GUARDRAIL: Solidifying ML-Integrated SQL Queries with Automated Integrity Constraint Synthesis

Pingchuan Ma
The Hong Kong University of Science and Technology

Zhaoyu Wang
The Hong Kong University of Science and Technology

Zhenlan Ji
The Hong Kong University of Science and Technology

Zongjie Li
The Hong Kong University of Science and Technology

Ao Sun
The Hong Kong University of Science and Technology

Shuai Wang
The Hong Kong University of Science and Technology

ABSTRACT

To date, machine learning (ML) models have become an integral part of modern data management systems to enhance data processing capabilities. As a result, ML-integrated SQL queries have been widely adopted in various data science applications. However, the credibility of these queries is often compromised by the presence of noisy data. This paper advocates the usage of *integrity constraints* in detecting and rectifying errors in the data before feeding them as inputs of ML models.

While the relational schema entails some coarse-grained constraints, they are often insufficient to capture the nuances of the data. In this paper, we propose a novel focus on discovering integrity constraints with program synthesis techniques from *noisy and opaque* input-output examples. To support the program synthesis task, we introduce a domain-specific language (DSL) and propose a sketch-based synthesis algorithm over the DSL. We demonstrate the effectiveness of our approach on 48 ML-integrated SQL queries using 12 real-world datasets. Evaluation shows that our approach can effectively synthesize integrity constraints using noisy data, and also solidify queries with an average reduction of 87% in the error rates.

PVLDB Reference Format:
Pingchuan Ma, Zhaoyu Wang, Zhenlan Ji, Zongjie Li, Ao Sun, and Shuai Wang. GUARDRAIL: Solidifying ML-Integrated SQL Queries with Automated Integrity Constraint Synthesis. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/prog-syn-126A.

1 INTRODUCTION

Machine learning (ML) models have become an integral part of modern database systems (e.g., the `PREDICT` clause in Microsoft SQL Server, Amazon Aurora Machine Learning and Google BigQuery ML). ML-integrated SQL queries, also known as Query 2.0 [40], have recently emerged as a promising paradigm to enhance the expressiveness of traditional SQL queries [20, 22, 24, 40]. A practical example of this can be seen in the following healthcare sector.

EXAMPLE 1.1. *Bob, a hospital administrator, aims to determine the average risk of patients likely to suffer from dyspnea (i.e., shortness of breath) across different floors within the hospital.*[1] *This information is crucial for optimizing the allocation of medical resources. To achieve this, Bob purchases a proprietary ML model from a third-party vendor and writes an ML-integrated SQL query, as shown in "ML-integrated SQL Query" in Fig. 1. Bob then uses the results, presented in "**Query Outcome without Guardrail**" in Fig. 1, to facilitate his decisions. However, Bob is uncertain about the credibility of these predictions.*

ML models are often built in-house and trained on high-quality data, whereas the serving data in the wild is often noisy. These data quality issues can undermine the reliability of these data-driven systems [12]. To continue with the healthcare example, the prediction can be compromised by noisy tuples in the database (e.g., faulty X-ray results or wrong disease codes). As a result, the errors in data can propagate to query outcomes. The biased query outcomes can further mislead the whole decision-making process and undermine the reliability of the final decision.

Recent studies have shown the power of data constraints to safeguard the ML models against noisy data [12]. In general, these methods first learn a set of constraints from the data and then use the constraints to alert the users of potential errors in the data. Indeed, in the security community, it has been recognized that certain constraints (e.g., value boundary) can be used in hardening the ML inference and therefore sanitizing potentially "out-of-bound" ML model inputs [6].

Inspired by these works, we advocate synthesizing and employing data constraints to solidify these ML-integrated queries. Constraints are first synthesized from potentially noisy data. Then, during the query execution process (i.e., the inference phase of the ML model), we anticipate alerting the users of potential errors by checking input tuples to the ML model against the data constraints. In this way, we can avoid the propagation of errors in the data to the final decision and even rectify some of the errors using the constraints (examples given soon). Nevertheless, we face several challenges in this endeavor.

Challenge: Tuple-Level Constraints for Discrete Data. To solidify the reliability of data-driven systems, some data profiling techniques have been proposed to discover various constraints from the data using statistical or ML techniques [12, 41, 44]. However, as

[1]The "Hospital Database Schema" is extended from the "asia" dataset used in [44].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

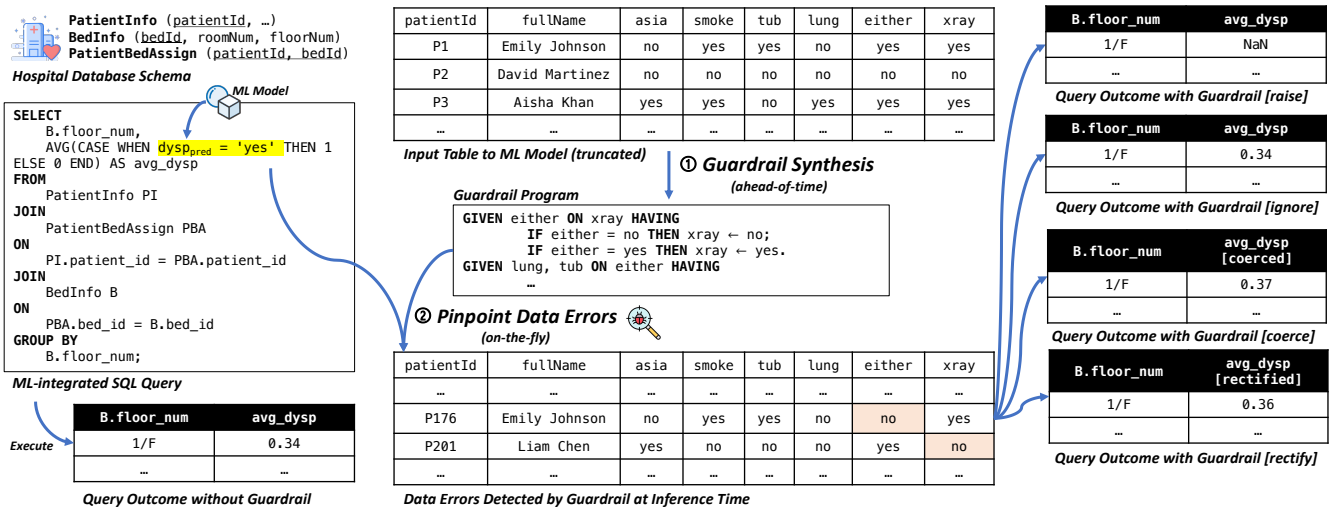

Figure 1: The pipeline of GUARDRAIL.

noted in [12], the majority of existing techniques focus on detecting *dataset-level* errors, such as inter-column correlations [41], functional dependencies [29] or denial constraints [8]. These techniques are usually non-parametric and are not able to detect *tuple-level* errors, such as the errors in the `either` attribute in Fig. 1. The most relevant work to ours is the conformance constraints [12], which are able to detect tuple-level arithmetic errors. Despite their effectiveness, the conformance constraints are only applicable to numerical data with linear relationships and fail short in modeling categorical data, a prevalent data type in real-life data science scenarios (i.e., ML on tabular data) that intensively use categorical features. In contrast, we focus on synthesizing constraints for categorical data to complement the existing solutions.

Challenge: Succinctness. As is frequently advocated in existing solutions [8, 12, 41], it is desirable to provide a *concise* set of constraints since verbose constraints are not only hard to interpret but also prone to overfitting. This desideratum in turn requires an inference system (e.g., the well-known Armstrong axioms for functional dependencies [3]) to reason about the redundancy or implication between constraints. However, it is unclear how to architect such an inference system for tuple-level constraints on categorical data.

Opportunity: Programmatically-Generated Data. On the other end of the spectrum, it has been recently recognized that a considerable proportion of data are *programmatically generated* [25, 42]. In this regard, a *functional dependency* (FD) inherently entails an unseen program that generates the dependent attribute from the determinant attributes [3] (e.g., state is functionally dependent on city). Therefore, when we are able to programmatically represent the underlying data-generating process (DGP) for the whole data, the program itself codifies a succinct set of integrity constraints that can be used to detect and even rectify the errors in the data.

Programming-by-"Noisy-and-Opaque"-Examples. In the programming language (PL) community, there have been great interests in synthesizing programs from input-output examples (i.e., programming-by-examples, PBE) [16]. In particular, recent advances in program synthesis have enabled the synthesis from *noisy* input-output examples [17–19]. Therefore, the procedure of inferring the DGP from the data can be viewed as program synthesis from noisy input-output examples. However, in contrast to the standard PBE scheme, where the input and output is explicitly specified, in our setting, the examples are not only noisy but also *opaque*. In other words, it is unclear what the input and output (corresponding to different attributes in the data) of the DGP are. The opaque nature of the DGP makes it challenging to synthesize the DGP from the data. Recalling the succinctness desideratum, it is also unclear to what extent the DGP can be succinctly synthesized.

Our Solution. In this paper, we propose, GUARDRAIL, a novel approach to synthesizing integrity constraints from noisy and opaque data, and focus on a timely and demanding application of these constraints, namely, safeguarding ML-integrated SQL queries. We formulate a DSL (domain-specific language) to capture the deterministic DGP. We employ a sketch-based synthesis procedure, which initially derives a high-level program sketch of the DGP. This sketch is then concretized into a full program. To identify the sketch, our solution introduces the integration of probabilistic graphical models (PGMs) and Markov equivalence classes (MECs) for sketch synthesis. By statistically interpreting the data-generating process, we establish a correspondence between the sketches of interest and the learned PGM structures. This enables us to synthesize program sketches that are both expressive and succinct.

EXAMPLE 1.2. *Continuing from Example 1.1, Bob uses* GUARDRAIL *to synthesize the integrity constraint from data ahead-of-time. Then, each time the ML model is invoked, input tuples are checked against the constraints. If the input tuple violates the constraints, in accordance with popular data science practices,* GUARDRAIL *offers three error handling schemes:* `raise`, `ignore`, `coerce`. *With the* `raise` *scheme,* GUARDRAIL *raises an error during the query execution process and replaces the erroneous value with* `NaN`. *With the* `ignore` *scheme,* GUARDRAIL *simply ignores the erroneous tuple. With the* `coerce` *scheme,* GUARDRAIL *replaces the erroneous value with* `NaN` *and continues the query execution process, where aggregations are processed with the* `dropna` *operator. With the synthesized integrity constraints,* GUARDRAIL *can go one step further and rectify the erroneous tuple. With* GUARDRAIL*, instead of blindly trusting the ML-integrated SQL*

query, Bob is aware of potential errors in the data and can make more informed decisions, as shown in Fig. 1.

Contributions. In summary, our contributions are as follows:

- **Constraint Discovery via Program Synthesis.** To solidify ML-integrated query systems, we advocate a novel focus on discovering data integrity constraints with program synthesis techniques. We introduce the concept of data-generating program (DGP) for modeling underlying data and precisely define the problem of succinct program synthesis under this setting where the input-output examples are noisy and opaque.
- **DSL and Sketch-based Synthesis.** To form the technical pipeline, we design GUARDRAIL, a DSL to describe the discrete DGP. We also design a sketch-based synthesis procedure to synthesize the DGP from the data. With canonical assumptions, we show that our solution guarantees the succinctness of the synthesized DGP.
- **Application to ML-Integrated SQL Queries.** We show that constraints derived from GUARDRAIL programs are effective in safeguarding ML-integrated SQL queries, an emerging paradigm for data-driven decision, against data errors. GUARDRAIL not only detects the errors but also rectifies them with an average reduction of 87% in the error rate.

Open-source Artifact. Our codebase is available at [2]. The artifact includes the source code of GUARDRAIL and the datasets used in the evaluation. We will make our prototype available on Python Package Index (PyPI) and provide detailed documentation to facilitate the usability of our work.

2 FORMULATING A DSL FOR DGP

In this section, we introduce the class of DGP that we consider in this work and present a DSL to describe them.

