GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables

GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables

MATHIEU HUOT, Massachusetts Institute of Technology, USA
MATIN GHAVAMI, Massachusetts Institute of Technology, USA
ALEXANDER K. LEW, Massachusetts Institute of Technology, USA
ULRICH SCHAECHTLE, Digital Garage, Japan
CAMERON E. FREER, Massachusetts Institute of Technology, USA
ZANE SHELBY, Digital Garage, Japan
MARTIN C. RINARD, Massachusetts Institute of Technology, USA
FERAS A. SAAD, Carnegie Mellon University, USA
VIKASH K. MANSINGHKA, Massachusetts Institute of Technology, USA

This article presents GenSQL, a probabilistic programming system for querying probabilistic generative models of database tables. By augmenting SQL with only a few key primitives for querying probabilistic models, GenSQL enables complex Bayesian inference workflows to be concisely implemented. GenSQL's query planner rests on a unified programmatic interface for interacting with probabilistic models of tabular data, which makes it possible to use models written in a variety of probabilistic programming languages that are tailored to specific workflows. Probabilistic models may be automatically learned via probabilistic program synthesis, hand-designed, or a combination of both. GenSQL is formalized using a novel type system and denotational semantics, which together enable us to establish proofs that precisely characterize its soundness guarantees. We evaluate our system on two case real-world studies—an anomaly detection in clinical trials and conditional synthetic data generation for a virtual wet lab—and show that GenSQL more accurately captures the complexity of the data as compared to common baselines. We also show that the declarative syntax in GenSQL is more concise and less error-prone as compared to several alternatives. Finally, GenSQL delivers a 1.7-6.8x speedup compared to its closest competitor on a representative benchmark set and runs in comparable time to hand-written code, in part due to its reusable optimizations and code specialization.

CCS Concepts: • **Mathematics of computing → Bayesian computation**; *Statistical software*; • **Software and its engineering → Semantics**.

Additional Key Words and Phrases: generative modeling, Bayesian data science, probabilistic query language

ACM Reference Format:
Mathieu Huot, Matin Ghavami, Alexander K. Lew, Ulrich Schaechtle, Cameron E. Freer, Zane Shelby, Martin C. Rinard, Feras A. Saad, and Vikash K. Mansinghka. 2024. GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables. *Proc. ACM Program. Lang.* 8, PLDI, Article 179 (June 2024), 26 pages. https://doi.org/10.1145/3656409

Authors' addresses: Mathieu Huot, mhuot@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Matin Ghavami, mghavami@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Alexander K. Lew, alexlew@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Ulrich Schaechtle, ulli-schaechtle@garage.co.jp, Digital Garage, Tokyo, Japan; Cameron E. Freer, freer@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Zane Shelby, zane-shelby@garage.co.jp, Digital Garage, Tokyo, Japan; Martin C. Rinard, rinard@csail.mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Feras A. Saad, fsaad@cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA; Vikash K. Mansinghka, vkm@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA.

ACM 2475-1421/2024/6-ART179
https://doi.org/10.1145/3656409

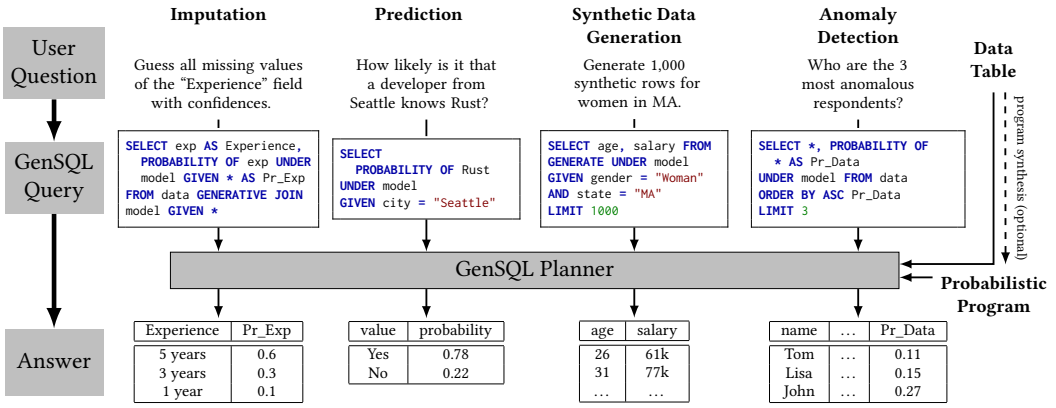


Fig. 1. Overview of GenSQL.

1 INTRODUCTION

Building generative models of tabular data is a central focus in Bayesian data analysis [28], probabilistic machine learning [55] and in applications such as econometrics [4], healthcare [38] and systems biology [84]. Motivated by these applications, researchers have developed techniques for automatically learning rich probabilistic models of tabular data [1, 30, 36, 50, 69]. To fully exploit these models for solving complex tasks, users must be able to easily interleave operations that access both tabular data records and probabilistic models. Examples computations include (i) generating synthetic data records that satisfy user constraints; (ii) conditioning distributions specified by probabilistic models given observed data records; and (iii) using database operations to aggregate the results of combined queries against tabular and model data. However, the majority of existing probabilistic programming systems are designed for specifying generative models and estimating parameters given observations. They do not support complex database queries that combine tabular data with generative models specified by probabilistic programs.

GenSQL. This article introduces GenSQL, a novel probabilistic programming system for querying generative models of database tables. GenSQL is structured as a declarative extension to SQL which seamlessly enables queries that integrate access to the tabular data with operations against the probabilistic model. Examples include predicting new data, detecting anomalies, imputing missing values, cleaning noisy entries, and generating synthetic observations [25, 29, 46, 73]. GenSQL introduces a novel interface and soundness guarantees that decouple user-level specification of high-level queries against probabilistic models from low-level details of probabilistic programming, such as probabilistic modelling, inference algorithm design, and high-performance machine implementations. GenSQL extends SQL with several constructs:

- To complement SELECT clauses that *retrieve* existing records from a table, GenSQL includes the clause GENERATE UNDER m to *generate* synthetic records from a probabilistic model m.
- To complement WHERE clauses that *filter* data via constraints, GenSQL introduces the clause m GIVEN e to *condition* a probabilistic model m on an event (i.e., a set of constraints) e.
- To complement *joins* between tables, GenSQL introduces a new *mixed join* clause t GENERATIVE JOIN m to join each row of a data table t with a synthetic row generated from a probabilistic model m, whose generation can be conditioned in a per-row fashion on the values of t.
- To complement arithmetic expressions, GenSQL introduces PROBABILITY OF e UNDER m expressions, which compute the probability (density) of an event e under a probabilistic model m.

In this work, we assume that an existing probabilistic program synthesis tool has been used to automatically generate a probabilistic model of the user's data satisfying a certain formal interface. The user then uploads the data and model to GenSQL which automatically integrates them. The user can then issue queries for a variety of tasks, as illustrated in Fig. 1. Although we envision most users using automatically discovered models on their data, the GenSQL implementation also supports hand-implemented or partly-learned probabilistic models. For instance, a user can develop custom models for harmonization across different sources, as shown in Appendix A.3.

The core of GenSQL is formalized as a simply-typed extension of SQL (Section 3.1). This extension includes standard SQL scalar expressions and tables as well as *rowModels* (probabilistic models of tables) and *events* (a set of constructs that allow users to issue probabilistic queries that leverage Bayesian conditioning). Together, rowModels and events enable a seamless integration of standard SQL databases with probabilistic models, which include queries that interleave accesses to the database records and probabilistic models.

The GenSQL *query planner* (Section 4) lowers queries into plans that execute against a new model interface for probabilistic models of tabular data. This *Abstract Model Interface* (AMI) (Section 4.1) provides a unifying specification of probabilistic models that are compatible with GenSQL. To implement the AMI, the model must be able to: (i) generative samples from a (potentially approximate) conditional distribution; (ii) compute probability densities for specified points; (iii) compute probabilities of sets in the support of the conditional distribution.

The open source GenSQL system includes a number of implementations of the AMI, including

- a Clojure implementation [61] of Gen [20], a general purpose probabilistic programming language; see Appendix A.3 for an example.
- models produced by CrossCat [50], a probabilistic program synthesis tool;
- SPPL [73], a probabilistic programming language for exact inference.

We provide a measure-theoretic denotational semantics for the language (Section 3.2). This semantics captures the interaction between deterministic SQL operations and probabilistic operations on the probabilistic model, enabling us to prove several correctness guarantees that query results satisfy. Specifically, we prove guarantees for (i) the *exact* case, where exact inference about marginal and conditional distributions of the probabilistic model is available (Theorem 4.2); and (ii) a range of *approximate* cases, where answers to marginal and conditional queries are obtained via approximate inference algorithms (Theorem 4.3).

We benchmark GenSQL on a set of representative queries, testing the runtime performance, overhead of the query planner, and effect of our optimizations. The results show that all queries execute in milliseconds against data tables of sizes up to 10,000 rows, with a speedup in the range 1.7–6.8x against the most closely related baseline, and that the query planner's overhead as compared to hand-written code is small. We evaluate our system on two case studies to test its applicability to solving real-world problems (conditional synthetic data generation for a virtual wet lab and an anomaly detection in clinical trials), comparing against a generalized linear model (GLM) and a conditional tabular generative adversarial network (CTGAN [88]) baseline.

Contributions. This paper makes the following contributions:

(1) The **GenSQL language** (Section 3.1), an extension of SQL with probabilistic models of tabular data as first-class constructs and probabilistic constructs to allow the integration of queries on these models with queries on the data.
(2) A **unifying abstract interface for models of tabular data** (Section 4.1), which bridges the query language and probabilistic models of database tables, to which all models must conform. The query planner lowers GenSQL queries on models to queries on this interface.
(3) **Soundness theorems**, which fall into two classes:

- **Exact:** We show that if models satisfy the exact interface, all deterministic computations will be exact (Theorem 4.2). This theorem works with an exact denotational semantics (Section 4.3) that precisely characterize the behavior of exact models.
- **Approximate:** If approximate models implement consistent estimators (i.e., estimators that converge to the true value), we prove that all queries return consistent results (Theorem 4.3). This theorem works with a novel denotational semantics that combines measure-theoretic aspects with sequences of random variables.

Together, these guarantees highlight some of the tradeoffs between using an exact model (which deliver stronger guarantees but may be difficult to obtain in some use cases) and an approximate model (which deliver weaker guarantees but are more easily available).

(4) An **open-source implementation** of GenSQL in Clojure (https://github.com/OpenGen/GenSQL.query), which can be compiled into JavaScript and run natively in the browser.

(5) A **performance evaluation** of our approach (Section 5) which establishes that GenSQL is competitive with hand-coded implementations and gives improved performance over a competitive baseline. Two case studies further demonstrate the utility of GenSQL.

