grained provenance information for computation rich programs that type [11, 27] or value-based [39, 40] abstractions cannot. While Sᴀᴄᴋʟᴇ currently builds on top of the enumerative search framework [11, 26, 38], the abstraction can also work in solver-based [30, 34] or learning-based [2, 9, 10] frameworks for pruning.

7 Conclusion

We proposed a new synthesis-based approach to make analytical SQL more accessible. We designed (1) a new synthesis specification, *programming by computation demonstration*, that allows the user to demonstrate the task using partial cell-level computation trace, and (2) a new abstraction-based algorithm that leverages *abstract data provenance* to prune infeasible search space more effectively. We implemented and tested our approach on 80 real-world analytical SQL tasks. Results show that Sᴀᴄᴋʟᴇ can efficiently solve 76 tasks from small demonstrations with a 600 second timeout. In the future, we envision Sᴀᴄᴋʟᴇ can work with novel user interaction models and ranking strategies to make demonstration crafting and solution disambiguation easier.

Acknowledgments

References

[1] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Trans. Graph.* 38, 6 (2019), 204:1–204:13. https://doi.org/10.1145/3355089.3356549

[2] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 168:1–168:27. https://doi.org/10.1145/3360594

[3] Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: program synthesis by reasoning over graphs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485511

[4] Sarah Chasins and Rastislav Bodík. 2017. Skip blocks: reusing execution history to accelerate web scripts. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 51:1–51:28. https://doi.org/10.1145/3133875

[5] Sarah E. Chasins, Maria Mueller, and Rastislav Bodík. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology, UIST 2018, Berlin, Germany, October 14-17, 2018*, Patrick Baudisch, Albrecht Schmidt, and Andy Wilson (Eds.). ACM, 963–975. https://doi.org/10.1145/3242587.3242661

[6] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM Sigmod record* 26, 1 (1997), 65–74.

[7] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. 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*. 3–14.

[8] Patrick Cousot and Radhia Cousot. 1977. 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*. 238–252.

[9] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. 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*. 990–998. http://proceedings.mlr.press/v70/devlin17a.html

[10] Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 1638–1645. https://doi.org/10.24963/ijcai.2017/227

[11] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. 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*. 420–435.

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. 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*. 422–436. https://doi.org/10.1145/3062341.3062351

[13] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proc. Symposium on Principles of Programming Languages*. ACM, 599–612.

[14] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 614–626. https://doi.org/10.1145/3379337.3415869

[15] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. https://doi.org/10.1145/1926385.1926423

[16] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1143–1158. https://doi.org/10.1145/3385412.3386025

[17] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. 2011. Research Directions in Data Wrangling: Visualizations and Transformations for Usable and Credible Data. *Information Visualization Journal* 10, 4 (2011), 271–288.

[18] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. In *IEEE Visual Analytics Science & Technology (VAST)*. http://idl.cs. washington.edu/papers/enterprise-analysis-interviews

[19] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

[20] Vu Le and Sumit Gulwani. 2014. FlashExtract: A framework for data extraction by examples. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 542–553.

[21] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, Celine Latulipe, Bjoern Hartmann, and Tovi Grossman (Eds.). ACM, 291–301. https://doi.org/10.1145/2807442.2807459

[22] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *PACMPL* 2, POPL (2018), 1:1–1:30. https://doi.org/10.1145/3158089

[23] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1049–1058. http://dl.acm.org/citation.cfm?id=1164217

[24] Aditya Parameswaran. 2019. Enabling data science for the majority. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2309–2322.

[25] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a read-eval-synth loop. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 159:1–159:30. https://doi.org/10.1145/3428227

[26] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. 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*. 297–310. https://doi.org/10.1145/2872362.2872387

[27] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. 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*. 522–538. https://doi.org/10.1145/2908080.2908093

[28] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 107–126.

[29] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 62:1–62:27. https://doi.org/10.1145/3371130

[30] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019,*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11562. Springer, 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

[31] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. 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*. 515–527.

[32] Rishabh Singh and Sumit Gulwani. 2012. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*. Springer, 634–651.

[33] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 326–340. https://doi.org/10.1145/2908080.2908102

[34] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis.. In *Proc. Asian Symposium on Programming Languages and Systems*. Springer, 4–13.

[35] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. 2021. PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns. *Proc. VLDB Endow.* 14, 11 (2021), 1937–1949. http://www.vldb.org/pvldb/vol14/p1937-takenouchi.pdf

[36] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-guided synthesis of relational queries. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1110–1125.

[37] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. 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*. 535–548. https://doi.org/10.1145/1559845.1559902

[38] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. *SIGPLAN Not.* 48, 6 (June 2013), 287–296. https://doi.org/10.1145/2499370.2462174

[39] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. 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*. 452–466. https://doi.org/10.1145/3062341.3062365

[40] Chenglong Wang, Yu Feng, Rastislav Bodík, Alvin Cheung, and Isil Dillig. 2020. Visualization by example. *Proc. ACM Program. Lang.* 4, POPL (2020), 49:1–49:28. https://doi.org/10.1145/3371117

[41] Chenglong Wang, Yu Feng, Rastislav Bodík, Isil Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*, Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, and Steven Mark Drucker (Eds.). ACM, 106:1–106:15. https://doi.org/10.1145/3411764.3445249

[42] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts using Finite Tree Automata. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 62:1–62:26.

[43] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *PACMPL* 1, OOPSLA (2017), 62:1–62:26. https://doi.org/10.1145/3133886

[44] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

[45] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[46] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 508–521.

[47] Moshé M Zloof. 1975. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*. ACM, 431–438.