2.1 Data-generating Process (DGP)

In statistics, it is common to model the data as a random vector X that is generated from an unseen DGP. In this regard, the DGP can be viewed as a (probabilistic) program that takes some exogenous inputs and generates the endogenous outputs [31]. The exogenous and endogenous attributes (together with other white noise) constitute the data. Hence, each tuple in the dataset can be viewed as a realization of the DGP. In general, DGPs exist in many forms, can be arbitrarily complex, and are often stochastic. In the context of error detection, we are interested in the deterministic part of the DGP on discrete data, which allows us to confidently pinpoint the errors in the data.

Discrete DGP. Many string-manipulating programs can be interpreted as discrete DGPs [10, 15]. For example, the program `concat` takes two strings as inputs and returns the concatenation of the two strings. Here, the two input strings are the exogenous inputs, and the concatenation of the two strings is the endogenous attribute. In many data science applications, the data is often categorical. For example, consider a relation R with attributes `PostalCode`, `City`, `State`, and `Country`. Here, `PostalCode` are the inputs, and `City` is the output. Given the state $\sigma = \{\texttt{PostalCode} \coloneqq \texttt{80201}\}$, the program p derived from the DGP assigns the value `Denver` to the variable `City` and yields the updated state $\sigma' = \{\texttt{PostalCode} \coloneqq \texttt{80201}, \texttt{City} \coloneqq \texttt{Denver}\}$. Formally, $[\![p]\!]_\sigma = \sigma'$, where $[\![\cdot]\!]_\sigma$ denotes the evaluation of the program with respect to the state σ. Moreover, the data itself does not contain any information about which column(s) are the inputs and which column(s) are the outputs.

DGP For Error Detection. When the (deterministic) DGP is known, we can use it to detect errors in the data. Specifically, let p be the program describing the DGP, and t be a tuple in the dataset (equivalently, the state of the program). Then, we can check whether the endogenous attributes in t is generated by p given the exogenous attributes in t. Formally, we want to check the assertion below.

$$\forall t \in D, [\![p]\!]_t = t' \models t = t' \tag{1}$$

If it does not hold, then we can conclude that there is an error in the data. Continue with the above example. Let an erroneous tuple $t = \{\texttt{PostalCode} \coloneqq \texttt{80201}, \texttt{City} \coloneqq \texttt{gibbon}\}$ where "Denver" is corrupted as some random strings like "gibbon". Then, by executing the program p, we can obtain an updated state $t' = \{\texttt{PostalCode} \coloneqq \texttt{80201}, \texttt{City} \coloneqq \texttt{Denver}\}$ and it is obvious that $t \neq t'$. Hence, we can conclude that there is an error in this tuple.

2.2 Domain-Specific Language (DSL)

We now present a DSL to describe the discrete DGP. Our later synthesis procedure will synthesize a program in this DSL, and the synthesized program (denoting a set of conformance constraints) can be used to detect errors in the data.

Syntax. The syntax of our DSL for the DGP, as illustrated in Fig. 2, is structured into key components. $p \in$ Prog represents the whole program that consists of a series of statements. Each statement $s \in$ Stmt, represents the core program structure linking determinants to dependents based on specific conditions; the `GIVEN` clause and the `ON` clause outline determinant and dependent attributes of the generating process for the dependent attribute. Then, each statement also encapsulates a list of branches in the `HAVING` clause; $b \in$ Branch offers conditional assignments for dependent attributes; and $c \in$ Condition defines the criteria for branch validity. $a \in$ Attribute and $l \in$ Literal depict attributes and possible attribute values, respectively.

Semantics. In general, a $p \in$ Prog describes the whole DGP over a dataset while each statement $s \in$ Stmt describes the DGP for a specific dependent attribute. In our context, p is applied to an individual tuple in the dataset. Hence, without loss of clarity and for simplicity, given a tuple $t \in D$, we also use t to denote the input state for a program. Here, variables in t are the attributes in the dataset, and their values are the values of the attributes in the tuple. Thus, given a tuple $t \in D$ and a $p \in$ Prog, we use $[\![p]\!]_t$ to denote the execution of p on the input state t. We present the denotational semantics of the DSL in Fig. 2. The semantics of the DSL is straightforward, and therefore, we omit the elaboration of the semantics of the DSL here.

Validation Rule. We clarify that the grammar of the DSL is too permissive to describe a valid DGP. Thus, to ensure that the $p \in$ Prog is well-defined, we provide two validation rules, as shown in Fig. 3. In particular, the first rule ensures that the assignment attributes in all branches within a statement are in line with the attribute specified in the preceding `ON` clause (i.e., the dependent attribute of the statement). The second rule ensures that the condition in the `IF` clause is derived from the determinant attributes of the statement. Here, $\text{comb}(a_1, \cdots, a_i)$ denotes the Cartesian product of the attribute

Syntax

$$p \in \text{Prog} := s^*$$
$$s \in \text{Stmt} := \texttt{GIVEN}\ a^+\ \texttt{ON}\ a\ \texttt{HAVING}\ b^+$$
$$b \in \text{Branch} := \texttt{IF}\ c\ \texttt{THEN}\ a \leftarrow l$$
$$c \in \text{Condition} := a = l \mid c\ \texttt{AND}\ c$$
$$a \in \text{Attribute} := \{a_1, \cdots, a_n\}$$
$$l \in \text{Literal} := \text{String} \cup \text{Number} \cup \text{Boolean}$$

Semantics

$$[\![l]\!] := l$$
$$[\![a]\!]_\sigma := \sigma(a)$$
$$[\![a = l]\!]_\sigma := \begin{cases} \text{true} & \text{if } [\![a]\!]_\sigma = [\![l]\!] \\ \text{false} & \text{otherwise} \end{cases}$$
$$[\![c_1\ \texttt{AND}\ c_2]\!]_\sigma := [\![c_1]\!]_\sigma \wedge [\![c_2]\!]_\sigma$$
$$[\![\texttt{IF}\ c\ \texttt{THEN}\ a \leftarrow l]\!]_\sigma := \begin{cases} \sigma[a \mapsto l] & \text{if } [\![c]\!]_\sigma = \text{true} \\ \text{skip} & \text{otherwise} \end{cases}$$
$$[\![\texttt{GIVEN}\ a^+\ \texttt{ON}\ a_j\ \texttt{HAVING}\ b^+]\!]_\sigma := [\![a_j]\!]_{[\![b^+]\!]_\sigma}$$
$$[\![s_1\ s_2]\!]_\sigma := [\![s_2]\!]_{[\![s_1]\!]_\sigma}$$

Example of Program under the DSL

```
GIVEN PostalCode ON City HAVING
    IF PostalCode = 80201 THEN City ← Denver
    ...
    IF PostalCode = 98101 THEN City ← Seattle
```

Figure 2: Formulating our DSL for the DGP and an example.

values of $a_1, \cdots, a_i$, encapsulating the warranted conditions for the IF clause.

Branch-level Loss Function. Given a dataset D and a branch b, the loss of b on D is denoted by $L(b, D)$. We adopt the 0/1 loss function as defined in [17]:

$$L(b, D) = \sum_{t \in D^b} \begin{cases} 1 & \text{if } [\![b]\!]_t \neq t \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where D^b is the set of tuples in D that satisfy the condition of b. In the remainder of the paper, we use D^s and D^p to denote the union of D^b for all $b \in s$ and p, where s is a statement and p is a DSL program, respectively. This implies that $L(b, D)$ represents the count of tuples in D that do not align with the specifications of the branch. A lower loss indicates a higher quality of b. While there exist other advanced loss functions, like the Damerau-Levenshtein Loss [17], we opt not to incorporate them in this work because 0/1 loss gives a user-friendly interpretation of the quality of Branch (e.g., 99% tuples in the dataset adhere to Branch). In contrast, while Damerau-Levenshtein may deliver a finer-grained measurement of the deviation, it would be potentially challenging for non-technical users to provide a meaningful threshold to rule out the "bad" programs.

ϵ-validity. We rule out the "bad" programs by a threshold ϵ. Specifically, given a dataset D and a $p \in \text{Prog}$, we say that p is ϵ-valid on D if

$$\forall b \in p, L(b, D) \leq |D^b|\epsilon \tag{3}$$

where $|D^b|$ is the number of tuples in D that satisfy the condition of b and $\epsilon \in [0, 1)$ is a threshold. Intuitively, Eqn. 3 states that the loss of each branch in p is less than the threshold, indicating that all branches in p are applicable to the dataset D with high probability. Likewise, the statement-level thresholding is defined as follows:

$$\forall b \in s, L(b, D) \leq |D^b|\epsilon \tag{4}$$

where s is a statement. A statement s satisfying Eqn. 4 is also called ϵ-valid on D.

Coverage. Given a dataset D, we first define the coverage of a branch b on D as follows:

$$\text{cov}(b, D) = \frac{|D^b|}{|D|} \tag{5}$$

where $|D^b|$ is the number of tuples in D that satisfy the condition of b. Then, for a statement s, we define the coverage of s on D as the sum of the coverage of all branches in s:

$$\text{cov}(s, D) = \sum_{b \in s} \text{cov}(b, D) \tag{6}$$

Equivalently, $\text{cov}(s, D) = \frac{|D^s|}{|D|}$. Intuitively, Eqn. 5 and Eqn. 6 measure the scope of the branch and the statement on the dataset D, respectively. Note that, due to variable size of statements within a program, the coverage of a program p is defined as the average coverage of all statements in p.

3 RESEARCH OVERVIEW

Holistically, we aim to synthesize a program that is likely to be the ground truth DGP. In the standard programming from example (PBE) setting, given a dataset D, the synthesis problem is to synthesize a $p^* \in \text{Prog}$ that adhere the specifications entailed by the dataset D. Formally, the synthesis problem is defined as follows:

$$p^* \in \{p \mid p \in \text{Prog} \wedge \bigwedge_{b \in p} L(b, D^b) \leq |D|\epsilon\} \tag{7}$$

where $\bigwedge_{b \in p} L(b, D^b) \leq |D|\epsilon$ implies that the program is ϵ-valid on D. Similar to many program synthesis problems [16], the synthesis problem defined in Eqn. 7 is ill-posed. For instance, an empty program trivially satisfies the synthesis problem. In the following, we will delve into the details of "genuinely optimal" program of interests and present a solution to the synthesis problem.

In the following subsections, we present the key challenge of synthesizing well-formed DGP in Sec. 3.1. Accordingly, we decompose the synthesis problem into two sub-problems and present the overview of our solution in Sec. 3.2.

3.1 Challenge: Expressiveness vs. Complexity

As shown above, the primary hurdle of DGP synthesis is to balance the expressiveness and complexity of the DSL. To illustrate this hurdle, consider the following example in which we present three

$$\textsc{Branch-Attr}\ \frac{a_1 \in \text{Attribute} \quad \forall i = 1, \ldots, n. b_i = (\texttt{IF}\ c_i\ \texttt{THEN}\ a_{b_i} \leftarrow l_{b_i}) \in \text{Branch} \qquad a_{b_i} = a_1}{\Gamma \vdash \texttt{GIVEN}\ a^+\ \texttt{ON}\ a_1\ \texttt{HAVING}\ b_1, \cdots, b_n}$$

$$\textsc{Branch-Cond}\ \frac{\forall b_j = (\texttt{IF}\ c_{b_j}\ \texttt{THEN}\ a \leftarrow l) \in b^+ \quad \exists (v_1, \ldots, v_i) \in \text{comb}(a_1, \cdots, a_i). c_{b_j} = (a_1 = v_1\ \texttt{AND}\ \cdots\ \texttt{AND}\ a_i = v_i)}{\Gamma \vdash \texttt{GIVEN}\ a_1, \cdots, a_i\ \texttt{ON}\ a\ \texttt{HAVING}\ b^+}$$

$\text{comb}(a_1, \cdots, a_i) = \text{dom}(a_1) \times \cdots \times \text{dom}(a_i)$ denotes the Cartesian product of the domains of $a_1, \cdots, a_i$ in the dataset.

Figure 3: Validation rules.

programs under the DSL, such that they satisfy the criterion in Eqn. 7 equally well.