2 EXAMPLE

Figure 2 presents an example GenSQL query. In this example, we work with a probabilistic model (`health_model`) derived from a national database of patient information, as well as a data table (`health_data`) from a set of local hospitals. The query uses the probabilistic model to estimate the mutual information—an information-theoretic measure used in data analysis— between the `age` and `bmi` columns (from the probabilistic model) for specific valueso f patient weights (selected from the data table). The mutual information is a statistical measure of the strength of the association between these two columns, defined as a sum or integral, over the joint distribution of `age` and `bmi`, of the logarithm of the ratio of the joint density and the product of the marginal density.

The query estimates the mutual information by Monte Carlo integration, i.e., it approximates the integral by sampling. We first generate 1000 copies of each row in the `health_data` table (line 15) and then use the GenSQL *generative join* construct (line 16) to complete each row as follows. For each such row r:

```
1  SELECT weight, AVG(log_pxy_div_px_py) AS mutual_information
2  FROM (
3    SELECT weight, LOG(pxy) - (LOG(px) + LOG(py)) AS log_pxy_div_px_py
4    FROM (
5      SELECT weight,
6        PROBABILITY OF h_model.age = table.age AND h_model.bmi = table.bmi
7          UNDER h_model GIVEN h_model.weight = table.weight AS pxy,
8        PROBABILITY OF h_model.age = table.age
9          UNDER h_model GIVEN h_model.weight = table.weight AS px,
10       PROBABILITY OF h_model.bmi = table.bmi
11         UNDER h_model GIVEN h_model.weight = table.weight AS py
12     FROM (
13       SELECT table.weight, table.age, table.bmi
14       FROM (
15         health_data DUPLICATE 1000 TIMES
16         GENERATIVE JOIN h_model
17         GIVEN h_model.weight = health_data.weight) AS table)))
18 GROUP BY weight
```

Fig. 2. Estimating the conditional mutual information between age and bmi given patient weights.

(1) a row r' is sampled from a version of the model conditioned on the `weight` value of row r;
(2) the rows r and r' are concatenated.

The resulting intermediate table is called `table` (line 17). Each synthetic row r' is used as a sample for the Monte Carlo integration of the conditional mutual information for the corresponding `weight` value. From this intermediate table, we select the `weight`, `age`, and `bmi` columns (line 13). Note that the `weight` column comes from the patient data while the `age` and `bmi` columns come from the rows sampled from the probabilistic model.

For each `weight` in the patient data, we compute the Monte Carlo approximation of the mutual information between `age` and `bmi` for that weight as

$$\frac{1}{1000k}\sum_{i=1}^{1000k}\log\frac{p(\texttt{age}_i,\texttt{bmi}_i)}{p(\texttt{age}_i)p(\texttt{bmi}_i)},$$

where k is the number of patients with that specific weight, and $(\texttt{age}_i, \texttt{bmi}_i)$ is a sample from the model for that weight. To do so, lines 6–11 compute the probability densities $p(\texttt{age}_i, \texttt{bmi}_i)$ (lines 6–7), $p(\texttt{age}_i)$ (lines 8-9), and $p(\texttt{bmi}_i)$ (lines 10–11). For instance, the `GIVEN` clause conditions the model on the `weight` column of the model being equal to the `weight` column of `table` (line 11). Line 10 then computes the probability density that the `bmi` column of the conditioned model is equal to the corresponding column in `table`. GenSQL computes these probability densities by invoking the **logpdf** function in the probabilistic model interface (Section 4.1).

A traditional SQL select statement (line 5) propagates the patient weights and corresponding probabilities `pxy`, `px`, and `py` to generate a table with four columns: the `weight` and the corresponding probability densities for that weight. Line 3 computes $\log p(\texttt{age}_i, \texttt{bmi}_i) - \log p(\texttt{age}_i)p(\texttt{bmi}_i)$ for each of the rows, naming this ratio `log_pxy_div_px_py`. Note that there are $1000k$ `log_pxy_div_px_py` values for each weight in the local patient data, where k is the number of patients with that specific weight. Finally, line 1 computes the mutual information estimate between `age` and `bmi`, for each weight, as the average of the `log_pxy_div_px_py` values for that weight. This example illustrates the expressivity of GenSQL, but we note that our implementation has a primitive which directly estimates conditional mutual information without the need to materialize intermediate tables.

3 SYNTAX AND SEMANTICS

3.1 Language

The core calculus extending SQL for querying from probabilistic models of tabular data is given in Fig. 3, and the type-system is given in Fig. 4.[1] As SQL is a subset of GenSQL, this calculus also includes a simply-typed formalization of SQL where terms are given in a pair of context: a local and a global one. We found this formalization interesting in its own right, as we could not find an equivalent formalization in the programming languages literature.

There are several noteworthy differences with variables and contexts from traditional simply-typed languages based on the lambda-calculus, which are explained below. We note early that we distinguish two types of conditioning through constructs called events and events-0. Events-0 come from a technical difficulty well-known in the PPL and measure theory literature [78, 86] when conditioning on a continuous variable taking a specific value. This creates a *possible event of probability 0*, and requires special treatment.

Names and Identifiers. We assume a countable set $\mathcal{C}$ of names $\textsc{col} \in \mathcal{C}$ for the columns of tables and rowModels, as well as a countable set $\mathcal{I}$ of identifiers $\textsc{id} \in \mathcal{I}$ for naming tables and rowModels.

[1]The core SQL-part of the language is minimal for expository purposes. Appendix E presents the formalization for a richer language, including the operations **GROUP BY** and **DUPLICATE** used in Fig. 2.

Description	SQL	Probabilistic Extension
Base/Event Type	$\sigma ::= \sigma_c \mid \sigma_d$	$\mathcal{E} ::= C^1\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\} \mid C^0\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$
Table/RowModel Type	$\mathcal{T} ::= T[\text{ID}]\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$	$\mathcal{M} ::= M[\text{ID}]\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$
Table Expression	$t ::= \text{ID} \mid \textsc{rename}\ t\ \textsc{as}\ \text{ID} \mid t_1\ \textsc{join}\ t_2 \mid t\ \textsc{where}\ e \mid \textsc{select}\ \bar{e}\ \textsc{as}\ \overline{\text{COL}}\ \textsc{from}\ t$	$t ::= \ldots \mid \textsc{generate under}\ m\ \textsc{limit}\ e \mid t\ \textsc{generative join}\ m$
RowModel Expression		$m ::= \text{ID} \mid m\ \textsc{given}\ c^i \mid \textsc{rename}\ m\ \textsc{as}\ \text{ID}$
Scalar Expression	$e ::= \text{ID.COL} \mid op(e_1, \ldots, e_n)$	$e ::= \ldots \mid \textsc{probability of}\ c^i\ \textsc{under}\ m$
Event Expressions		$c^1 ::= c_1^1 \wedge c_2^1 \mid c_1^1 \vee c_2^1 \mid \text{ID.COL}\ op\ e$
Event-0 Expressions		$c^0 ::= c_1^0 \wedge c_2^0 \mid \text{ID.COL} = e$
Primitive Domains: $op \in \mathbf{Op}$, $\text{ID} \in \mathcal{I}$, $\text{COL} \in C$, $\sigma_c \in \{\mathbf{Real}, \mathbf{PosReal}, \mathbf{Ranged}(a,b), \ldots\}$, $\sigma_d \in \{\mathbf{Int}, \mathbf{Str}, \mathbf{Nat}, \mathbf{Bool}, \ldots\}$		
Syntactic Sugar: $\bar{e}\ \textsc{as}\ \overline{\text{COL}} \equiv e_1\ \textsc{as}\ \text{COL}_1, \ldots, e_n\ \textsc{as}\ \text{COL}_n$.		

Fig. 3. Syntax of GenSQL.

Base types. Cells of tables can have a base type σ, which is either a continuous type σ_c or a discrete type σ_d. Continuous types are **Real** for reals, **PosReal** for non-negative reals, or **Ranged**(a, b) for reals in the range $[a, b]$. Discrete types are **Nat** for natural numbers, **Int** for integers, **Str** for strings, **Cat**$(\text{N}_1, \ldots, \text{N}_k)$ for a categorical type over k attributes, and **Bool** for Boolean.

Table and rowModel types. We denote these types by $D[?\text{ID}]\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$. D is either T for tables or M for rowModels. ?ID is an optional identifier, allowing access to columns of a table or rowModel in a query. For instance, in SELECT ID.weight, the identifier ID refers to a table and weight to a column of that table. The identifier can be be optional, e.g. there is no default identifier for a table created after a join. The notation $\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$ indicate that the table has columns COL_i of type σ_i. Therefore, we can think of each row of a table as an element of a record type $\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$, a bag of rows as a table, and a rowModel as a row generator.

Scalar Expressions. **Op** is a set of primitive operations on base types including standard operations such as $+, *, <, >, =$ on integers and reals, $\wedge, \vee$ on Booleans, as well as constants for every value of a base type. For any $op \in \mathbf{Op}$, we write $op : \sigma_1, \ldots, \sigma_n \to \sigma$ if op has arity n, takes arguments of base types $\sigma_1, \ldots, \sigma_n$ and returns a value of type σ. In particular, operations with no arguments are constants of the appropriate type such as **true** and **false** at the boolean type. All base types have an additional constant **Null** representing a missing value. This constant is preserved by primitive operations (e.g. **Null** $+ 3 \mapsto$ **Null**, **Null** $* 4.1 \mapsto$ **Null**). By convention, WHERE **Null** clauses act as WHERE **false**.

Table expressions. Apart from typical SQL operations, we have two ways to generate synthetic data. GENERATE UNDER returns a synthetic table with a given number of rows specified by the LIMIT clause, where each row is generated by sampling from a given rowModel. GENERATIVE JOIN takes a rowModel and a table, and returns a synthetic table with the same number of rows, where each row is generated by concatenating the current row of the table with a sample from the rowModel. The model generating the samples can be conditioned on the current row of the table. RENAME renames a table or rowModel with a new identifier, therefore changing the identifier in its type and the way to access their column in a select of event clause.

Event and event-0 expressions. Events are Boolean expressions on tables and rowModels, which include equality on discrete values but not on continuous values, which is reserved for events-0. The only probability 0 events are impossible under a given model, e.g. $x > 6 \wedge x < 3$, and those do not require a separate treatment. Events and events-0 are used in the PROBABILITY OF and GIVEN constructs.