Example 3.1. *Recall a relation R with attributes* PostalCode, City, State, *and* Country. *Suppose PostalCode decides City, City decides State, and State decides Country in the ground-truth DGP. That is, the ground-truth program p^* should be expressed as follows.*

$$p^* = \text{Stmt}_1\ \text{Stmt}_2\ \text{Stmt}_3$$

where

Stmt_1 = GIVEN PostalCode ON City HAVING $\cdots$

Stmt_2 = GIVEN City ON State HAVING $\cdots$

Stmt_3 = GIVEN State ON Country HAVING $\cdots$

where Stmt_i denotes the i-th statement in the program p. Then, a trivial program $p_1 = \emptyset$ would satisfy Eqn. 7, where $\emptyset$ denotes an empty program. In other words, the program p_1 simply does nothing and returns the original state. It is clear that the loss of the program is zero for arbitrary datasets. However, it does not help us to detect any errors in the data. On the other hand, a saturated program p_2 satisfies Eqn. 7 would be as follows.

$$p_2 = \text{Stmt}_1 \cdots \text{Stmt}_k$$

where

Stmt_1 = GIVEN PostalCode ON City HAVING $\cdots$

Stmt_2 = GIVEN City ON State HAVING $\cdots$

Stmt_3 = GIVEN State ON Country HAVING $\cdots$

Stmt_4 = GIVEN PostalCode ON State HAVING $\cdots$

Stmt_5 = GIVEN PostalCode ON Country HAVING $\cdots$

$\cdots$

Stmt_k = GIVEN PostalCode, City, State ON Country HAVING $\cdots$

Conceptually, $\text{Stmt}_1, \text{Stmt}_2, \text{Stmt}_3$ are sufficient to depict the required DGP well. However, it is non-trivial to rule out Stmt_4, Stmt_5, $\cdots$, Stmt_k from p_2 in the synthesis process. For instance, Stmt_4 cannot be ruled out in the absence of Stmt_2. Likewise, Stmt_k cannot be ruled out in the absence of Stmt_3. The complex dependencies among attributes in the dataset make it challenging to identify a program that is both expressive and succinct.

3.2 Synthesis from Sketch

To overcome the hurdle noted in Sec. 3.1, our key insight is that the verbosity is mainly on the GIVEN and ON clauses of the statement while the content in the HAVING clause is independent of the verbosity. Naturally, if the GIVEN and ON clauses are fixed, the synthesis problem would be much more tractable.

$$p[\cdot] \in \text{ProgSketch} := s^*$$
$$s[\cdot] \in \text{StmtSketch} := \texttt{GIVEN}\ a^+\ \texttt{ON}\ a\ \texttt{HAVING}\ \square$$
$$a \in \text{Attribute} := \{a_1, \cdots, a_n\}$$

Figure 4: Syntax of the sketch language $\mathcal{S}$

Algorithm 1: Fill program sketch $p[\cdot]$

Input: Dataset D, Program Sketch $p[\cdot]$
Output: Concrete Program p

$p \leftarrow \emptyset$;
foreach $s[\cdot] \in p[\cdot]$ **do**
  $s \leftarrow \texttt{FillStmtSketch}(s[\cdot], D)$;
  if $s \neq \perp$ **then** $p \leftarrow p \cup \{s^*\}$;
end
return p;
Function FillStmtSketch($s[\cdot], D$):
  $b^+ \leftarrow \emptyset$;
  det $\leftarrow s[\cdot]$.GIVEN;
  dep $\leftarrow s[\cdot]$.ON;
  Let C be the set of all warranted conditions for det according to comb(det);
  foreach $c \in C$ **do**
    $b^*[\cdot] \leftarrow$ (IF c THEN dep $\leftarrow \square$);
    $l^* \leftarrow \arg\min_{l \in \text{dep}} L(b^*[l], D)$;
    if $L(s[b^*], D) < |D^{b^*}|\epsilon$ **then**
      $b^+ \leftarrow b^+ \cup \{b^*[l^*]\}$;
    end
  end
  if $b^+ \neq \emptyset$ **then return** $s[b^+]$;
  else return $\perp$;

Motivated by this observation and also inspired by the sketching idea in many program synthesis problems [28, 33], we first assume the sketch of the DGP is given. Then, the task of the synthesis procedure is to fill the "holes" (i.e., the HAVING clauses) in the sketch. In this regard, we are able to bind the complexity of the synthesis procedure by the sketch and offload the major complexity to the sketch learning procedure. In summary, the synthesis of DGP comprises two steps: (1) sketch learning and (2) synthesis from sketch. In the remainder of this section, we assume the sketch is given a priori and focus on the synthesis from the sketch. We will delve into the details of learning the sketch in Sec. 4.

Sketch Language and Notations. Now, we present the syntax of the sketch language, as shown in Fig. 4. The sketch language is derived from the DSL presented in Fig. 2. In accordance with many previous works [28], on the sketch level, we only focus on discovering inter-variable dependencies among attributes in the dataset and abstracting away their concrete interactions. In particular, a program sketch remains all statements in the concrete program. However, each statement in the sketch only focuses on the GIVEN and ON clauses while leaving the branches in the HAVING clause as a hole; we express those holes as $\square$ in the sketch syntax in Fig. 4. In the context of sketching, we extend the definition of ϵ-validity and coverage (defined in Sec. 2.2) to the sketch language: given a dataset D, a statement sketch $s[\cdot]$ is said to be ϵ-valid if and only if there exists a concretized statement $s[b^+]$ such that $s[b^+]$ is ϵ-valid. Similarly, given a dataset D, a program sketch $p[\cdot]$ is said to be ϵ-valid if and only if there exists a concretized program $p[h]$ (h is the parameter for filling the hole) such that $p[h]$ is ϵ-valid (see detailed definition in Sec. 4.1). In addition, given a dataset D, the coverage of a statement sketch $s[\cdot]$ is defined as the coverage of the its best-fit concretized statement $s[b^+]$ (as defined in Eqn. 8).

Program Statement: Synthesis from Opaque Examples. Formally, this concretized statement $s[b^+]$ is defined as follows.

$$s[b^+] = \underset{s[b^+]\in s[\cdot]}{\arg\max}\ \text{coverage}(s[b^+], D) \text{ s.t. } L(s[b^+], D) < |D^b|\epsilon \quad (8)$$

Synthesis Procedure. The introduction of sketch language effectively offloads the major complexity of the synthesis procedure to the sketch synthesis (introduced in Sec. 4). In the context of synthesis from sketches, given a dataset D and a sketch $p[\cdot]$, we aim to synthesize a concrete program p that satisfies the criterion in Eqn. 7. We outline the procedure in Alg. 1. Given the sketch, the synthesis procedure is straightforward. For each statement $s[\cdot]$ in the sketch, Alg. 1 first initializes an empty branch list b^+ (line 8). Then, Alg. 1 extracts the determinant and dependent attributes from the GIVEN and ON clauses, respectively (lines 9–10). Then, Alg. 1 enumerates the Cartesian product for the values of the determinant attributes to constitute all warranted conditions for the determinant attributes (line 11). Next, Alg. 1 enumerates each condition c in C and initializes a branch sketch $b^*[\cdot]$ (line 13). Then, Alg. 1 picks the best-fit assignment for the dependent attribute l^* (line 14). By filling the hole in the branch sketch $b^*[\cdot]$ with l^*, Alg. 1 then checks whether the branch is ϵ-valid on D (line 15). If the branch is ϵ-valid, Alg. 1 adds the branch to the branch list b^+ (line 16). If the resulting branch list b^+ is not empty, it is filled to the hole in the statement sketch $s[\cdot]$ which is then added to the concrete program p (lines 19). Finally, Alg. 1 returns the concrete program p when all statements in the sketch are concretized (line 6). In short, the procedure between lines 7–20 in Alg. 1 aims to find the concretized statement $s[b^+]$ that is ϵ-valid on D with the highest coverage (compared to other possible ϵ-valid statements under the same sketch).

4 BUILDING DGP SKETCH

This section first formulate the sketches of interest following the discussion in Sec. 3.1. Then, we present the workflow to synthesize the sketches from the data step by step.

4.1 Characterizing Sketches of Interest

We first characterize the class of sketches that are of interest. To this end, we define three criteria to characterize the sketch for programs, namely ϵ-validity (as an extension of ϵ-validity on concrete programs in Sec. 2.2) and (locally/globally) non-triviality.

Definition 4.1 (ϵ-validity). *Given a dataset D and a noise threshold ϵ, a sketch $p[\cdot]$ is said to be ϵ-valid if and only if there exists a concretized program $p[h]$ (where h is the parameter for filling the hole) such that $p[h]$ is ϵ-valid on D.*

However, even though a sketch is ϵ-valid, it does not necessarily mean that it is "meaningful" with respect to the underlying DGP. For example, consider two attributes a_i and a_j in the dataset D. Even if a_i and a_j are purely independent (i.e., $a_i \perp\!\!\!\perp a_j$), we can still construct a ϵ-valid statement sketch $s[\cdot]$ with ϵ being the accuracy of random guess. However, such a trivial statement sketch is neither informative nor helpful for detecting errors.

Definition 4.2 (Local Non-triviality). *Given a dataset D with joint probability distribution P_D and a statement sketch $s[\cdot]$, we say that $s[\cdot]$ is locally non-trivial if and only if $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}}$, where a_j is the dependent attribute in the ON clause, $a_{\boldsymbol{k}}$ is the determinant set of a_j in the GIVEN clause, and $\not\perp\!\!\!\perp$ denotes statistical dependence. A program sketch $p[\cdot]$ is said to be locally non-trivial if and only if all statement sketches in $p[\cdot]$ are locally non-trivial.*

Def. 4.2 regulates the notion of validity of statement sketches. Recall that the sketch encodes the inter-variable dependencies of a program. In particular, it enforces that the statement sketches must entail a statistical correlation between the dependent attribute and the determinant set. Local non-triviality implies that there exists a concretized statement $s[b^+]$ that is at least better than a random guess. For example, in the relation described in Example 3.1, the statement sketch $s[\cdot]$ = `GIVEN PostalCode ON City` is locally non-trivial because `PostalCode` and `City` are correlated.

Definition 4.3 (Global Non-triviality). *Given a dataset D with its underlying joint probability distribution P_D and a program sketch $p[\cdot]$, we say that a locally non-trivial $p[\cdot]$ is global non-triviality if each statement sketch $s[\cdot]$ in $p[\cdot]$ fulfills the following condition: for every other statement sketch $s'[\cdot]$ in $p[\cdot]$, there exists a concretized statement $s'[b^+]$ such that $\exists b \in s'[b^+]$ $s[\cdot]$ is locally non-trivial on D^b.*

Global non-triviality extends the concept of local non-triviality to the entire program sketch and ensures that each statement sketch is deliberated in a holistic manner. The condition for global non-triviality is that each statement sketch can be interpreted as non-trivial on the subset of the data that is conditioned on the other statement sketches. In a statistical sense, it implies that the correlation entailed by the statement sketches will not be vanished by conditioning on other statement sketches. For instance, in the relation in Example 3.1, the correlation between `PostalCode` and `State` will be vanished if we already know `City`. In other words, it requires that each statement sketch shall provide additional information that is not already captured by other statement sketches. This criterion avoids constructing a program sketch that includes statement sketches, which, although individually non-trivial, do not

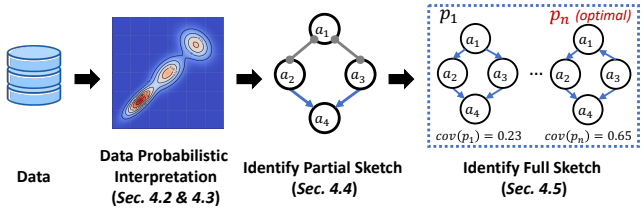

Figure 5: Sketch synthesis workflow.

jointly capture the true structure of the data when their interactions are considered. To illustrate, we present the following example.