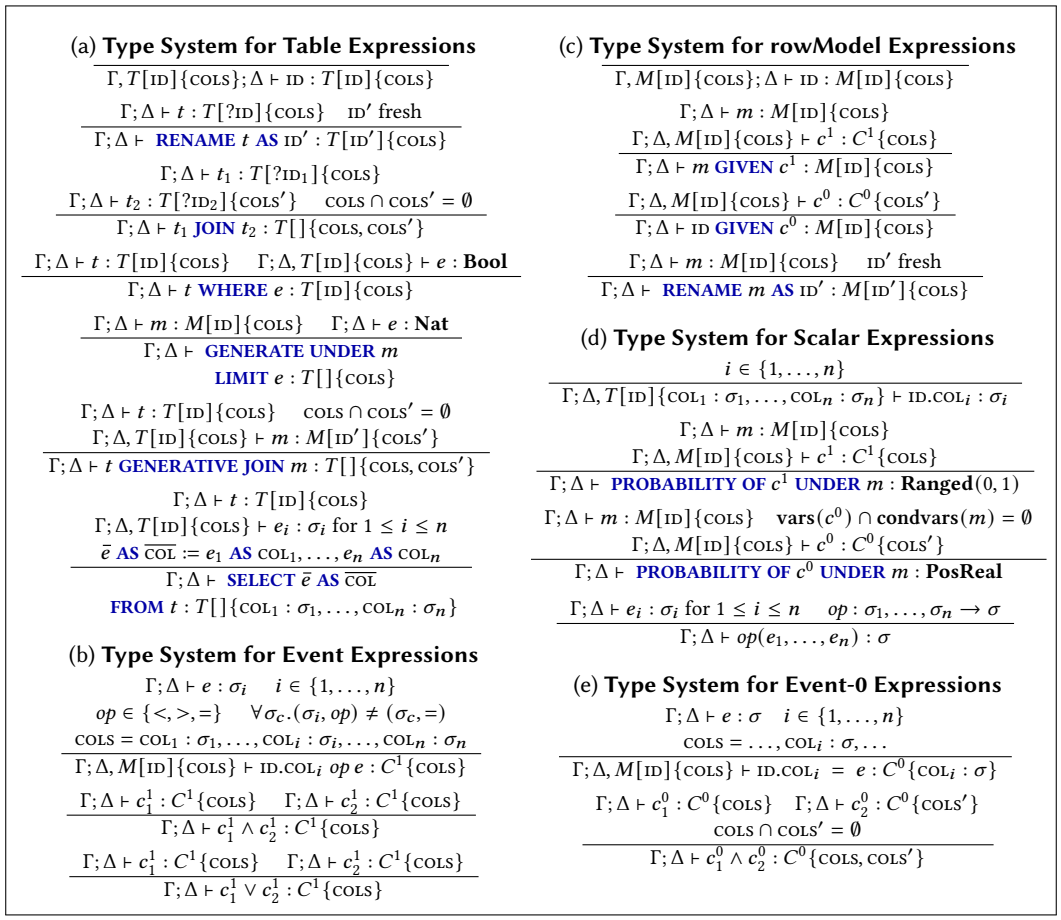


Fig. 4. Type system of GenSQL.

PROBABILITY OF takes an event (or event-0) expression and a rowModel to query and returns the probability (or probability density) of the event under the model.[2]

RowModel expressions. GIVEN takes a rowModel and an event (or event-0) expression, and returns a new rowModel, the conditional distribution of the original rowModel on the event. The event expression can be given by a list of inequalities on *arbitrary variables* and equalities on discrete variables, in which case GIVEN acts as a set of constraints on the possible returned values of the model. Otherwise, the event expression can be a set of equalities on possibly continuous values and is understood as conditioning the model on the given values.

Contexts. Expressions are typed in a pair of contexts $\Gamma; \Delta$ containing table and rowModel types. As these types include identifiers, there is no need for the more classical notation $x : \tau$ pairing a variable with its type. Γ is a *set* of types, while Δ is an *ordered list* of types. In the premise of a

[2]It may be confusing for people familiar with probabilistic programming languages (PPLs) to use PROBABILITY OF for both a probability mass and a probability density. Our implementation has two versions of the syntax: a strict one and a permissive one. The strict syntax distinguishes between the two, and in particular on events-0 one the primitive is PROBABILITY DENSITY OF. The permissive syntax allows to use PROBABILITY OF for both, and the system will automatically choose the right version based on the type of the event.

typing rule such as PROBABILITY OF , only the last element of Δ will be accessible to an expression. We denote the empty context by []. Intuitively, Γ contains the ambient tables in the database schema and any loaded models, and within Γ, all identifiers ID are assumed distinct. Δ is the *value environment*, and contains only tables that are "in scope" for a particular expression. Scalar, event, and event-0 expressions all depend on the value environment. If an expression has a table in scope, it will be iterated over the rows of that table and can only access the current row. If a PROBABILITY OF expression has a rowModel in scope, it will query the model for the probability of an event under that model. If it's a GIVEN expression, it will condition that model on an event.

Typing rules. Judgments are of the form $\Gamma;\Delta \vdash \mathtt{e} : \mathtt{t}$ where $\mathtt{e}$ is an expression (t, e, m, c^1, c^0 in Figure 3); $\mathtt{t}$ is a type (σ, $\mathcal{T}$, $\mathcal{E}$, $\mathcal{M}$ in Figure 3), and $\Gamma;\Delta$ is a context. Given some loaded tables and rowModels forming environment Γ, the objects of interest are "closed expressions" of table type, i.e., expressions of the form $\Gamma;[] \vdash t : T[?\text{ID}]\{\text{COLS}\}$. "Closed" here refers to Δ being empty, not Γ. Notable rules include those that need the same identifier twice, such as the PROBABILITY OF or the WHERE rule. For instance, in the t WHERE e rule, where t has identifier ID, a valid SQL e would be ID.COL = 3 where COL is a column of t. This reflects the fact that the expression e should have access to the identifier ID in its local environment, and that the column COL of t will be iterated over by the expression e.

Notations used in the type system. ?ID indicates an optional identifier and ID. "ID′ fresh" means that ID′ is not in the contexts Γ, Δ or in the type of a subterm of the expression. We will often abbreviate $\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}$ as $\{\text{COLS}\}$. We write $\text{COLS} \cap \text{COLS}' = \emptyset$ when the set of column names in COLS and in COLS′ should be disjoint. In the first typing rule for events, we write $\forall \sigma_c.(\sigma_i, op) \neq (\sigma_c, =)$ to mean that op cannot be an equality on a continuous type. We recursively define the following two macros:

$$\mathbf{vars}(\text{ID.COL}\ op\ t) = \{\text{COL}\} \quad \mathbf{vars}(c \wedge c') = \mathbf{vars}(c) \cup \mathbf{vars}(c') \quad \mathbf{vars}(c \vee c') = \mathbf{vars}(c) \cup \mathbf{vars}(c')$$

$$\mathbf{condvars}(m \text{ GIVEN } c^0) = \mathbf{vars}(c^0) \quad \mathbf{condvars}(m \text{ GIVEN } c^1) = \mathbf{condvars}(m) \quad \mathbf{condvars}(\text{ID}) = \emptyset$$

Restrictions imposed by the type system. If the same identifier ID appears twice in the premise of a typing rule, the two identifiers must equal, and two different identifiers ID and ID′ must be distinct. The JOIN and GENERATIVE JOIN operations require that the columns of the two tables have disjoint names. As explained above, events are disallowed to be equalities on continuous types. A model can only be conditioned once on an event-0, which is enforced by the restriction ID GIVEN c^0. Events-0 follow a linear typing system to avoid contradictory statements such as ID.COL = 1.0 ∧ ID.COL = 2.0. Events-0 in a PROBABILITY OF query on a conditioned model cannot refer to the conditioned columns of the model, which is enforced by the restriction $\mathbf{vars}(c^0) \cap \mathbf{condvars}(m) = \emptyset$.[3]

Syntactic sugar. Our implementation includes various syntactic sugars that are not present in the formalization but which are used in several figures. Given $t{:}T[\text{ID}]\{\text{COL}_1{:}\sigma_1, \ldots, \text{COL}_n{:}\sigma_n\}$, $m{:}M[\text{ID}']\{\text{COL}'_1{:}\sigma_1, \ldots, \text{COL}'_n{:}\sigma_n\}$, we have the following equivalences:

- SELECT ∗ FROM t ⇝ SELECT ID.COL$_1$, …, ID.COL$_n$ FROM t
- PROBABILITY OF ∗ ⇝ PROBABILITY OF e for any query of the form SELECT PROBABILITY OF ∗ UNDER m GIVEN c FROM t, where $e \coloneqq \text{ID}'.\text{COL}'_1 = \text{ID.COL}_1 \wedge \ldots \wedge \text{ID}'.\text{COL}'_n = \text{ID.COL}_n$.
- m GIVEN ∗ ⇝ m GIVEN e within a SELECT FROM t query. The event $e \coloneqq \text{ID}'.\text{COL}'_{i_1} = \text{ID.COL}_{i_1} \wedge \ldots \wedge \text{ID}'.\text{COL}'_{i_k} = \text{ID.COL}_{i_k}$, where the COL_{i_j} are columns t that do not appear in the SELECT clause.
- ∗ EXCEPT ID.COL removes the column COL from list of columns that ∗ selects.

[3]Our implementation is less restricted. It allows join variants such as SQL's left join where the tables do not have disjoint columns. It also allows multiple conditionings on the same model, which are then normalized to the form above. See Appendix D.1 for details about the normalization.

3.2 Semantics

We define denotational semantics using measure theory, shown in Fig. 5. Even though the SQL subset of GenSQL is not probabilistic, our probabilistic semantics ensures compositional reasoning about the semantics of SQL queries combined with probabilistic GenSQL expressions, such as synthetic tables generated by rowModels. Per usual, the semantics of expressions is defined compositionally on typing judgement derivations, and $[\![\mathtt{e}]\!]$ is a shorthand for $[\![\Gamma;\Delta \vdash \mathtt{e} : \mathtt{t}]\!]$.

Base types (Fig. 5c). We assign to each type σ a measure space $[\![\sigma]\!] := (X_\sigma, \Sigma_{X_\sigma}, \nu_\sigma)$ consisting of a set X_σ, a sigma-algebra Σ_{X_σ}, and reference measure ν_σ. $\mathbb{Z}$ denotes the set of integers, $\mathbb{N}$ natural numbers, and $\mathbb{B}$ Booleans, which are equipped with the discrete sigma-algebra. We equip the reals $\mathbb{R}$ with the Borel sigma-algebra. We interpret **Null** by adding a fresh element $\{\star\}$ to the standard interpretation of each base type, equipped with the discrete sigma-algebra. The semantics of a base type σ is then given by the smallest sigma-algebra making $\{\star\}$ measurable, as well as ensuring that every previously measurable set remains measurable. (This construction is also called the "direct-sum sigma-algebra" [24, 214L]).

The base measure on discrete types σ_d such as **Int**, **Nat**, **Bool**, **Str** is the counting measure. On continuous types σ_c such as **Real**, the base measure is the Lebesgue measure λ. These are extended to base measures ν_σ on $[\![\sigma]\!]$ by using the dirac measure $\delta_{\{\star\}}$ on $\{\star\}$, e.g. the base measure on $[\![\mathbb{R}]\!]$ is $\lambda_{\mathbb{R}} + \delta_{\{\star\}}$. We write $\mu \otimes \nu$ for the product of measures. We extend the reference measure to the product space $\prod_{1\le i\le n}[\![\sigma_i]\!]$ by taking the product of the reference measures $\nu := \bigotimes_{1\le i\le n} \nu_{\sigma_i}$.

Table types (Fig. 5a). Our semantics has two modes of interpreting table types, a "tuple mode" $\mathsf{Tup}[\![-]\!]$, and a "table mode" $\mathsf{Tab}[\![-]\!]$. $\mathsf{Tab}[\![-]\!]$ interprets tables as measures on bags of tuples, while $\mathsf{Tup}[\![-]\!]$ interprets a table as a tuple, representing the current row of the table being processed by a scalar, event or event-0 expression. More precisely, we denote by $\mathcal{P}(X)$ the measurable space of probability measures on the standard Borel space X [32]. The table semantics interprets table types as measures on bags of tuples $\mathsf{Tab}[\![T[?\textsc{id}]\{\textsc{cols}\}]\!] = \mathbf{Bag}(\mathsf{Tup}[\![T[?\textsc{id}]\{\textsc{cols}\}]\!])$, where $\mathbf{Bag}(X) = \{f : X \to \mathbb{N} \mid f(x) = 0 \text{ except for finitely many } x\}$. $\mathbf{Bag}(X)$ is equipped with the least sigma-algebra containing the generating sets $\{b \in \mathbf{Bag}(X) \mid b \text{ contains exactly } k \text{ elements in } A\}$ for measurable sets A of X [21].

Contexts (Fig. 5b). We interpret the global context Γ with the table semantics $\mathsf{Tab}[\![-]\!]$ and the local context Δ with the tuple semantics $\mathsf{Tup}[\![-]\!]$. We write γ for an element of $\mathsf{Tab}[\![\Gamma]\!]$, and see it as a finite map from identifiers to values. Likewise, we write δ for an element of Δ. We write $\delta[\textsc{id} \mapsto v]$ for the extended finite map mapping $\textsc{id}$ to v.

Scalar expressions (Fig. 5d). We then interpret scalar expressions $\Gamma;\Delta \vdash e : \sigma$ as measurable functions $\mathsf{Tab}[\![\Gamma]\!] \times \mathsf{Tup}[\![\Delta]\!] \to [\![\sigma]\!]$. We lift operations op to interpret **Null**, and write op_s for the extended version of op which sends $\star$ to $\star$.

Event expressions (Fig. 5g). An event expression $[\![c^1 : C^1\{\textsc{cols}\}]\!](\gamma, \delta)$ is interpreted as a measurable subset S of $[\![\textsc{cols}]\!]$ (disjoint union of hyper-rectangles [73]). Depending on the expression, this set S is used in different ways. We interpret the probability clause PROBABILITY OF c^1 UNDER m as $\int_{[\![\textsc{cols}]\!]} \mathbb{1}_S d\mu$, where μ is the measure denoting the model m, i.e. S is used in an indicator function $\mathbb{1}_S$. When used in a GIVEN clause, we constrain the model to the event S, which is then renormalized. If the event has probability 0, we instead return a row of **Null**. A similar situation to WHERE **Null** arises for GIVEN , e.g. in GIVEN ID.COL op **Null**. Following the principle of least surprise, **Null** acts by convention as a unit for conditioning, i.e. ID GIVEN ID.COL op **Null** behaves the same as ID. To ensure this we interpret boolean expressions op differently in the semantics of events, and write op_l for the extended version of op which sends $\star$ to **true**. The denotation of ID.COL op **Null** will therefore be the entire space, and conditioning a model on this event will not change its denotation.

(a) **Semantics of Table and rowModel Types**

$$\mathrm{Tup}[\![T[?\textsc{id}]\{\textsc{col}_1 : \sigma_1, \ldots, \textsc{col}_n : \sigma_n\}]\!] = \prod_{1\le i\le n}[\![\sigma_i]\!]$$
$$[\![M[?\textsc{id}]\{\textsc{col}_1 : \sigma_1, \ldots, \textsc{col}_n : \sigma_n\}]\!] = \mathcal{P}_{\mathrm{dens}}\Big(\prod_{1\le i\le n}[\![\sigma_i]\!]\Big)$$
$$\mathrm{Tab}[\![T[?\textsc{id}]\{\textsc{col}_1 : \sigma_1, \ldots, \textsc{col}_n : \sigma_n\}]\!] = \mathcal{P}\mathbf{Bag}\Big(\prod_{1\le i\le n}[\![\sigma_i]\!]\Big)$$
$$\mathrm{Tab}[\![M[?\textsc{id}]\{\textsc{col}_1 : \sigma_1, \ldots, \textsc{col}_n : \sigma_n\}]\!] = \mathcal{P}_{\mathrm{adm}}\Big(\prod_{1\le i\le n}[\![\sigma_i]\!]\Big)$$

(b) **Semantics of Contexts**

$$\mathrm{Tup}[\![\Delta := \mathcal{T}, \Delta']\!] = \mathrm{Tup}[\![\mathcal{T}]\!] \times \mathrm{Tup}[\![\Delta']\!]$$
$$\mathrm{Tup}[\![\Delta := \mathcal{M}, \Delta']\!] = [\![\mathcal{M}]\!] \times \mathrm{Tup}[\![\Delta']\!]$$
$$\mathrm{Tab}[\![\Gamma := \mathcal{T}, \Gamma']\!] = \mathrm{Tab}[\![\mathcal{T}]\!] \times \mathrm{Tab}[\![\Gamma']\!]$$
$$\mathrm{Tab}[\![\Gamma := \mathcal{M}, \Gamma']\!] = \mathrm{Tab}[\![\mathcal{M}]\!] \times \mathrm{Tab}[\![\Gamma']\!]$$

(c) **Semantics of Base Types**

$$[\![\mathbf{Bool}]\!] := (\mathbb{B} \cup \{\star\}, \mathcal{P}(\mathbb{B} \cup \{\star\}), \nu_{\mathbb{B}})$$
$$[\![\mathbf{Int}]\!] := (\mathbb{Z} \cup \{\star\}, \mathcal{P}(\mathbb{Z} \cup \{\star\}), \nu_{\mathbb{Z}})$$
$$[\![\mathbf{Str}]\!] := (\mathrm{Str} \cup \{\star\}, \mathcal{P}(\mathrm{Str} \cup \{\star\}), \nu_{\mathrm{Str}})$$
$$[\![\mathbf{Real}]\!] := (\mathbb{R} \cup \{\star\}, \mathcal{B}(\mathbb{R} \cup \{\star\}), \nu_{\mathbb{R}})$$
$$[\![\mathbf{PosReal}]\!] := (\mathbb{R}^+ \cup \{\star\}, \mathcal{B}(\mathbb{R}^+ \cup \{\star\}), \nu_{\mathbb{R}^+})$$

(d) **Semantics of Scalar Expressions**

$$[\![\textsc{id}.\textsc{col}_i]\!](\gamma, \delta) = \pi_i(\delta(\textsc{id})) \quad \text{where } T[?\textsc{id}]\{\textsc{cols}\} \in \Delta$$
$$[\![op(e_1, \ldots, e_n)]\!](\gamma, \delta) = op_s\,([\![e_1]\!](\gamma, \delta), \ldots, [\![e_n]\!](\gamma, \delta))$$
$$[\![\ \textbf{PROBABILITY OF}\ c^1\ \textbf{UNDER}\ m]\!](\gamma, \delta) = [\![m]\!](\gamma, \delta).\mathrm{meas}([\![c^1]\!](\gamma, \delta))$$
$$[\![\ \textbf{PROBABILITY OF}\ c^0\ \textbf{UNDER}\ m]\!](\gamma, \delta) = \textbf{let}\ (\pi, v) =$$
$$[\![c^0]\!](\gamma, \delta[\textsc{id} \mapsto [\![m]\!](\gamma, \delta)])\ \textbf{in}\ [\![m]\!](\gamma, \delta).\mathrm{pdf}(v)$$

(e) **Semantics of Table Expressions**

$$[\![\textsc{id} : T[?\textsc{id}]\{\textsc{cols}\}]\!](\gamma, \delta) = \gamma(\textsc{id}) \qquad [\![\ \textbf{RENAME}\ t\ \textbf{AS}\ \textsc{id}']\!](\gamma, \delta) = [\![t]\!](\gamma, \delta)$$
$$[\![t_1\ \textbf{JOIN}\ t_2]\!](\gamma, \delta) = (\lambda x, y.\textbf{map2}\ (\lambda r_1, r_2.(r_1, r_2))\ x\ y)_*([\![t_1]\!](\gamma, \delta) \otimes [\![t_2]\!](\gamma, \delta))$$
$$[\![t : T[?\textsc{id}]\{\textsc{cols}\}\ \textbf{WHERE}\ e]\!](\gamma, \delta) = \Big(\lambda x.\textbf{filter}\ (\lambda r.[\![e]\!](\gamma, \delta[\textsc{id} \mapsto r]))\ x\Big)_*[\![t]\!](\gamma, \delta)$$
$$[\![\ \textbf{GENERATE UNDER}\ m\ \textbf{LIMIT}\ e]\!](\gamma, \delta) = \textbf{let}\ n = [\![e]\!](\gamma, \delta)\ \textbf{in}\ \Big(\lambda(x_1, \ldots, x_n).\textstyle\bigcup_{1\le i\le n}\{x_i\}\Big)_* \bigotimes_{1\le i\le n}[\![m]\!](\gamma, \delta).\mathrm{meas}$$
$$[\![\ \textbf{SELECT}\ e_1\ \textbf{AS}\ \textsc{col}_1, \ldots, e_n\ \textbf{AS}\ \textsc{col}_n\ \textbf{FROM}\ t : T[?\textsc{id}]\{\textsc{cols}\}]\!](\gamma, \delta) =$$
$$\Big(\lambda x.\textbf{map}\ (\lambda r.([\![e_1]\!](\gamma, \delta[\textsc{id} \mapsto r]), \ldots, [\![e_n]\!](\gamma, \delta[\textsc{id} \mapsto r])))\ x\Big)_*[\![t]\!](\gamma, \delta)$$
$$[\![(t : T[?\textsc{id}]\{\textsc{cols}\})\ \textbf{GENERATIVE JOIN}\ m]\!](\gamma, \delta) = [\![t]\!](\gamma, \delta) \gg\!= \Big(\lambda y.\textbf{fold}\ (\lambda \mu, r.\mu \gg\!=$$
$$\Big(\lambda x.(\lambda\{r'\}.x \cup \{(r, r')\})_*[\![\ \textbf{GENERATE UNDER}\ m\ \textbf{LIMIT}\ 1]\!](\gamma, \delta[\textsc{id} \mapsto r]).\mathrm{meas}\Big)\ \delta_{\{\}}\ y\Big)$$