Example 4.1. *Consider the relation described in Example 3.1, and assume three statement sketches:* $s_1[\cdot]$ = `GIVEN PostalCode ON City`, $s_2[\cdot]$ = `GIVEN PostalCode ON State` *and* $s_3[\cdot]$ = `GIVEN City ON State`. *Individually, each statement sketch is locally non-trivial, suggesting a meaningful relationship between the determinant and dependent attributes. However, when we consider these sketches together, the direct relationship between* `PostalCode` *and* `State` *may not be meaningful if* `City` *is already known. In terms of global non-triviality, we would find that* `PostalCode` *is actually irrelevant to* `State` *when* `City` *is specified (i.e., on the subset of the dataset* D^b *with* b *being a condition on* `City`*). Therefore, the combined program sketch* $p[\cdot] = s_1[\cdot]s_2[\cdot]s_3[\cdot]$ *does not exhibit global non-triviality because it includes redundant or irrelevant relationships. In Appendix [1], we provide a proof that elaborates on the deficiency (lacking global non-triviality) of this program sketch.*

Implication and Minimality. In data profiling, it is important that discovered constraints are minimal for reducing the cognitive load on users [4, 12, 30]. However, for solidifying ML-integrated SQL queries, minimality is less critical since testing constraints is not costly for machines. Nonetheless, our global non-triviality criterion implicitly enforces minimality. This criterion ensures that each statement sketch's determinant attributes are dependent on its dependent attribute in every subspace of other statement sketches. This means each statement sketch entails a unique dependency fact not captured by others and makes the program sketch minimal and free of redundant relationships. Thus, we consider a globally non-trivial program sketch to be "succinct."

In the remainder of this section, we will demonstrate how a globally non-trivial program sketch can be derived from the data, by following the workflow illustrated in Fig. 5. In Sec. 4.2 and 4.3, we will provide a probabilistic interpretation of the input data through the lens of the DGP and the resulting probabilistic graphical models (PGMs). Crucially, we will show that these PGMs inherently contain a globally non-trivial program sketch. Next, in Sec. 4.4, we will demonstrate the identifiability of the PGMs in Markov equivalence classes (MECs), which serve as a partial representation of program sketches. Finally, in Sec. 4.5, we will illustrate how to generate the full program sketch from the MEC by enumeration.

4.2 Data Generating Process (DGP)

In statistics, it often assumes that real-world data is generated from an (unknown) DGP. Here, we adopt the structural equation model (SEM) as the DGP to represent the data, which is defined as follows.

Definition 4.4 (Structural Equation Model [35]). *A SEM* M *consists of: (1) A directed acyclic graph (DAG)* G *with nodes* V *and edges* E*; (2) Exogenous variables* U*, representing factors outside the model; (3) Observed endogenous variables* V*, with each variable* $X \in V$ *functionally dependent on* $U_X \cup Pa_G(X)$*, where* $U_X \subseteq U$ *and* $Pa_G(X)$ *denotes the parent set of* X *in* G*; (4) Deterministic functions* $f_X \in F$*, each* $f_X : Pa_G(X) \times U_X \to X$ *computing the value of* X*.*

Each attribute in the data is represented by a random variable and the value of each attribute is derived from a deterministic generative process. The randomness of the data is captured by the unseen exogenous variables U. In the context of SEM, the dependency among attributes can be represented by a directed acyclic graph (DAG) G (also known as Bayesian network), a popular type of probabilistic graphical models (PGMs). In the DAG, each node represents an attribute and each edge represents a function.

Recall the definition of our DSL (defined in Sec. 2.2) and its sketch language (defined in Sec. 3.2), it is evident that a statement by definition corresponds to a function in the SEM. For example, given the aforementioned dataset (Example 3.1), the statement sketch below:

$$s[\cdot] = \texttt{GIVEN PostalCode ON City HAVING } \square$$

corresponds to a function f : `PostalCode` $\to$ `City` with $U_{\texttt{City}} = \emptyset$ in the SEM. Hence, one might be tempted to deduce a statement sketch from the graphical structure of the SEM (e.g., an edge from `PostalCode` to `City`). More importantly, the DAG-based representation sheds light on learning succinct sketches, as it explicitly distinguishes direct and indirect dependencies among attributes. In this regard, it has potential to prioritize f : `City` $\to$ `State` over f' : `PostalCode` $\to$ `State`, because the `PostalCode` is merely an indirect dependency of `State` through `City` in the SEM.

However, despite these seemingly appealing property, the challenge stems from the theoretical difficulty of formally establishing the correspondence between the sketches and the PGMs and the practical limitations of the existing structure learning algorithms.

4.3 From PGMs to Program Sketches

Now, we show that the PGMs not only provide a locally non-trivial sketch, but also inherently entails a globally non-trivial (i.e., succinct) sketch. Before the discussion, we first introduce the faithfulness assumption and the Markov property as follows.

Definition 4.5 (Faithfulness Assumption [35]). *A distribution* P *is faithful to a DAG* G *if and only if conditional independence relationships in* P *imply d-separations in* G*. Here, d-separation is a structural property of three sets of nodes in a DAG* G*. Two sets of nodes* A *and* B *are d-separated by a set of nodes* C *if and only if every path between a node in* A *and a node in* B *is blocked by* C*.*

Definition 4.6 (Markov Property [35]). *A distribution* P *is Markovian to a DAG* G *if and only if d-separations in* G *imply conditional independence relationships in* P*.*

The faithfulness assumption is a standard assumption when learning probabilistic graphical models. The main justification is that the set of unfaithful distributions is of measure zero (i.e., given a G, the set of such parameters has Lebesgue measure zero) [27]. Therefore, we take this assumption by default. Likewise, the Markov

property is an inverse of the faithfulness assumption. In other words, the two assumptions constitute a necessary and sufficient condition for the PGM to be a faithful representation of the underlying joint probability distribution.

Proposition 4.1. *Given a dataset D with its underlying joint probability distribution P_D and a PGM G, if P_D is Markovian to G, for a variable a_i and its parent variables $a_{\boldsymbol{k}}$ in G, the statement sketch $s[\cdot]$ with the determinant set of $a_{\boldsymbol{k}}$ and the dependent attribute a_i is locally non-trivial.*

Therefore, the above proposition (see proof in Appendix [1]) implies that, when the structure of the PGM is available, one can effortlessly read off a locally non-trivial program sketch according to the above proposition. However, it is unclear whether this locally non-trivial program sketch is also globally non-trivial. In the following theorem, we augment the claim and show that the PGMs inherently entail a globally non-trivial program sketch.

Theorem 4.2. *Given a dataset D with its underlying joint probability distribution P_D and a PGM G, if P_D is faithful to G, then the program sketch $p[\cdot]$ that is derived from G is globally non-trivial.*

Proof Sketch. Without loss of generality, let $s[\cdot]$ be an arbitrary statement sketch in $p[\cdot]$. In the context of the PGM, $s[\cdot]$ corresponds to one or a few directed edges from $a_{\boldsymbol{k}}$ to a_j in G, where $a_{\boldsymbol{k}}$ is the determinant attributes and a_j is the dependent attribute in $s[\cdot]$. By Proposition 4.1, $s[\cdot]$ is locally non-trivial. Therefore, we have that $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}}$. To show that $s[\cdot]$ is globally non-trivial, we need to show that $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}}$ on any subset of the data that is conditioned on other statement sketches in $p[\cdot]$ (i.e., D^b where b is a condition on other statement sketches in $p[\cdot]$). That is, the above condition is equivalent to showing that $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}} \mid b$, where b is the condition. Since we are to show that $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}}$ on any possible b, we can further rewrite the condition as $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}} \mid a_{\boldsymbol{z}}$ where $a_{\boldsymbol{z}}$ is any set of determinant attributes that are comprised by $p[\cdot]$. Since P_D is faithful to G, we have that $a_j \not\perp\!\!\!\perp a_{\boldsymbol{k}} \mid a_{\boldsymbol{z}}$ for any $a_{\boldsymbol{z}}$. Therefore, we have that $s[\cdot]$ is globally non-trivial. □

4.4 Characterizing Practically Learnable Sketches from MECs

In the previous paragraphs, we have shown that the PGMs can be used to generate a globally non-trivial program sketch. However, due to the limitations of the existing structure learning algorithms, the full structure of the PGM is not practically learnable from the data. In this section, we aim to understand under what conditions can we still (partially) deduce a globally non-trivial program sketch in practice. To this end, we first introduce the notion of Markov equivalence class (MEC) and then characterize the situations where we can directly read off the statement sketches from the MEC and the situations where post-processing is required.

Indeed, restricted by the identifiability of the PGM [35], under standard causal Markov and faithfulness assumptions, we cannot uniquely identify the PGM from the auxiliary distribution in general. Instead, we can learn up to the Markov equivalence class (MEC) of the PGM. Compared to the original DAG, the MEC is a summary of multiple possible DAGs given the data, which is defined as follows.

Definition 4.7 (Markov Equivalence Class [35]). *Given a joint probability distribution P, the MEC is the set of DAGs that are Markov equivalent under P, denoted as $[G]_P$ and the ground-truth DAG G is within $[G]_P$ (abbreviated as $[G]$). Two DAGs G_1 and G_2 are Markov equivalent if and only if they have the same skeleton (i.e., undirected graph) and the same set of v-structures (i.e., induced subgraphs of the form $A \rightarrow B \leftarrow C$ where A and C are non-adjacent).*

Typically, a MEC consists of multiple DAGs that are indistinguishable from each other given conditional independence relationships in the data. Instead of analyzing the ground-truth DAG, we have to establish the correspondence between the sketches and the PGMs in the context of MECs. Having said that, the DAGs within an MEC share the same adjacency relationships among attributes. That is, we can at least learn the adjacency relationships among attributes from the data. Below, we present some useful properties of the MEC to characterize the situation where we can directly read off the statement sketches.

Proposition 4.3. *Consider a locally non-trivial statement sketch $s[\cdot]$ derived from a MEC $[G]$, where $a_{\boldsymbol{k}}$ is the determinant set and a_j is the dependent attribute in $s[\cdot]$. If a_j is adjacent to every attribute in $a_{\boldsymbol{k}}$ within the MEC $[G]$, and each attribute in $a_{\boldsymbol{k}}$ is non-adjacent to the others in $[G]$, then in every G of the MEC $[G]$, there exists a directed edge from each attribute in $a_{\boldsymbol{k}}$ to a_j. Furthermore, there is no other directed edge to a_j in every G of the MEC $[G]$.*

Proposition 4.4. *Consider two conjunctive statement sketch $s_1[\cdot]$ and $s_2[\cdot]$ that are locally non-trivial where $s_1[\cdot]$ has a determinant set $a^1_{\boldsymbol{k}}$ with more than one attributes and a dependent attribute a_i, and $s_2[\cdot]$ has a determinant set $a^2_{\boldsymbol{k}} = \{a_i\}$ and a dependent attribute a_j. Then, in every G of the MEC $[G]$, there exists a directed edge from a_i to a_j. Furthermore, there is no other directed edge to a_j in every G of the MEC $[G]$.*

Proposition 4.5. *Consider two conjunctive statement sketch $s_2[\cdot]$ and $s_3[\cdot]$ that are locally non-trivial where $s_2[\cdot]$ has a determinant set $a^2_{\boldsymbol{k}} = \{a_i\}$ and a dependent attribute a_j, and $s_3[\cdot]$ has a determinant set $a^3_{\boldsymbol{k}} = \{a_j\}$ and a dependent attribute a_l. If $s_2[\cdot]$ satisfies the condition in Proposition 4.3, then, in every G of the MEC $[G]$, there exists a directed edge from a_j to a_l. Furthermore, there is no other directed edge to a_l in every G of the MEC $[G]$.*

Due to space limits, we prove Proposition 4.3 in Appendix [1]. The proofs of Proposition 4.4 is a direct consequence of Proposition 4.3. Proposition 4.5 can be proved by extending the latter part of the proof of Proposition 4.3. Therefore, we omit them.