(f) **Semantics of rowModel Expressions**

$$[\![\textsc{id} : M[?\textsc{id}]\{\textsc{cols}\}]\!](\gamma, \delta) = (\gamma(\textsc{id}).\mathrm{meas}, \gamma(\textsc{id}).\mathrm{pdf}) \qquad [\![\ \textbf{RENAME}\ m\ \textbf{AS}\ \textsc{id}']\!](\gamma, \delta) = [\![m]\!](\gamma, \delta)$$
$$[\![m\ \textbf{GIVEN}\ c^1 : C^1\{\textsc{cols}\}]\!](\gamma, \delta) = \mathbf{Cond}([\![m]\!](\gamma, \delta), [\![c^1]\!](\gamma, \delta[\textsc{id} \mapsto [\![m]\!](\gamma, \delta)]))$$
$$[\![\textsc{id}\ \textbf{GIVEN}\ c^0 : C^0\{\textsc{cols}'\}]\!](\gamma, \delta) = \textbf{let}\ (\pi, v) = [\![c^0]\!]((\gamma, \delta[\textsc{id} \mapsto [\![\textsc{id}]\!](\gamma, \delta)]))\ \textbf{in}\ \mathbf{Dis}(\gamma(\textsc{id}), \pi, v)$$

(g) **Semantics of Event Expressions**

$$[\![\textstyle\bigwedge_{1\le i\le 2} c_i^1]\!](\gamma, \delta) = \bigcap_{1\le i\le 2}[\![c_i^1]\!](\gamma, \delta)$$
$$[\![\textstyle\bigvee_{1\le i\le 2} c_i^1]\!](\gamma, \delta) = \bigcup_{1\le i\le 2}[\![c_i^1]\!](\gamma, \delta)$$
$$[\![\textsc{id}.\textsc{col}_i\ op\ e : C^1\{\textsc{cols}\}]\!](\gamma, \delta) = \{(x_1, \ldots, x_n) \in [\![\textsc{cols}]\!] \mid x_i\ op_l\ [\![e]\!](\gamma, \delta)\}$$

(h) **Semantics of Event-0 Expressions**

$$[\![\bigwedge_{1\le i\le 2} c_i^0]\!](\gamma, \delta) = \begin{cases}\textbf{let}\ _{1\le i\le 2}(f_i, v_i) = [\![c_i^0]\!](\gamma, \delta)\ \textbf{in} \\ (\lambda x.(f_1(x), f_2(x)), (v_1, v_2))\end{cases}$$
$$[\![\textsc{id}.\textsc{col}_i\ =\ e]\!](\gamma, \delta) = (\pi_i, [\![e]\!](\gamma, \delta))$$

Fig. 5. Denotational semantics of GenSQL.

Event-0 expressions (Fig. 5h). $[\![c^0 : C^0\{\textsc{cols}\}]\!](\gamma, \delta)$ is interpreted as a pair of a projection function π and a value v in the codomain of the projection. v is used to specify the point at which we want to condition or evaluate a density, and π is used to project the model to the relevant subspace, which we detail in the paragraph on rowModel expressions.

Table expressions (Fig. 5e). We interpret closed tables expressions $\vdash t : T[?\textsc{id}]\{\textsc{cols}\}$ as measures on their columns, i.e. elements of $\mathcal{P}\Big(\mathrm{Tab}[\![T[?\textsc{id}]\{\textsc{cols}\}]\!]\Big)$. We write $\mu \gg\!= \kappa$ for the composition of a measure μ on X with a kernel $X \to \mathcal{P}Y$, defined by $\mu \gg\!= \kappa(dy) = \int \kappa(x, dy)\mu(dx)$. Given a measurable function $f : X \to Y$, we denote the pushforward measure by $f_*\mu(A) := \mu(f^{-1}(A))$.

We use functional programming notation for the mathematical functions **filter** , **map** , **map2** , **fold** . Given a bag S and a function $f : S \to \mathbb{B}$, we define the bag **filter** $f\ S := \{x \in S \mid f(x) \neq 0\}$. Likewise, we define **map** $f\ S := \{f(x) \mid x \in S\}$ and **map2** $f\ S\ T := \{f(x, y) \mid x \in S, y \in T\}$. A function $f : X \times Y \to Y$ is commutative [21] if $f(x_1, f(x_2, y)) = f(x_2, f(x_1, y))$ for all x_1, x_2, y. Given a commutative function $f : X \times Y \to Y$, we further define **fold** $f : Y \times \mathbf{Bag}(X) \to Y$ by **fold** $f\ y_0\{x_1, \ldots, x_n\} = f(x_1, f(x_2, \ldots f(x_n, y_0) \ldots))$.

RowModel expressions (Fig. 5f). The semantics of rowModels is more involved, as conditioning statements GIVEN c^0 require conditioning on events of probability 0. We first review the minimal setting that helps us define conditioning on event-0 expressions. Given measurable spaces A, B with reference measure ν_A, ν_B, a measure μ on $A \times B$ admits an (A, B) *disintegration* if we can write $\mu = \nu_A \otimes \kappa$ for some measure kernel κ such that for all $a \in A$, $\kappa(a)$ has a density $p(- \mid a)$ w.r.t. ν_B. A *valid decomposition* (A, B) for $\prod_{1 \leq i \leq n} [\![\sigma_i]\!]$ is given by $A = \prod_{j \in J} [\![\sigma_j]\!]$ for some $J \subseteq \{1, \ldots, n\}$ and $B = \prod_{j \in \{1,\ldots,n\} - J} [\![\sigma_j]\!]$. A measure $\mu \in \mathcal{P}(\prod_{1 \leq i \leq n} [\![\sigma_i]\!])$ is *admissible* if it admits an (A, B) disintegration for all valid decompositions (A, B) of $\prod_{1 \leq i \leq n} [\![\sigma_i]\!]$.

We consider measures μ on spaces X with chosen disintegrations and (marginal) densities w.r.t. the reference measure. More precisely, we interpret a rowModel ID from the global context Γ as a quadruple $\mathrm{Tab}[\![\text{ID}]\!] := (\mu, p, \{\kappa_A\}_A, \{p\}_A)$. Here, μ is a measure denoting the unconditioned model, and p a density of μ w.r.t. the reference measure. For each valid decomposition (A, B) of the columns of ID, the kernel κ_A is an (A, B)-disintegration of μ. For all $a \in A$, $p_A(- \mid a)$ is a density for $\kappa_A(a)$ w.r.t. the reference measure ν_B. If v is a partial assignment of the variables in B, we also write $p_A(v \mid a)$ for the marginal density of $\kappa_A(a)$ at v obtained from $p_A(- \mid a)$ by integrating out the missing variables in v. We denote by $\mathcal{P}_{\mathrm{adm}}(X)$ the set of such quadruples $(\mu, p, \{\kappa_A\}_A, \{p\}_A)$, where μ is a measure on X. Given $m \in \mathcal{P}_{\mathrm{adm}}(X)$, we write $m.\mathrm{meas}$ for its first component μ, $m.\mathrm{pdf}$ for the density p, $m.A$ for the kernel κ_A, and $m.A.\mathrm{pdf}$ for the density p_A. Using this notation, given an event-0 c^0 denoting a projection π and value v, the expression $m.\mathrm{pdf}(v)$ gives a marginal density of the model m at v; i.e. $m.\mathrm{pdf}(v)$ is a version of the density of $\pi_* m.\mathrm{meas}$ evaluated at v. We assume that all the models in the context are admissible, which is enforced in the semantics of contexts.

The models used in queries are built from admissible models and will carry chosen densities, which is enforced in the semantics of rowModel expressions. We write $\mathcal{P}_{\mathrm{dens}}(X)$ for the set of pairs (μ, p) where μ is a measure on $X := X_1 \times \ldots \times X_n$ and p is either a density of μ w.r.t. the reference measure, or of the form $\lambda(x_1, \ldots, x_n).q(x_{i_1}, \ldots, x_{i_k})$ for some $i_1, \ldots, i_k$, and where q is a marginal density of μ on $X_{i_1} \times \ldots \times X_{i_k}$ w.r.t. the reference measure. The second case is used to represent the density of a model conditioned on an event-0 expression.

Conditioning on events-0 requires access to a disintegration of the model at the point v, which is possible thanks to the restriction from the type-system. For $m \in \mathcal{P}_{\mathrm{adm}}(X)$, $\pi : X \to Y$ a projection function, and $v \in Y$, we define $\mathbf{Dis}(m, \pi, v) := (m.\pi(X)(v) \otimes \delta_v, m.\pi(X).\mathrm{pdf}(v))$.

For conditioning on events, given $m \in \mathcal{P}_{\mathrm{dens}}(X)$ and a measurable $S \subseteq X$, we define

$$\mathbf{cond}(m, S) := \begin{cases} \left(\lambda S'.\, \dfrac{m.\mathrm{meas}(S \cap S')}{m.\mathrm{meas}(S)}, \lambda x.\, \dfrac{\mathbb{1}_S(x) m.\mathrm{pdf}(x)}{m.\mathrm{meas}(S)}\right) & \text{if } m.\mathrm{meas}(S) > 0 \\ \left(\delta_{\{\star,\ldots,\star\}}, \mathbb{1}_{\{(\star,\ldots,\star)\}}\right) & \text{otherwise} \end{cases}$$

4 ABSTRACT MODEL INTERFACE AND QUERY PLANNER

This section presents a query planner that automatically lowers GenSQL queries to programs that operate on tables and rowModels. This lowering depends on the Abstract rowModel Interface (AMI) which we assume all loaded rowModels must satisfy. The AMI is a flexible interface that many rowModel implementations can easily satisfy. This flexibility means that model implementations can strike different expressiveness-speed-accuracy trade-offs, and give different guarantees.

Appendix A.2 compares an exact SPPL backend to an approximate Gen.clj backend on 5 queries.

In what follows, we define the AMI and show how to lower GenSQL queries to programs that access rowModels through the AMI interface. We showcase the flexibility of the AMI by proving formal guarantees for two different implementations of the AMI. We show in Section 4.3 that if the AMI is implemented in a PPL with exact inference, then GenSQL queries can be lowered to programs in a semantics-preserving way. We then show in Section 4.4 that if the AMI is implemented in a PPL with approximate inference, then GenSQL queries can be lowered to programs that encode asymptotically sound estimators for PROBABILITY OF expressions and asymptotically sound samplers for GENERATE UNDER expressions.

4.1 Abstract Model Interface (AMI)

A rowModel represents a probability distribution on rows with a fixed set of columns. The AMI captures the intuition that a model should be able to produce samples and compute probabilities and densities for all conditioned versions of the distribution it represents. For each rowModel $M[?\text{ID}]\{\text{COLS}\}$, the AMI requires the existence of the following three methods:

$$\begin{aligned}
\textbf{simulate}_{\text{ID}} &: (C^0\{\text{COLS}'\}, C^1\{\text{COLS}\}) \to T[?\text{ID}]\{\text{COLS}\} \\
\textbf{logpdf}_{\text{ID}} &: (C^0\{\text{COLS}'\}, C^1\{\text{COLS}\}, C^0\{\text{COLS}''\}) \to \textbf{Real} \\
\textbf{prob}_{\text{ID}} &: (C^0\{\text{COLS}'\}, C^1\{\text{COLS}\}, C^1\{\text{COLS}\}) \to \textbf{Ranged}(0,1).
\end{aligned}$$

where $\text{COLS}', \text{COLS}'' \subseteq \text{COLS}$. These methods should behave as follows:

- $\textbf{simulate}_{\text{ID}}(c^0, c^1)$ returns a sample from a model with identifier ID, conditioned on the event-0 c^0 and event c^1.
- $\textbf{logpdf}_{\text{ID}}(c^0, c^1, c_2^0)$ returns the (marginal if $\text{COLS}'' \subsetneq \text{COLS}$) log-density of the model ID conditioned on the event-0 c^0 and event c^1, at the point c_2^0.
- $\textbf{prob}_{\text{ID}}(c^0, c^1, c_2^1)$ returns the probability of the event c_2^1 under the model ID, conditioned on the event-0 c^0 and event c^1.

A non-conditioned model is recovered by letting the subset COLS' to be empty. The precise usage of these methods is given in the next section. The AMI methods can have different formal semantics, capturing different aspects of the backend probabilistic model it abstracts. These semantics reflect different implementation strategies implementing conditional sampling and probability evaluation. Appendix C shows how different model classes can implement the AMI. In particular, we show that SPPL [73] and truncated multivariate Gaussians satisfy the exact AMI, and that any PPL implementing ancestral sampling will satisfy the approximate AMI. We next describe how the GenSQL query planner lowers queries to programs that rely on the AMI, before giving details about the semantics and correctness guarantees.

4.2 Lowering GenSQL to Queries on the AMI

The lowering procedure from GenSQL to a lowered language is given in two steps: (i) a normalization transform for GenSQL queries; and (ii) a program transform to the lowered language.

Normalization of GenSQL Queries. The normalization (see Appendix D.1) simply simplifies RENAME statements and aggregates events in a single conditioning statement. It leads to the following normal forms, where GIVEN clauses are optional:

- Probability queries: PROBABILITY OF c_1^i UNDER (ID GIVEN c^0 GIVEN c^1).
- Generate queries: GENERATE UNDER (ID GIVEN c^0 GIVEN c^1) LIMIT e and t GENERATIVE JOIN (ID GIVEN c^0 GIVEN c^1).

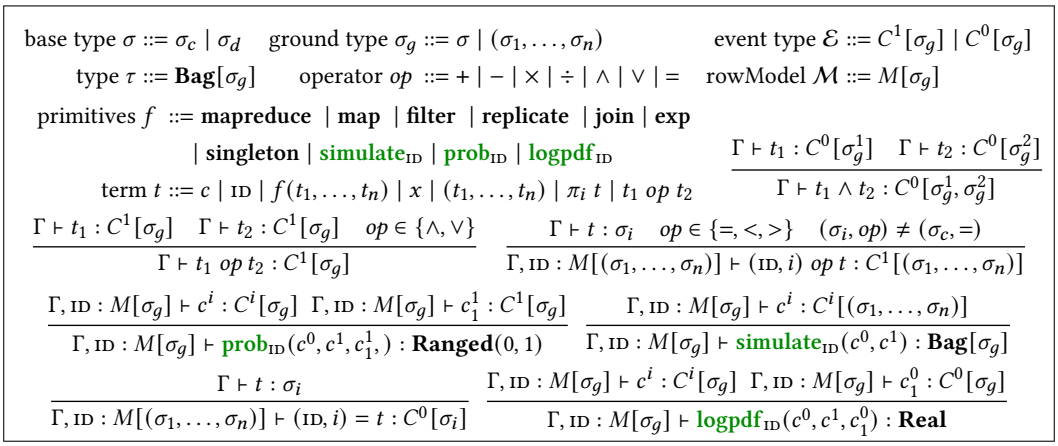


Fig. 6. A selected subset of the syntax and type system of the lowered language.

Lowering Language (Fig. 6). It is a first-order simply-typed lambda calculus with second-order operations acting on bags, and primitives for the AMI. It also contains a version of events and events-0 which can be used by AMI primitives. Operations like **map** , **filter** and **exp** have their usual meaning, and their typing along with those for constants, tuples, projections, and arithmetic operations are standard and recalled in Appendix D (Fig. 20). **join** takes two bags of tuples and returns their Cartesian product. **replicate** evaluates its bag argument n times and returns the union of all the resulting bags. **mapreduce** takes a bag of tuples and a function f from tuples to bags, and returns the union of all the bags obtained by applying f to each tuple in the input bag.

Lowering program transform (Fig. 7). After obtaining a normal form query, the planner applies a program transformation $\mathcal{T}_\delta\ \{\cdot\}$ from normalized GenSQL queries to the lowered language, defined by pattern matching on the structure of the query. It carries a local context δ of variables (a finite map from identifiers to variable names) which are bound in the surrounding program. Similarly to the local context Δ in GenSQL, δ will start empty [] at the root of the syntax tree. It is used to rename variables in the lowered query. The rationale is that a table identifier ID in Δ will be transformed to a variable r representing a tuple being iterated over by a **map** or **fold** primitive. A rowModel identifier ID, on the other hand, will be uniquely accessible and identified from the global context Γ, thanks to the normalization procedure which ensures that no rowModel is renamed in the normalized query. A simple proof by induction shows that the transformation preserves typing.

Proposition 4.1. *If* $\Gamma, [] \vdash t : T[?\text{ID}]\{\text{COLS}\}$, *then* $\mathcal{T}\ \{\Gamma\} \vdash \mathcal{T}_{[]}\ \{t\} : \mathcal{T}\ \{T[?\text{ID}]\{\text{COLS}\}\}$.

4.3 Lowering Guarantees for Exact Backend

A large class of models supports exact inference, e.g. those expressible in SPPL [73] and truncated multivariate Gaussians. These models satisfy the exact AMI and are able to return exact samples from **simulate**, and compute exact marginal **logpdf** and **prob** queries, even for conditioned models. We make this precise by giving a measure semantics on the lowered language (Fig. 21) and show that the program transform $\mathcal{T}\ \{\cdot\}$ preserves the semantics of the lowered query (Theorem 4.2). In particular, all the scalar computations in the query are deterministic and that the generated synthetic data comes from exact conditional distributions.

The denotational semantics (Appendix D, Fig. 21) of the lowered language is mostly standard and resembles the measure-theoretic semantics of GenSQL given in Fig. 5. Terms $\Gamma \vdash e : \sigma_g$ are

(a) **Translating Types and Contexts**

$$\mathcal{T}\{T[?\text{ID}]\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}\} = \textbf{Bag}[(\sigma_1, \ldots, \sigma_n)]$$
$$\mathcal{T}\{M[?\text{ID}]\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}\} = M[(\sigma_1, \ldots, \sigma_n)]$$
$$\mathcal{T}\left\{C^i\{\text{COL}_1 : \sigma_1, \ldots, \text{COL}_n : \sigma_n\}\right\} = C^i[(\sigma_1, \ldots, \sigma_n)]$$
$$\mathcal{T}\{\Gamma, T[?\text{ID}]\{\text{COLS}\}\} = \mathcal{T}\{\Gamma\}, \text{ID} : \mathcal{T}\{T[?\text{ID}]\{\text{COLS}\}\}$$
$$\mathcal{T}\{\Gamma, M[?\text{ID}]\{\text{COLS}\}\} = \mathcal{T}\{\Gamma\}, \text{ID} : \mathcal{T}\{M[?\text{ID}]\{\text{COLS}\}\}$$
$$\mathcal{T}\{\sigma\} = \sigma \quad \mathcal{T}\{[]\} = []$$

(b) **Translating Event and Event-0 Expressions**

$$\mathcal{T}_\delta\{\text{ID.COL}_i = e\} = (\text{ID}, i) = \mathcal{T}_\delta\{e\}$$
$$\mathcal{T}_\delta\{\text{ID.COL}_i > e\} = (\text{ID}, i) > \mathcal{T}_\delta\{e\}$$
$$\mathcal{T}_\delta\{\text{ID.COL}_i < e\} = (\text{ID}, i) < \mathcal{T}_\delta\{e\}$$
$$\mathcal{T}_\delta\{c_1 \wedge c_2\} = \mathcal{T}_\delta\{c_1\} \wedge \mathcal{T}_\delta\{c_2\}$$
$$\mathcal{T}_\delta\{c_1 \vee c_2\} = \mathcal{T}_\delta\{c_1\} \vee \mathcal{T}_\delta\{c_2\}$$

(c) **Translating RowModel Queries**

$$\mathcal{T}_\delta\left\{\textbf{PROBABILITY OF } c_2^0 \textbf{ UNDER ID GIVEN } c^0 \textbf{ GIVEN } c^1\right\} = \exp\left(\textbf{logpdf}_{\text{ID}}(\mathcal{T}_\delta\{c^0\}, \mathcal{T}_\delta\{c^1\}, \mathcal{T}_\delta\{c_2^0\})\right)$$
$$\mathcal{T}_\delta\left\{\textbf{PROBABILITY OF } c_2^1 \textbf{ UNDER ID GIVEN } c^0 \textbf{ GIVEN } c^1\right\} = \textbf{prob}_{\text{ID}}(\mathcal{T}_\delta\{c^0\}, \mathcal{T}_\delta\{c^1\}, \mathcal{T}_\delta\{c_2^1\},)$$
$$\mathcal{T}_\delta\left\{\textbf{GENERATE UNDER ID GIVEN } c^0 \textbf{ GIVEN } c^1 \textbf{ LIMIT } e\right\} = \textbf{replicate}\,(\mathcal{T}_\delta\{e\}, \textbf{simulate}_{\text{ID}}(\mathcal{T}_\delta\{c^0\}, \mathcal{T}_\delta\{c^1\}))$$
$$\mathcal{T}_\delta\left\{t : T[\text{ID}']\{\text{COLS}\} \textbf{ GENERATIVE JOIN ID GIVEN } c^0 \textbf{ GIVEN } c^1\right\} =$$
$$\textbf{mapreduce}\left(\lambda r.\textbf{join}\left(\textbf{singleton}(r), \textbf{simulate}_{\text{ID}}(\mathcal{T}_{\delta[\text{ID}' \mapsto r]}\{c^0\}, \mathcal{T}_{\delta[\text{ID}' \mapsto r]}\{c^1\})\right), \mathcal{T}_\delta\{t\}\right)$$