In summary, we have demonstrated that statement sketches can be directly read off from the MEC in many scenarios. More precisely, direct extraction is possible except in instances where a statement sketch's determinant set consists of a single attribute that is not preceded by any statement sketch satisfying Proposition 4.4 or 4.5. In such cases, additional post-processing is needed to identify a statement sketch, as will be shown in Sec. 4.5.

4.5 Read off Program Sketches from the MEC

In the previous sections, we have shown that $[G]$ encodes essential (albeit not all) information from which we can read off a globally

non-trivial program sketch. In this section, we will show how to synthesize the program sketch from the MEC $[G]$.

Proposition 4.6. *Given a dataset D with its underlying joint probability distribution P_D and a MEC $[G]$, let a_j be a node and $a_{\boldsymbol{k}}$ be its parent set in $[G]$. If $a_{\boldsymbol{k}}$ is non-empty, then there exists a statement sketch $s[\cdot]$ with the determinant set of $a_{\boldsymbol{k}}$ and the dependent attribute a_j that can constitute a globally non-trivial program sketch.*

The above proposition (see proof in Appendix [1]) implies that, in most cases, we can directly read off a globally non-trivial program sketch from the MEC $[G]$. However, there are instances where multiple directions are possible, as shown in the following example.

Example 4.2. *For example, consider an MEC $[G]$ with chain-like structure as $a_1 - a_2 - a_3$. In this case, there might be two globally non-trivial program sketches: $p_1[\cdots] = s_1^1[\cdot]s_2^1[\cdot]$ where $s_1^1[\cdot]$ =* `GIVEN` a_1 `ON` a_2 `HAVING` $\square$ *and $s_2^1[\cdot]$ =* `GIVEN` a_2 `ON` a_3 `HAVING` $\square$*; and $p_2[\cdots] = s_1^2[\cdot]s_2^2[\cdot]$ where $s_1^2[\cdot]$ =* `GIVEN` a_3 `ON` a_2 `HAVING` $\square$ *and $s_2^2[\cdot]$ =* `GIVEN` a_2 `ON` a_1 `HAVING` $\square$.

Algorithm 2: Synthesize optimal program p from $[G]$

Input: Dataset D, MEC $[G]$
Output: Program p

1 $p^* \leftarrow \emptyset$;
2 **foreach** $G \in [G]$ **do**
3 $p[\cdot] \leftarrow \emptyset$;
4 **foreach** $a_j \in D$ **do**
5 $\text{pa}_j \leftarrow \text{Parent}(a_j, [G])$;
6 **if** $pa_j \neq \emptyset$ **then**
7 $s[\cdot] \leftarrow$ `GIVEN` pa_j `ON` a_j `HAVING` $\square$;
8 $p[\cdot] \leftarrow p[\cdot] \cup \{s[\cdot]\}$
9 **end**
10 $p \leftarrow p[s^*]$ according to Alg. 1;
11 **if** $cov(p, D) > cov(p^*, D)$ **then**
12 $p^* \leftarrow p$;
13 **end**
14 **end**
15 **return** p^*;

Here, we need to find the "optimal" one out of many feasible sketches. We outline the procedure for optimal program synthesis from the MEC $[G]$ in Alg. 2. Specifically, we enumerate all DAGs within the MEC $[G]$ (line 3; subject to a maximal enumeration of DAGs, omitted in Alg. 2 for brevity) and synthesize the program p from each DAG G (lines 4–10). Then, we select the program p with the highest coverage as the optimal program p^* (lines 11-13). We consider the coverage of a program p on a dataset D as the fitness metric because it indicates the "discriminative power" of p and helps distinguish the best program from suboptimal ones.

Example 4.3. *Consider a dataset with two attributes a_1 (*`City`*) and a_2 (*`State`*) and the MEC $[G]$ with the following two DAGs:*

(1) $G_1 = a_1 \rightarrow a_2$;
(2) $G_2 = a_1 \leftarrow a_2$.

In this case, there are two feasible program sketches: $p_1[\cdot]$ = `GIVEN` a_1 `ON` a_2 `HAVING` $\square$ *and $p_2[\cdot]$ =* `GIVEN` a_2 `ON` a_1 `HAVING` $\square$*. Based on common sense, $p_1[\cdot]$ is more meaningful than $p_2[\cdot]$ because one can hardly infer the* `City` *from the* `State`*. Yet, it does not necessarily mean that $p_2[\cdot]$ is not ϵ-valid. One possible concretized program $p_2[h]$ is $p_2[h]$ =* `GIVEN` a_2 `ON` a_1 `HAVING` `State = DC THEN City = Washington, DC` *where $p_2[h]$ only has one branch but is obviously valid where The District of Columbia (DC) has only one city, Washington. Using the coverage as the fitness criterion, we can prioritize the program $p_1[\cdot]$ over $p_2[\cdot]$ because $p_1[\cdot]$ is applicable for more tuples in the dataset D. In contrast, it is not possible to distinguish $p_1[\cdot]$ and $p_2[\cdot]$ solely using MEC or the ϵ-validity.*

5 IMPLEMENTATION

We implement our proposed system in Python and Julia with approximately 3K lines of code. To learn PGM from data, we adopt BLIP [32], an efficient algorithm whose implementation is well-optimized for modern multi-core CPUs. However, it is worth noting that our framework is agnostic to the choice of the PGM learning algorithm. Other algorithms, such as the PC algorithm and the Greedy Equivalence Search (GES) algorithm [35], can also be used to learn the PGM. Below, we discuss several key design and implementation choices and elaborate on details in Appendix [1].

Structure-Invairant Transformation. The complexity of real-world DGPs makes it challenging to accurately learn the PGM from real-world data. Hence, Zhang et al. [44] propose to learn the PGM from a transformed data collections. We formulate the transformation as an auxiliary distribution $P_{\mathbb{I}}$ derived from the data.

Definition 5.1 (Auxiliary Distribution). *Given a dataset D with joint probability distribution P_D for its attributes, we define a random vector $\mathbb{I}$. For any two tuples $t_1, t_2 \sim P_D$, the k-th component, $\mathbb{I}_k$, is defined as follows:*

$$\mathbb{I}_k = \begin{cases} 1 & t_1(a_k) = t_2(a_k), \\ 0 & t_1(a_k) \neq t_2(a_k) \end{cases} \tag{9}$$

As shown in Eqn. 9, the auxiliary distribution $P_{\mathbb{I}}$ is constructed by recasting the equality (or inequality) constraint between two tuples in the original data D (e.g., $t_1(a_k) = t_2(a_k)$) into the constant equality constraint on the binary-valued random vector $\mathbb{I}$. Loosely speaking, since $\mathbb{I}_k$ is purely determined by a_k, the graphical structure of $P_{\mathbb{I}}$ is identical to the graphical structure of P_D via a direct extension of the following proposition.

Proposition 5.1. *Given a dataset D with joint probability distribution P_D for its attributes $a_1, \cdots, a_n$, and its auxiliary binary distribution $P_{\mathbb{I}}$ for the random vector $\mathbb{I}$, we have*

$$a_i \perp\!\!\!\perp a_j \mid a_{\boldsymbol{k}} \iff \mathbb{I}_i \perp\!\!\!\perp \mathbb{I}_j \mid \mathbb{I}_{\boldsymbol{k}}. \tag{10}$$

where $\perp\!\!\!\perp$ denotes conditional independence, and $\boldsymbol{k}$ is a set of indices of attributes.

Despite implicitly assumed in the literature, we for the first time prove Proposition 5.1 in Appendix [1]. Also, the auxiliary distribution $P_{\mathbb{I}}$ is more "analysis-friendly" than the original data D because it ignores considerable information in D (e.g., the values of attributes) and complex interactions among attributes.

Error Handling Schemes. In accordance to the error handling schemes in Python `pandas` package, we implement three error handling schemes in our system: ❶ `raise` (i.e., raise an error when

encountering an error), ❷ `ignore` (i.e., ignore the error and continue the execution), and ❸ `coerce` (i.e., replace the error with a special value, such as `NaN`). We also implement an additional error handling scheme, ❹ `rectify`, to rectify the data errors by replacing the erroneous values with the correct ones entailed by the synthesized integrity constraints. This `rectify` scheme offers a lightweight capability to improve the accuracy of ML-integrated SQL queries, as will be shown in Sec. 6.3; in contrast, previous relevant works [12] do not provide such a capability.

6 EVALUATION

We evaluate our system on both synthetic and real-world datasets. Holistically, we aim to answer the following research questions:

- **RQ1: Data Error vs. Mis-prediction.** Is there a relationship between the data errors and model mis-predictions? How effective is GUARDRAIL in detecting data errors?
- **RQ2: Error Rectification.** How effective is GUARDRAIL in rectifying data errors and consequently improving the accuracy of ML-integrated SQL queries?
- **RQ3: Ablation Study.** How do different components and design choices of GUARDRAIL affect the overall performance?

Below, we first describe the experimental setup and then present the results for each research question in turn.

6.1 Experimental Setup

Datasets & ML Model Setup. To our best knowledge, there are no publicly available datasets that contain such ground truth data constraints and annotated data errors. Therefore, we first use 12 real-world datasets and inject data errors into them. We list the full details of these datasets in Appendix [1]. In short, the Adult dataset is a widely used dataset for classification tasks. The Lung Cancer dataset is a dataset used in the literature for causal discovery tasks with a ground truth causal graph, where some causal relationships enforce the integrity constraints on the data. The remaining datasets are from the OpenML repository or Kaggle and are used in the literature for tabular data classification tasks.

In accordance to Fariha et al. [12], we split the dataset into training and test sets and apply GUARDRAIL on the training split to synthesize constraints and then apply the learned constraints on the test split to detect data errors.

ML-Integrated SQL Queries. For each dataset, one author manually creates four ML-integrated SQL queries with varied complexity (48 queries in total). The remaining authors cross-check the queries and ensure that they are representative and meaningful. We describe the used queries in Appendix [1].

Error Injection. Since we primarily focus on data error-triggered mis-predictions in this work, we first rule out tuples where the ML model fails to make correct predictions without data errors. Then, we further exclude tuples that are not subject to constraints (i.e., the tuples that are not covered by the conditions in the learned constraints). We then inject data errors into the remaining tuples. To do so, we randomly flip the value of an attribute into another value (e.g., changing the value of "gender" from "male" to "female").

Environment. All experiments are conducted on a server with an AMD Threadripper 3970X CPU and 256 GB memory.

6.2 RQ1: Data Error and Mis-prediction

In this RQ, we aim to understand whether GUARDRAIL can effectively detect data errors in ML-integrated SQL queries. And we study if the detected data errors help us to further identify mis-predictions made by the ML model.

Effectiveness in Detecting Data Errors. We first evaluate the effectiveness of GUARDRAIL in detecting data errors, as shown in Table 1. We observe that GUARDRAIL can detect data errors with both high precision (i.e., the fraction of detected errors that are actually errors) and recall (i.e., the fraction of actual errors that are detected). Among 12 datasets, GUARDRAIL achieves a precision of 1.00 in all datasets and a recall of 1.00 in 8 out of 12 datasets. The high precision indicates that the detected errors are trustworthy. We interpret the performance of GUARDRAIL on recall as follows: First, high recall suggests that GUARDRAIL can comprehensively detect all data errors; considering that we introduce errors by randomly altering the values of attributes, some jointly injected errors may make a tuple appear logical and not constrained by the learned constraints, thus being inherently undetectable. As will be shown shortly, these errors are usually harmless and do not lead to mis-predictions.

Data Error vs. Mis-prediction. We then investigate how would these data errors affect the predictions made by the ML model. Recall that we have ruled out tuples that are inherently unpredictable before injecting data errors. Therefore, the resulting mis-predictions can only be caused by the data errors. As shown in Table 1, we observe that a considerable proportion of detected data errors lead to mis-predictions (0.235 ± 0.238; up to 0.833 in the Contraceptive Method dataset). This observation suggests that the data errors detected by GUARDRAIL are indeed the root cause of many mis-predictions made by the ML model. Moreover, if a data error is missed by GUARDRAIL, it is not likely to lead to a mis-prediction. Specifically, on the four datasets where GUARDRAIL misses data errors (i.e., Cylinder Bands, Diabetes, Phishing Websites, and Hotel Bookings), surprisingly, none of the missed data errors lead to mis-predictions. This observation further promotes the real-world applicability of GUARDRAIL in solidifying the ML-integrated SQL queries.