(d) **Translating Scalar Expressions**

$$\mathcal{T}_\delta\{c\} = c \quad \mathcal{T}_{\delta[\text{ID} \mapsto r]}\{\text{ID.COL}_i\} = \pi_i(r) \quad \mathcal{T}_\delta\{op(e_1, \ldots, e_n)\} = op(\mathcal{T}_\delta\{e_1\}, \ldots \mathcal{T}_\delta\{e_n\})$$

(e) **Translating Table Expressions**

$$\mathcal{T}_\delta\{\textbf{RENAME } t \textbf{ AS ID}\} = \mathcal{T}_\delta\{t\}; \ \mathcal{T}_\delta\{\text{ID}\} = \text{ID}$$
$$\mathcal{T}_\delta\{t_1 \textbf{ JOIN } t_2\} = \textbf{join}(\mathcal{T}_\delta\{t_1\}, \mathcal{T}_\delta\{t_1\})$$
$$\mathcal{T}_\delta\{t : T[\text{ID}]\{\text{COLS}\} \textbf{ WHERE } e\} = \textbf{filter}\,(\lambda r.\mathcal{T}_{\delta[\text{ID} \mapsto r]}\{e\}, \mathcal{T}_\delta\{t\})$$
$$\mathcal{T}_\delta\left\{\begin{array}{l}\textbf{SELECT } \overline{e} \textbf{ AS } \overline{\text{COL}} \\ \textbf{FROM } t : T[\text{ID}]\{\text{COLS}\}\end{array}\right\} = \textbf{map}\,(\lambda r.\mathcal{T}_{\delta[\text{ID} \mapsto r]}\{\overline{e}\}, \mathcal{T}_\delta\{t\})$$

Fig. 7. The lowering transformation $\mathcal{T}\{\cdot\}$.

interpreted as deterministic measurable functions $[\![\Gamma]\!]_{\text{exact}} \to [\![\sigma_g]\!]_{\text{exact}}$. Terms $\Gamma \vdash e : \textbf{Bag}[\sigma_g]$ are interpreted as probability kernels $[\![\Gamma]\!]_{\text{exact}} \to \mathcal{P}\textbf{Bag}([\![\sigma_g]\!]_{\text{exact}})$, where substitution for these programs is interpreted using the Kleisli composition for the point process monad [21]. By induction on the structure of GenSQL programs t in context $\Gamma; \Delta$, we can show (proof in Appendix D.3):

Theorem 4.2 (Exact AMI Guarantee). *Let* $\Gamma, [] \vdash t : T[?\text{ID}]\{\text{COLS}\}$. *Then, for every evaluation of the context* γ,

$$[\![t]\!](\gamma, []) = [\![\mathcal{T}_{[]}\{t\}]\!]_{exact}(\gamma).$$

4.4 Approximate Backend Guarantee

By relying on approximate probabilistic inference, general-purpose PPLs can express large classes of models in which exact inference is intractable. In addition, programmable inference [51] ensures PPLs can support a diverse class of probabilistic models without sacrificing inference quality. We give a new denotational semantics for the lowered language that is appropriate for reasoning in scenarios where the rowModels are implemented in PPLs with approximate Monte Carlo inference.

Monte Carlo algorithms are typically parameterized by a positive integer n specifying a compute budget, such as the number of particles in a sequential Monte Carlo (SMC) algorithm [18] or the number of samples in a Markov Chain Monte Carlo (MCMC) algorithm [64]. The algorithm specifies a sequence of distributions or estimators that converge in some sense to a quantity of interest as $n \to \infty$. In the case of approximate sampling algorithms, most typically the distribution of the generated samples converges weakly to the target distribution, and in the case of parameter estimation the algorithm produces a strongly consistent estimator of the target parameter [18, 64].

Random variable semantics. Our denotational semantics for approximate AMI implementations is motivated by the above discussion. We assume the existence of an ambient probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and associate with each term a sequence of random variables approximating the term

in the exact semantics. As an example, the approximate semantics of $[\![\textbf{map}\ (x.t_1)\ t_2]\!]_{\text{approx}}$ in the context γ and at the "random seed" $\omega \in \Omega$ is given at the n-th approximation by

$$[\![t_2]\!]_{\text{approx}}(\gamma,\omega)_n \gg\!= \left(\lambda S.\textbf{return}\ \left\{\lambda x'.[\![t_1]\!]_{\text{approx}}(\gamma[x \mapsto x'],\omega)_n\ y \;\middle|\; y \in S\right\}\right).$$

This means that we first obtain the n-th approximation of the input t_2, which is a measure on tables, which we then evaluate to obtain a concrete table, S. We then apply the function to each row obtained by the n-th approximation of t_1. The full semantics is given in Appendix D.4, Fig. 23. We assume the following hold:

- For each rowModel identifier $\text{ID} : M[(\sigma_1,\ldots,\sigma_k)]$ in environment γ, event $c^1 : C^1[(\sigma_1,\ldots,\sigma_k)]$, and event-0 $c^0 : C^0[(\sigma_1,\ldots,\sigma_k)]$, there exists a sequence of probability measures $\{\mu^n_{\text{ID};[\![c^0]\!]_{\text{approx}}(\gamma)_n,[\![c^1]\!]_{\text{approx}}(\gamma)_n}\}$ on $\textbf{Bag}\ \prod_{i=1}^k [\![\sigma_i]\!]$;
- for ID, γ, c^1 and c^0 as above, and $c_2^1 : C^1[(\sigma_1,\ldots,\sigma_k)]$, there exists a sequence of real random variables $\{P^n_{\text{ID};[\![c^0]\!]_{\text{approx}}(\gamma)_n,[\![c^1]\!]_{\text{approx}}(\gamma)_n,[\![c_2^1]\!]_{\text{approx}}(\gamma)_n}\}$ which takes values in $[0,1]$ $\mathbb{P}$-almost surely;
- for ID, γ, c^1 and c^0 as above, and $c_2^0 : C^0[(\sigma_1,\ldots,\sigma_k)]$, there exists a sequence of real random variables $\{L^n_{\text{ID};[\![c^0]\!]_{\text{approx}}(\gamma)_n,[\![c^1]\!]_{\text{approx}}(\gamma)_n,[\![c_2^0]\!]_{\text{approx}}(\gamma)_n}\}$.

These random sequences represent the sequences of approximations produced by the implementation of the AMI. In general, for a given term t the convergence of sequences associated with its sub-terms do not imply that the sequence associated with t converges. For instance, consider evaluating the following query in an appropriate context (γ,δ):

$$\textsc{select}\ \text{ID.COL}\ \textsc{from}\ \text{ID}\ \textsc{where}\ \text{ID.COL} \leq (\ \textsc{probability of}\ \text{ID}'.\text{COL}' = 7).$$

If the value of the term $\textsc{probability of}\ \text{ID}'.\text{COL}' = 7$ is approximated, even if we can make this approximation arbitrarily accurate, the output of the query need not converge. For example, if the table ID contains a row in which the value of COL is exactly $[\![\ \textsc{probability of}\ \text{ID}'.\text{COL}' = 7]\!](\gamma,\delta)$ but the approximation converges to the true value from below, this row will not be included in the query result no matter the accuracy of the approximation. Intuitively, this arises from the fact that the indicator functions of half intervals are not continuous.

In order for the lowered queries to denote asymptotically sound estimators for the original queries, we require that the implementation of the AMI methods are asymptotically sound, and write $\lim_n \gamma_n$ to denote an evaluation of the context γ in which each random variable is replaced by its limit as $n \to \infty$. In Appendix D.4, we formalize the notions of *safe* queries and asymptotically *sound* AMI implementations and details of the proofs. We then give the following guarantee.

Theorem 4.3 (Consistent AMI Guarantee). *Let* $\Gamma, [] \vdash t : T[?\text{ID}]\{\text{COLS}\}$ *be a safe query and suppose the AMI methods have asymptotically sound implementations. Then, for every evaluation of the context* γ, $\mathbb{P}$*-almost surely*

$$\lim_n([\![\mathcal{T}_{[]}\ \{t\}]\!]_{approx})(\gamma) = [\![t]\!](\lim_n \gamma, []).$$

5 EVALUATION

The performance of an open-source Clojure implementation of GenSQL is evaluated against other systems that have similar capabilities. We test runtime, the effect of optimizations, and runtime overhead of our system over alternative implementations of the same task. Experiments were run on an Amazon EC2 C6a instance with Ubuntu 22.04, 4 vCPUs and 8.0 GiB RAM.

The probabilistic models used in the evaluation are obtained using probabilistic program synthesis [74, Chapter 3]. Each model is an ensemble of "MultiMixture" probabilistic programs [69, Section 6], which are posterior samples from the CrossCat model class [50], generated using ClojureCat [16]. An ensemble of 10 probabilistic programs is used in Section 5.1 and 12 programs in Section 5.2.

Table 1. Runtime (sec) comparison of GenSQL and BayesDB [52] on 10 benchmark queries (Appendix F) for evaluating probability densities of measure-zero events.

	GenSQL (ClojureCat Backend)	BayesDB (CGPM Backend)	Speedup
Q1	0.24 ± 0.03	0.59 ± 0.16	2.5x
Q2	0.29 ± 0.03	1.15 ± 0.2	4.0x
Q3	0.43 ± 0.06	1.72 ± 0.28	4.0x
Q4	0.48 ± 0.06	2.25 ± 0.27	4.7x
Q5	0.57 ± 0.07	2.68 ± 0.36	4.7x
Q6	0.33 ± 0.06	0.55 ± 0.23	1.7x
Q7	0.49 ± 0.05	1.53 ± 0.26	3.1x
Q8	0.46 ± 0.03	1.81 ± 0.21	3.9x
Q9	0.37 ± 0.03	2.51 ± 0.32	6.8x
Q10	0.45 ± 0.04	2.87 ± 0.39	6.4x
Mean	0.41 ± 0.11	1.77 ± 0.83	4.3x

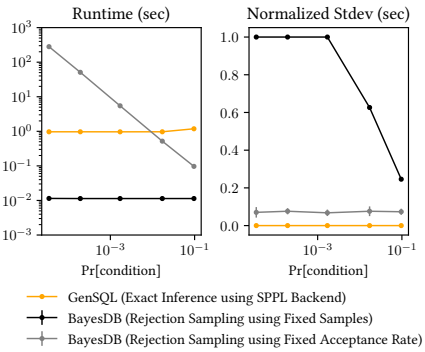


Fig. 8. Runtime/stdev comparison of GenSQL and BayesDB [52] on 5 benchmark queries for evaluating probabilities of positive measure events.