Processing Time. The computational overheads of constraint synthesis are proportional to both the number of DAGs in the MEC and the number of attributes. Therefore, we report these metrics in addition to the total processing time in Table 2. For datasets with fewer than 10 attributes, the processing time is 624.0 ± 34.1 seconds. Generally, datasets with more attributes take longer to process. However, some datasets, such as dataset #1, have lower processing times despite higher attribute counts. This is due to the relatively simple structure of DAGs in the MEC and the effectiveness of our statement-level cache mechanism in reducing overhead. Even for the most challenging dataset, the processing time remains reasonable at 1,376 seconds. Since the synthesis process is a one-off effort for each dataset, the processing time is practical for real-world applications.

6.3 RQ2: Error Rectification

In addition to detecting data errors, GUARDRAIL is capable of rectifying the data errors by replacing the erroneous values with the

Table 1: The effectiveness of GUARDRAIL in detecting data errors. "-" denotes there is no missed data errors by GUARDRAIL.

Dataset	# Errors	Precision	Recall	F1	#Mis-pred.	# Detected Mis-pred. / #Total Detected Data Errors	# Missed Mis-pred. / #Total Missed Data Errors
Adult	3377	1.00	1.00	1.00	426	0.13	-
Lung Cancer	1419	0.99	1.00	1.00	336	0.24	0.00
Cylinder Bands	35	0.95	1.00	0.97	2	0.06	-
Diabetes	19	0.95	1.00	0.97	5	0.26	-
Contraceptive Method	6	1.00	1.00	1.00	5	0.83	-
Blood Trans. Serv. Ctr.	48	1.00	1.00	1.00	14	0.29	-
Steel Plates Faults	124	1.00	1.00	1.00	14	0.11	-
Jungle Chess	521	1.00	1.00	1.00	321	0.62	-
Telco Customer Churn	444	1.00	1.00	1.00	25	0.06	-
Bank Marketing	1404	1.00	1.00	1.00	33	0.02	0.00
Phishing Websites	808	0.98	1.00	0.99	41	0.05	0.00
Hotel Bookings	2591	0.99	1.00	0.99	383	0.15	0.00

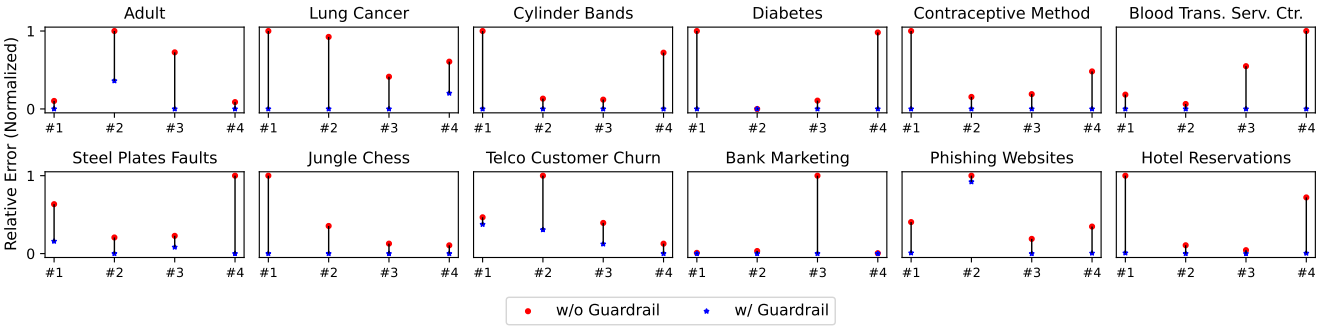

Figure 6: Effectiveness on rectifying data errors.

Table 2: Processing time.

Dataset ID	#1	#2	#3	#4	#5	#6
# DAGs	216	1	5	8	5	8
# Attr.	15	5	40	9	10	4
Time (s)	665	607	1205	690	605	604
Dataset ID	#7	#8	#9	#10	#11	#12
# DAGs	8	120	18	60	168	180
# Attr.	28	7	21	17	31	18
Time (s)	604	614	1376	820	1227	1301

correct ones entailed by the synthesized integrity constraints (using its `rectify` error handling scheme). In this RQ, we evaluate the effectiveness of GUARDRAIL in rectifying data errors and improving the accuracy of ML-integrated SQL queries.

For each dataset, we prepare four different ML-integrated SQL queries (48 queries in total; see details in our research artifact) and use the result of the query over clean data as the ground truth. We then compare the accuracy of different query execution modes: ❶ the vanilla query execution mode (i.e., w/o GUARDRAIL), ❷ the GUARDRAIL-augmented query execution mode with data errors being rectified by our synthesized constraints. However, it is worth noting that different queries have different base units of values. For example, query #1 may aggregate the average of the label by "`CASE WHEN label=1 THEN 1 ELSE 0`" clauses, while query #2 may directly count the number of tuples with label 1. Therefore, we first compute L_1 distance between the query outcomes over the clean data and the error-injected data as the *absolute error*. Next, we calculate the *relative error* by dividing the absolute error by the L_1 norm of the clean data query outcome. Finally, we normalize the relative error by min-max normalization such that all different queries have the same scale of error.

We report the results in Fig. 6 where red dots represent the relative error of the query over the data with data errors and blue dots represent the relative error of the query over the data with errors being rectified by GUARDRAIL. Overall, we observe that GUARDRAIL improves the accuracy for all the ML-integrated SQL queries on all datasets. In particular, it delivers an average reduction of 0.87 ± 0.25 compared to the vanilla query execution mode. To further demonstrate the effectiveness of GUARDRAIL, we also conduct a case study on the Adult dataset in Appendix [1].

6.4 RQ3: Ablation Study

In this RQ, we first investigate the impact of the sampler type on the coverage of the synthesized integrity constraints (see Sec. 5). We then study the impact of the threshold ϵ on the coverage and the loss of the synthesized integrity constraints (see Eqn. 8 and Eqn. 2), two conflicting objectives in the synthesis process.

As expected, sampling from the auxiliary distribution leads to a higher coverage of the synthesized integrity constraints compared to using the original data (i.e., the identity sampler). As shown in Fig. 8, auxiliary sampler yields significantly better results (p-value $= 0.037 \leq 0.05$). More importantly, the identity sampler remains unusable on three datasets with the coverage of 0. We presume that it is because the high cardinality of the attributes in these datasets makes it difficult for off-the-shelf structure learning solutions to identify the PGM. In contrast, the auxiliary sampler is more robust and yields a higher coverage.

In Fig. 7, we investigate the impact of the threshold ϵ on the coverage of the synthesized integrity constraints. Here, ϵ controls

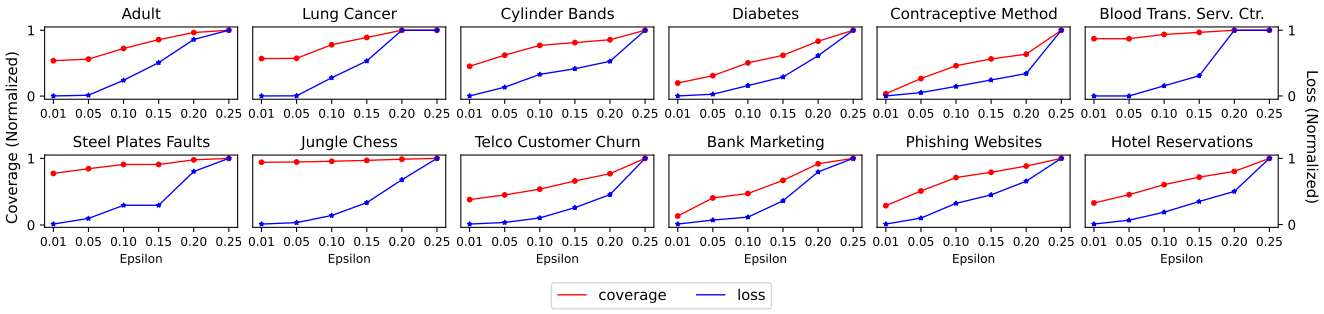

Figure 7: Ablation Study: Epsilon.

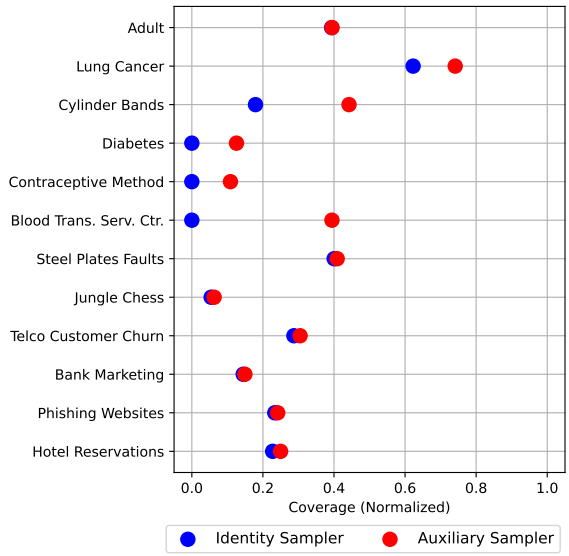

Figure 8: Coverage of Identity and Auxiliary Sampler.

tolerance of the integrity constraints to the noisy examples in the synthesis process. We observe that the coverage increases with the threshold ϵ for most datasets. As the cost of the coverage increases, the synthesized integrity constraints may become less accurate (i.e., higher loss; blue line in Fig. 7). Given the trade-off between the coverage and the loss, we recommend setting $\epsilon = 0.01 \sim 0.05$ for most datasets as it delivers a reasonable trade-off.

7 RELATED WORK

Program Synthesis. Program synthesis is the task of automatically generating a program that meets a given specification [16]. It has been widely used in many data science applications, such as data transformation [13] and SQL synthesis [37]. Recently, program synthesis has been extended to generate programs from noisy input-output examples [17–19]. In this paper, we make one step further to synthesize parametric integrity constraints from the data, which is a set of noisy and opaque input-output examples.

ML Reliability. Our work is similar to the detection of concept drifts or outliers in ML inference [14, 21, 26, 43]. However, GUARDRAIL distinguishes itself in several aspects. First, GUARDRAIL is purely data-driven and does not rely on ML models while many existing works are dependent on the probability predicted by the ML models to identity outliers. Second, modern outlier detection methods are often black-box and opaque. In contrast, as a constraint-based method, GUARDRAIL is naturally interpretable and can be easily integrated into the existing data processing pipeline (e.g., one can easily translate our DSL into standard SQL queries). Apart from these solutions, GUARDRAIL shares the same spirit with CCSynth [12] which aims to detect arithmetic conformance violations in ML models. However, GUARDRAIL complements, rather than competes with, it by focusing on the discrete data.

Data Validation. Data validation shares the same objective with our work, which is to ensure the quality of the data. However, existing data validation solutions often focus on data errors such as numeric outliers [5, 23], mis-spellings [38], uniqueness constraints [9], value range constraints [34, 36], etc. These validation rules primarily focus on *intra-column* constraints while GUARDRAIL focuses on using the underlying *inter-column* data-generating process to detect and rectify errors in the data.

8 LIMITATIONS

In this section, we discuss the potential limitations of GUARDRAIL and possible future directions to address them. First, GUARDRAIL is designed to work with categorical data, which presents unique opportunities for rectification (e.g., it is generally difficult to rectify a numeric outlier using constraints). When used for error detection, GUARDRAIL can work in conjunction with existing solutions (e.g., conformance constraints [12]) to handle numeric data. Second, due to the expressiveness of the DSL, GUARDRAIL may not be able to handle complex DGPs (e.g., string manipulation). In such cases, we can follow the formulation in [18] to support these operations and extend our synthesis algorithm accordingly. Third, as noted in the case study in Appendix [1], GUARDRAIL focuses on detecting and rectifying errors triggered by integrity constraints described in our DSL. However, in real-world scenarios, there may be other types of errors (e.g., change of units, change of value standard [36]) that cannot be captured by our DSL. We anticipate that existing data validation solutions can complement GUARDRAIL in these scenarios.

9 CONCLUSION

In this paper, we propose a novel approach to solidify ML-integrated SQL queries with automated integrity constraint synthesis. We recast traditional integrity constraint discovery as a program synthesis problem from noisy and opaque input-output examples. We introduce a DSL and a sketch-based synthesis algorithm to facilitate program synthesis. Evaluations demonstrate the high effectiveness of our approach across various real-world datasets.

REFERENCES

[1] 2024. Appendix. https://anonymous.4open.science/r/prog-syn-126A/appendix.pdf.

[2] 2024. Research Artifact. https://anonymous.4open.science/r/prog-syn-126A.

[3] William Ward Armstrong. 1974. Dependency structures of data base relationships. In *IFIP congress*, Vol. 74. Geneva, Switzerland, 580–583.

[4] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient denial constraint discovery with hydra. *Proceedings of the VLDB Endowment* 11, 3 (2017), 311–323.

[5] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 93–104.

[6] Yanzuo Chen, Yuanyuan Yuan, and Shuai Wang. 2023. OBSan: An Out-Of-Bound Sanitizer to Harden DNN Executables.. In *NDSS*.

[7] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding Language Models in Symbolic Languages. In *The Eleventh International Conference on Learning Representations*.

[8] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *Proceedings of the VLDB Endowment* 6, 13 (2013), 1498–1509.

[9] Tamraparni Dasu, Theodore Johnson, Shanmugauelayut Muthukrishnan, and Vladislav Shkapenyuk. 2002. Mining database structure; or, how to build a data quality browser. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 240–251.

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*. PMLR, 990–998.

[11] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505* (2020).

[12] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Alexandra Meliou. 2021. Conformance constraint discovery: Measuring trust in data-driven systems. In *Proceedings of the 2021 International Conference on Management of Data*. 499–512.

[13] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

[14] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 1–37.

[15] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[16] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[17] Shivam Handa and Martin Rinard. 2021. Inductive Program Synthesis over Noisy Datasets using Abstraction Refinement Based Optimization. *arXiv preprint arXiv:2104.13315* (2021).

[18] Shivam Handa and Martin Rinard. 2021. Program Synthesis Over Noisy Data with Guarantees. *arXiv preprint arXiv:2103.05030* (2021).

[19] Shivam Handa and Martin C Rinard. 2020. Inductive program synthesis over noisy data. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 87–98.

[20] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711.

[21] Dan Hendrycks and Kevin Gimpel. 2016. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv preprint arXiv:1610.02136* (2016).

[22] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. 2019. Extending relational query processing with ML inference. *arXiv preprint arXiv:1911.00231* (2019).

[23] Edwin M Knox and Raymond T Ng. 1998. Algorithms for mining distancebased outliers in large datasets. In *Proceedings of the international conference on very large data bases*. Citeseer, 392–403.

[24] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.

[25] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2023. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1208–1221.

[26] Haochuan Lu, Huanlin Xu, Nana Liu, Yangfan Zhou, and Xin Wang. 2019. Data sanity check for deep learning systems via learnt assertions. *arXiv preprint arXiv:1909.03835* (2019).

[27] Christopher Meek. 1995. Strong completeness and faithfulness in Bayesian networks. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. 411–418.

[28] Aditya V Nori, Sherjil Ozair, Sriram K Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices* 50, 6 (2015), 208–217.

[29] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1082–1093.

[30] Eduardo HM Pena, Eduardo C De Almeida, and Felix Naumann. 2019. Discovery of approximate (and exact) denial constraints. *Proceedings of the VLDB Endowment* 13, 3 (2019), 266–278.

[31] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[32] Mauro Scanagatta, Cassio P de Campos, Giorgio Corani, and Marco Zaffalon. 2015. Learning Bayesian networks with thousands of variables. *Advances in neural information processing systems* 28 (2015).

[33] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 281–294.

[34] Jie Song and Yeye He. 2021. Auto-validate: Unsupervised data validation using data-domain patterns inferred from data lakes. In *Proceedings of the 2021 International Conference on Management of Data*. 1678–1691.

[35] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. 2000. *Causation, prediction, and search*. MIT press.

[36] Dezhan Tu, Yeye He, Weiwei Cui, Song Ge, Haidong Zhang, Shi Han, Dongmei Zhang, and Surajit Chaudhuri. 2023. Auto-Validate by-History: Auto-Program Data Quality Constraints to Validate Recurring Data Pipelines. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4991–5003.

[37] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 452–466.

[38] Pei Wang and Yeye He. 2019. Uni-detect: A unified approach to automated error detection in tables. In *Proceedings of the 2019 International Conference on Management of Data*. 811–828.

[39] Marcel Wienöbst, Malte Luttermann, Max Bannach, and Maciej Liskiewicz. 2023. Efficient enumeration of Markov equivalent DAGs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 12313–12320.

[40] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1317–1334.

[41] Jing Nathan Yan, Oliver Schulte, MoHan Zhang, Jiannan Wang, and Reynold Cheng. 2020. Scoded: Statistical constraint oriented data error detection. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 845–860.

[42] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing data constraints in database-backed web applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1098–1109.

[43] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. 2021. {CADE}: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*. 2327–2344.

[44] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A statistical perspective on discovering functional dependencies in noisy data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 861–876.

APPENDIX

A Proof of Proposition 4.1

Proof. By the definition of the SEM, since $a_{\boldsymbol{k}}$ are the parent variables of a_i, there exists direct edges from $a_{\boldsymbol{k}}$ to a_i, which implies that $a_i \not\perp\!\!\!\perp a_{\boldsymbol{k}}$; because there does not exist any other nodes that that block the path between a_i and $a_{\boldsymbol{k}}$ in G. □

B Proof of Proposition 4.3

Proof. We first prove the presence of directed edges from each attribute in $a_{\boldsymbol{k}}$ to a_j in every DAG G within the MEC $[G]$ by contradiction. Assume that for some pair of attributes $a_l, a_m \in a_{\boldsymbol{k}}$, the triplet a_l, a_j, a_m does not form a v-structure in $[G]$. This means that we do not have the configuration $a_l \to a_j \leftarrow a_m$. According to the definition of faithfulness, the absence of a v-structure implies that for any set of attributes $a_{\boldsymbol{z}} \subset D$ that renders a_l and a_m conditionally independent, a_j must not be in $a_{\boldsymbol{z}}$. However, by the definition of a locally non-trivial statement sketch, we have that a_l is not conditionally independent of a_m given a_j, denoted as $a_l \not\perp\!\!\!\perp a_m \mid a_j$. This is a contradiction to the assumption that a_l, a_j, a_m do not form a v-structure, as the presence of a_j in the conditioning set should not affect the conditional independence of a_l and a_m if they do not form a v-structure. Therefore, the only consistent configuration under the MEC $[G]$ is that a_l, a_j, a_m must form a v-structure, which implies that there is a directed edge from a_l to a_j and from a_m to a_j in every DAG G within the MEC $[G]$.

Then, we prove the absence of other directed edges to a_j in every DAG G within the MEC $[G]$ by contradiction. Assume that there exists a directed edge from $a_q \notin a_{\boldsymbol{k}}$ to a_j in some DAG G within the MEC $[G]$. $a_q \to a_j \leftarrow a_m$ (where $a_m \in a_{\boldsymbol{k}}$) forms a v-structure in G. This implies that $a_q \not\perp\!\!\!\perp a_m \mid a_j$. However, by the definition of statement sketch, it is clear that $a_{\boldsymbol{k}}$ are the only determinant set of a_j. Therefore, the dependency between a_q and a_m must be mediated by a_j. That is, $a_q \perp\!\!\!\perp a_m \mid a_j$. This is a contradiction to $a_q \not\perp\!\!\!\perp a_m \mid a_j$. □

C Proof of Proposition 4.6

Proof. If $|a_{\boldsymbol{k}}| > 1$, according to Proposition 4.3, the statement sketch $s[\cdot]$ can be directly read off from G. If $|a_{\boldsymbol{k}}| = 1$, there is no more other directed edge to a_j in G (otherwise, it would be a v-structure according to Proposition 4.3 and contradict the fact that $|a_{\boldsymbol{k}}| = 1$). Therefore, $a_{\boldsymbol{k}}$ is the exact parent set of a_j in G. Hence, the statement sketch $s[\cdot]$ is consistent for all DAGs within $[G]$. Furthermore, according to Theorem 4.2, the ground-truth DAG G corresponds to a globally non-trivial program sketch. Therefore, the statement sketch $s[\cdot]$ can constitute a globally non-trivial program sketch. □

D Proof of Proposition 5.1

Proof. To prove the proposition, we need to show that the conditional independence between attributes a_i and a_j given a set of attributes $a_{\boldsymbol{k}}$ is equivalent to the conditional independence between the corresponding components $\mathbb{I}_i$ and $\mathbb{I}_j$ of the auxiliary binary distribution given the components $\mathbb{I}_{\boldsymbol{k}}$.

First, let's consider the left-hand side of the equivalence: $a_i \perp\!\!\!\perp a_j \mid a_{\boldsymbol{k}}$. This means that for any values $x_i, x_j, x_{\boldsymbol{k}}$ of $a_i, a_j, a_{\boldsymbol{k}}$, respectively, we have:

$$P(a_i = x_i, a_j = x_j \mid a_{\boldsymbol{k}} = x_{\boldsymbol{k}}) = P(a_i = x_i \mid a_{\boldsymbol{k}} = x_{\boldsymbol{k}}) \cdot P(a_j = x_j \mid a_{\boldsymbol{k}} = x_{\boldsymbol{k}}) \tag{11}$$

Now, let's consider the right-hand side of the equivalence: $\mathbb{I}_i \perp\!\!\!\perp \mathbb{I}_j \mid \mathbb{I}_{\boldsymbol{k}}$. This means that for any values $b_i, b_j, b_{\boldsymbol{k}} \in \{0, 1\}$ of $\mathbb{I}_i, \mathbb{I}_j, \mathbb{I}_{\boldsymbol{k}}$, respectively, we have:

$$P(\mathbb{I}_i = b_i, \mathbb{I}_j = b_j \mid \mathbb{I}_{\boldsymbol{k}} = b_{\boldsymbol{k}}) = P(\mathbb{I}_i = b_i \mid \mathbb{I}_{\boldsymbol{k}} = b_{\boldsymbol{k}}) \cdot P(\mathbb{I}_j = b_j \mid \mathbb{I}_{\boldsymbol{k}} = b_{\boldsymbol{k}}) \tag{12}$$

Given two tuples $t_1, t_2 \sim P_D$, we have:

$$P(\mathbb{I}_i = 1, \mathbb{I}_j = 1 \mid \mathbb{I}_{\boldsymbol{k}} = b_{\boldsymbol{k}}) = P(t_1(a_i) = t_2(a_i), t_1(a_j) = t_2(a_j) \mid t_1(a_{\boldsymbol{k}}) = t_2(a_{\boldsymbol{k}}) = x_{\boldsymbol{k}}) \tag{13}$$

where $b_{\boldsymbol{k}}$ is a binary vector indicating whether the corresponding attributes in $a_{\boldsymbol{k}}$ are equal in t_1 and t_2, and $x_{\boldsymbol{k}}$ is the common value of those attributes.

Now, if $a_i \perp\!\!\!\perp a_j \mid a_{\boldsymbol{k}}$, then:

$$\begin{aligned}
&P(t_1(a_i)=t_2(a_i), t_1(a_j)=t_2(a_j) \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
=&\sum_{x_i,x_j} P(t_1(a_i)=t_2(a_i)=x_i, t_1(a_j)=t_2(a_j)=x_j \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
=&\sum_{x_i,x_j} \frac{P(a_i=x_i, a_j=x_j, a_{\boldsymbol{k}}=x_{\boldsymbol{k}})^2}{P(a_{\boldsymbol{k}}=x_{\boldsymbol{k}})^2} \\
=&\sum_{x_i,x_j} \left(\frac{P(a_i=x_i, a_{\boldsymbol{k}}=x_{\boldsymbol{k}})}{P(a_{\boldsymbol{k}}=x_{\boldsymbol{k}})}\right)^2 \cdot \left(\frac{P(a_j=x_j, a_{\boldsymbol{k}}=x_{\boldsymbol{k}})}{P(a_{\boldsymbol{k}}=x_{\boldsymbol{k}})}\right)^2 \\
=&\sum_{x_i,x_j} P(t_1(a_i)=t_2(a_i)=x_i \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
&\cdot P(t_1(a_j)=t_2(a_j)=x_j \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
=&P(t_1(a_i)=t_2(a_i) \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
&\cdot P(t_1(a_j)=t_2(a_j) \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}})
\end{aligned} \tag{14}$$

which implies $\mathbb{I}_i \perp\!\!\!\perp \mathbb{I}_j \mid \mathbb{I}_{\boldsymbol{k}}$.

Conversely, if $\mathbb{I}_i \perp\!\!\!\perp \mathbb{I}_j \mid \mathbb{I}_{\boldsymbol{k}}$, then:

$$\begin{aligned}
&P(t_1(a_i)=t_2(a_i), t_1(a_j)=t_2(a_j) \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
=&P(t_1(a_i)=t_2(a_i) \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \cdot P(t_1(a_j)=t_2(a_j) \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}})
\end{aligned} \tag{15}$$

Summing over all possible values of x_i and x_j, we get:

$$\begin{aligned}
&\sum_{x_i,x_j} P(t_1(a_i)=t_2(a_i)=x_i, t_1(a_j)=t_2(a_j)=x_j \mid t_1(a_{\boldsymbol{k}})=t_2(a_{\boldsymbol{k}})=x_{\boldsymbol{k}}) \\
=&\sum_{x_i,x_j} \frac{P(a_i=x_i, a_j=x_j, a_{\boldsymbol{k}}=x_{\boldsymbol{k}})^2}{P(a_{\boldsymbol{k}}=x_{\boldsymbol{k}})^2} \\
=&\frac{P(a_i=x_i, a_j=x_j, a_{\boldsymbol{k}}=x_{\boldsymbol{k}})}{P(a_{\boldsymbol{k}}=x_{\boldsymbol{k}})} \\
=&P(a_i=x_i, a_j=x_j \mid a_{\boldsymbol{k}}=x_{\boldsymbol{k}})
\end{aligned} \tag{16}$$

which implies $a_i \perp\!\!\!\perp a_j \mid a_{\boldsymbol{k}}$.

Therefore, we have shown that $a_i \perp\!\!\!\perp a_j \mid a_{\boldsymbol{k}} \iff \mathbb{I}_i \perp\!\!\!\perp \mathbb{I}_j \mid \mathbb{I}_{\boldsymbol{k}}$.

□

E Revisiting FDX

In this section, we revisit FDX [44], a technically relevant work, and discuss its limitations in learning FD (a special case of the integrity constraints described in the main text).

Data Distribution. The first assumption made by FDX is that the resulting auxiliary distribution $P_{\mathbb{I}}$ forms a (additive) linear non-Gaussian structural causal model. In other words, it assumes that each attribute in $P_{\mathbb{I}}$ is generated in the following form:

$$\mathbb{I}_k = \frac{1}{|Pa_G(a_k)|} \sum_{a_i \in Pa_G(a_k)} \mathbb{I}_i + \eta_k, \tag{17}$$

where $Pa_G(a_k)$ is the parent set of a_k. In the sense, $\mathbb{I}_k$ is subject to the count of ones in the parent set of a_k and an additive noise η_k. However, recall that the real auxiliary distribution $P_{\mathbb{I}}$ is a binary-valued distribution. It is easy to see that the formulated linear non-Gaussian distribution deviates from the real distribution.

Furthermore, the assumption on the additivity of the noise term η_k is also questionable. Consider a simplified relational schema $R = \{a_1, a_2\}$ with an FD $a_1 \rightarrow a_2$ and the corresponding auxiliary distribution $P_{\mathbb{I}}$. We have $\mathbb{I}_2 = \mathbb{I}_1 + \eta_2$. Since η_k is additive in the functional form, we have $\eta_k \perp\!\!\!\perp \mathbb{I}_1$. However, this is not true in the real distribution. In fact, when $\mathbb{I}_1 = 0$, we have $P(\eta_2 = -1 \mid \mathbb{I}_1 = 0) = 0$ (as $\mathbb{I}_2 \in \{0, 1\}$) whereas $P(\eta_2 = -1 \mid \mathbb{I}_1 = 1) \neq 0$. Therefore, η_2 is dependent to $\mathbb{I}_1$ and the additivity assumption is violated.

Relationship between FDs and PGM. The second assumption made by FDX is that every FD on relational schema R corresponds to a direct edge in the probabilistic graphical model learned from $P_{\mathbb{I}}$, and vice versa. However, this assumption can be easily refuted by a simple counterexample. Consider a relational schema `Flight(IsRainy, IsDelayed)`. While there is no deterministic FD between `IsRainy` and `IsDelayed`, the two attributes are correlated. Given Proposition 4.3, there is an edge between `IsRainy` and `IsDelayed` in the PGM of $P_{\mathbb{I}}$, which may deduce a spurious FD.

F Implementation of Query Execution & ML Training/Inference

Because of the nature of GUARDRAIL, we need to intercept predictions made by the ML model and then enforce input error checking. However, off-the-shelf database with ML integration (e.g., SQL Server) does not support such interception. Hence, we spend considerable engineering effort to implement a SQL query executor with `pandas` ecosystem that supports the integration. Specifically, we first parse the SQL query and extract the ML model and the input attributes. Then, we feed the input tuples to GUARDRAIL and then pass them to the ML backend to make predictions. Finally, we replace the ML-related expressions in the SQL query with the predictions and execute the query accordingly. We implement several standard query optimization techniques, such as predicate pushdown, to improve the performance of the query execution. As a research prototype, our query executor only supports a subset of SQL features. For example, it does not natively support the `JOIN` operation. Currently, one can use the materialized views to pre-compute the intermediate results and then use our query executor to execute the SQL queries.

The design of GUARDRAIL is agnostic to the choice of the ML model. In our implementation, we adopt a popular AutoML library, `autogluon` [11], to perform end-to-end ML model training and inference. It trains various ML models (NN, tree-based models, etc.) and create an ensemble to achieve the best performance. Looking ahead, we plan to integrate more advanced ML models (e.g., LLM-enabled in-context learning [7]) and commercial AutoML services (e.g., Google Vertex AI) into our system.

G Implementation-level Optimizations

We implement our system in `Python` with `pandas` ecosystem. To accelerate the synthesis procedure, we adopt three implementation-level optimizations. First, we take the circular shift trick [44] for efficiently sampling from the auxiliary distribution and employ the vectorization to further speed up the computation. Second, we adopt a statement-level cache mechanism to avoid redundant concretizations of the same statement sketch incurred by different DAGs in the MEC. Third, we leverage a highly-efficient Julia implementation of PDAG (partial DAG; a generalized form of MEC) enumeration [39] to enable the MEC enumeration operation in Alg. 2.

H Dataset Description

Table 3: Dataset description

Dataset Name	Link	Category	# Attributes	# Tuples
Adult	https://archive.ics.uci.edu/dataset/2/adult	Demographic	15	48842
Lung Cancer	https://www.bnlearn.com/bnrepository/discrete-small.html#cancer	Medical	5	20000
Cylinder Bands	https://www.openml.org/search?type=data&id=6332	Manufacturing	40	540
Diabetes	https://www.openml.org/search?type=data&id=37	Medical	9	520
Contraceptive Method Choice	https://www.openml.org/search?type=data&id=23	Demographic	10	1473
Blood Transfusion Service Center	https://www.openml.org/search?type=data&id=1464	Medical	4	748
Steel Plates Faults	https://www.openml.org/search?type=data&id=40982	Manufacturing	28	1941
Jungle Chess	https://www.openml.org/search?type=data&id=41027	Game	7	44819
Telco Customer Churn	https://www.kaggle.com/datasets/blastchar/telco-customer-churn/data	Business	21	7043
Bank Marketing	https://archive.ics.uci.edu/dataset/222/bank+marketing	Business	17	45211
Phishing Websites	https://www.openml.org/search?type=data&id=4534	Security	31	11055
Hotel Reservations	https://www.kaggle.com/datasets/ahsan81/hotel-reservations-classification-dataset	Business	18	36275

We describe the datasets used in the experiments in Table 3. The Adult and Bank Marketing datasets are two widely used datasets for classification tasks. The Lung Cancer dataset is a dataset used in the literature for causal structure learning tasks with a ground truth causal graph, where some causal relationships enforces the integrity constraints on the data. The remaining datasets are from the OpenML repository or Kaggle and are used in the literature for tabular ML tasks.

I Case Study: Adult Dataset

In this section, we demonstrate the effectiveness of our proposed approach using a case study on the Adult dataset. We compare the results of ML-integrated SQL queries executed in three different modes: *ignore*, *rectify*, and *gt (ground truth)*. The following example illustrates this comparison.

The adult dataset contains demographic information of individuals, including their age, workclass, education. The task is to predict the income of an individual based on the demographic information. With GUARDRAIL, we can synthesize the integrity constraints from the data. Below, we show one of the synthesized integrity constraints and the rectification process to correct the prediction error.

```
...
GIVEN relationship ON marital-status HAVING
  IF relationship = Husband THEN marital-status ← Married-civ-spouse;
  IF relationship = Wife THEN marital-status ← Married-civ-spouse
...
```
(18)

The above integrity constraint states that if the relationship is Husband or Wife, then the marital status should be Married-civ-spouse. Then, we launch a ML-integrated SQL query to calculate the average age of individuals from the private work class with different income.

```
SELECT AVG(adult.age) FROM adult GROUP BY adult.income_pred WHERE adult.workclass == 'Private';
```

We first execute the query on the clean data (ground truth).

	income [predicted]	mean(age)
0	<=50K	34.90
1	>50K	43.72

However, for data with errors, there would result in a deviation from the ground truth, as shown below.

	income [predicted]	mean(age)
0	<=50K	35.25 (0.35)
1	>50K	44.00 (0.28)

The error introduces a deviation of 0.35 and 0.28 in the mean age for the two income groups. Indeed, the deviation could attribute to the violation of the integrity constraint. For example, the following tuple violates the integrity constraint

ID	age	workclass	fnlwgt	education	edu-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	$\text{income}_{\text{pred}}$
1064	45	Private	255014	Some-college	10	Separated	Craft-repair	Husband	White	Male	0	0	55	United-States	<=50K

Per the integrity constraint, given the individual's relationship is Husband, the marital status should be Married-civ-spouse instead of Separated. This error in turn leads to the prediction error in the income (from >50K to <=50K). With GUARDRAIL, we can rectify the error by correcting the marital status to Married-civ-spouse. By iterating the rectification process to all tuples, we run the query again and obtain the corrected result.

	income [predicted]	mean(age)
0	<=50K	34.90
1	>50K	43.72

As can be seen from the table above, the rectification process successfully corrects the prediction and reduces the deviation to 0.00 for both income groups.

Having said that, we acknowledge that in some cases, one tuple may be hard to correct due to the complex dependencies between the attributes. For instance, if the values for the marital status and relationship are both corrupted (e.g., the marital status is "Married-civ-spouse" → "Separated" and the relationship is "Husband" → "Not-in-family"), it is hard to determine the correct value for the marital status. In such cases, the rectification process may fail to correct the potential prediction error. However, we argue that such cases beyond the scope that the integrity constraints could cover, as the combination ("Separated" and "Not-in-family") stands for a valid relationship. In this case, the prediction error is not due to the violation of the integrity constraints, but rather the inherent ambiguity in the data.