5.1 Performance and Usability

Runtime comparison. Table 1 compares the runtime on 10 benchmark queries (Appendix F) adapted from Charchut [16, Tables 4.2 and 4.3] using GenSQL (with the ClojureCat backend) and BayesDB (with the CGPM backend [66]) for evaluating exact probability densities. Section 5 compares the runtime and standard deviation for computing the probabilities of positive measure events. GenSQL (with the SPPL backend [73]) delivers exact solutions, whereas BayesDB delivers approximate solutions using rejection sampling. Two rejection strategies in BayesDB are shown in Section 5: a fixed number of samples (faster but higher variance) or a fixed acceptance rate (slower but lower variance), which both are inferior to exact solutions from GenSQL.

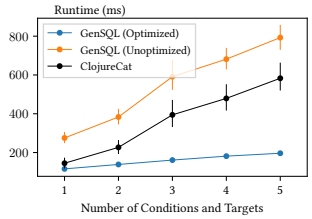
The performance gains in GenSQL are due to three main reasons: the ClojureCat backend is faster than the CGPM backend in BayesDB, GenSQL has optimizations (discussed below) that exploit repetitive computations, and GenSQL itself is implemented in Clojure, a performant language.

Optimizations and system overhead. GenSQL leverages two classes of optimizations: caching (of the likelihood queries and conditioned models) and exploiting independence relations between columns. The latter allows us to simplify a query such as PROBABILITY OF ID.$x > 42$ UNDER ID GIVEN ID.$y = 17$ to the semantically equivalent query PROBABILITY OF ID.$x > 42$ UNDER ID if the columns x and y are independent. Appendix B gives a detailed account of the optimizations.

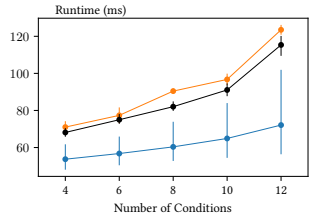
In Fig. 9, the unoptimized GenSQL queries have a 1.1-1.6x overhead compared to the pure ClojureCat baseline. The optimizations reduce the overhead and can sometimes drastically improve performance, while caching significantly reduces the variance in the runtime of the queries. In Fig. 9b, the effect of the independence optimization varies between replicates, as these are different CrossCat model samples, which explains the higher variance in query runtime.

Code comparison. Figure 10 compares the code required in GenSQL, pure Python using SPPL [73], and pure Clojure using ClojureCat [16], for a conditional probability query. Figure 10a shows how GenSQL gains clarity by specializing in data that comes from database tables. In contrast, both SPPL and ClojureCat require users to hand-write the looping/mapping over the data, which is error prone. For instance, the code in Fig. 10c will crash if the table has missing values. In Fig. 10b, ClojureCat requires conditions to be maps. Users can decide if they encode columns with strings, symbols, or keywords. If this choice does not align with the key type returned by the CSV reader, the query will run but conditioning will result in a null-op.

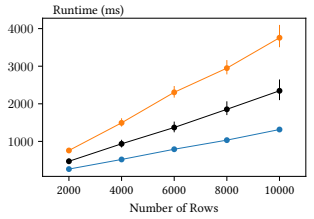
In Appendix A.1, we compare a single line query on a conditioned model in GenSQL to the equivalent code in Scikit-learn [60] on the Iris data from the UCI ML repository. The model querying



(a) Varying number of conditions and targets in the **PROBABILITY OF** queries shown in Table 1.



(b) Varying number of conditions in **GIVEN** clause for **GENERATE UNDER** queries (caching does not apply).



(c) Varying number of rows in a data table used in the **FROM** clause of **SELECT** with a **PROBABILITY OF** query.

Fig. 9. Runtime comparison between GenSQL (ClojureCat backend) and raw ClojureCat [16].

```
SELECT PROBABILITY OF
    Period_minutes = 98.6,
    Type_of_Orbit = "Sun-Synchro.",
    Contractor = "Lockheed Martin"
  UNDER model GIVEN Country_of_Operator, Launch_Mass_kg
FROM data
```

(a) GenSQL

```
(let
  [event {:Period_minutes 98.6
          :Type_of_Orbit "Sun-Synchro."
          :Contractor "Lockheed Martin"}
   cols-in-condition [:Country_of_Operator,
                      :Launch_Mass_kg]]
  (->> data
       (map #(select-keys cols-in-condition %))
       (mapv #(exp (gpm/logpdf model event %)))))
```

(b) ClojureCat (Clojure)

```
event = {
    "Period_minutes": 98.6,
    "Type_of_Orbit": "Sun-Synchro.",
    "Contractor": "Lockheed Martin",
}
cols_in_condition = ["Country_of_Operator", "Launch_Mass_kg"]

targets = {sppl.transforms.Identity(k): v for k, v in event.
  items()}
constraints = [
    {
        sppl.transforms.Identity(column): value
        for column, value in row.items()
        if column in cols_in_condition
    }
    for _, row in data.iterrows()
]
print([exp(spe.constrain(constraint).logpdf(targets)) for
  constraint in constraints])
```

(c) SPPL (Python)

Fig. 10. Comparison of GenSQL, ClojureCat [16], and SPPL [73] code for a conditional probability query.

a	b	c
a_0	b_0	c_0
a_1	b_1	c_1
a_0	b_0	c_0
a_0	b_1	c_0
...	...	...

(a) Table `foo`

a	b	x	y	z
a_0	b_1	4.2	4.1	0.6
a_1	b_1	-4.4	-5.4	0.2
a_1	b_0	-3.7	-6.2	0.5
a_1	b_1	-6.2	-4.2	0.1
...	...	...	...	...

(b) Table `bar` to build model

```
SELECT * FROM foo GENERATIVE JOIN bar_model GIVEN *
```

(c) GenSQL

```
# escape into SQLite
ALTER TABLE bar ADD COLUMN c TEXT
UPDATE bar SET c = 'placeholder'
INSERT INTO bar SELECT
  a,b, NULL AS x, NULL AS y, NULL AS z, c
  FROM foo
```

```
INFER a, b, c, x, y, z
FROM bar WHERE rowid > [num rows in foo]
```

(d) BayesDB

Fig. 11. Comparison of GenSQL and BayesDB [52] code. The latter does not support **GENERATIVE JOIN**.

alone in Scikit-learn is more than 50 lines long and clearly error prone, and we find that GenSQL offers a significant advantage in simplicity over such baselines.

Code comparison with BayesDB. Figure 11 shows GenSQL and its closest relative, BayesDB [52], on a **GENERATIVE JOIN** query on synthetic data. The GenSQL code is more concise and simpler than BayesDB's code, which is possible due to the language abstractions for manipulating models. In BayesDB, the user must exit to SQL and hand-code column manipulations to fit the expected fixed pattern to query a model. Section 6 provides a detailed comparison of GenSQL and BayesDB.

5.2 Case Studies on Real World Data

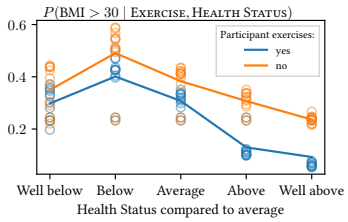
We present two case studies to demonstrate the application of GenSQL to real-world problems: one in medicine (clinical trial data) and one in synthetic biology (wetlab data). The datasets can be costly to obtain and researchers are interested in understanding and analyzing their data.

```
SELECT
  BMI, exercise, health_status, age, smoker
FROM clinical_trial_records
WHERE
  (BMI > 20.3) AND
  (BMI < 38.4) AND
  (PROBABILITY OF BMI UNDER clinical_trial_model GIVEN *) < 0.01
```

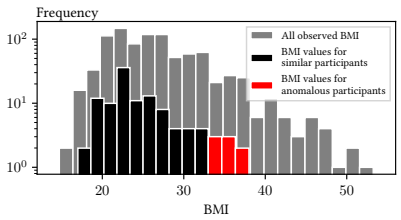
(a) Find anomalous BMI values, defined by $P(\text{BMI} \mid *) < 0.01$ for all BMI values within the 5th (20.3) and 95th (38.4) percentiles in the US.

BMI	exercise	health_status	age	smoker
38	yes	Above average	40-49	never
33	yes	Well above average	30-39	never
33	yes	Well above average	50-59	never
35	yes	Above average	40-49	never
37	yes	Well above average	40-49	never
33	yes	Well above average	50-59	never
36	yes	Above average	30-39	former
35	yes	Above average	40-49	former

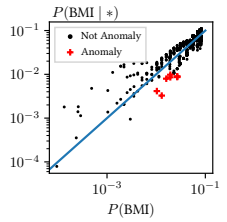
(b) Result from the query in (a). Eight anomalous participants are returned: all are clinically obese but report above-average health status and exercise.



(c) Conditonal probabilites encoded by the underlying model.



(d) Compare anomalous BMI values to normal ones.



(e) Compare conditional and marginal BMI.

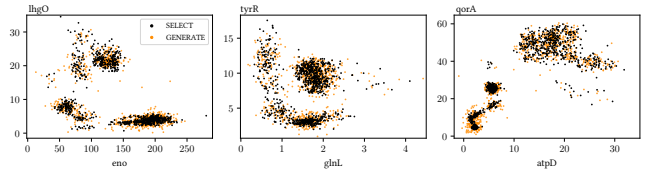
Fig. 12. Case study: Anomaly detection in clinical trials.

In the first case, we show how anomaly detection in GenSQL can be used to check for probable mislabelling of the data. The anomalous rows can also be investigated further to understand the reasons for the anomaly. In the second case, we show how GenSQL can be used to generate accurate synthetic data, capturing the complex relationships between different host genes and experimental conditions. Capturing these relationships with the model helps predict whether a certain experimental condition or modification of the genome has cascading downstream effects through the interrelations between the genes. Such effects can render the cell toxic and kill the bacterium, leading to a failed experiment. The virtual wet lab allows researchers to check for such effects before running costly experiments in the real world.

Anomaly detection in clinical trials. The (BEAT19) clinical trial [87] contains data about COVID-19 and records behavior, environment variables, and treatments. Figure 12a shows a query used for anomaly detection [15]. For each row, it computes the model likelihood of the value BMI given the other values of the row. The trial participants labeled anomalous (Fig. 12b) all report above-average or well-above-average health and that they exercise, while meeting the World Health Organization's definition of clinical obesity [83]. Fig. 12d compares the overall population in the trial (grey), anomalous individuals (red), and the subset of the population with the same behavioral covariates (exercise, health status, etc.) as the anomalous individuals (black). For similar individuals, the data generally suggests a lower BMI. We can also compare the marginal and the conditional probability of BMI values in the table of clinical trial records (Fig. 12d). Anomalous data (red) is lower than the diagonal line, highlighting the "contextualization" of BMI values that happens by conditioning the models: the BMI values are less likely given the context of the other values in the row, while not necessarily extreme. To demonstrate this effect, we first apply a WHERE filter that removes BMI values outside of the 5th and the 95th percentile, excluding one-dimensional extreme values (Fig. 12a). We then compute the conditional probabilities of the BMI values in each row for the remaining data and return anomalies. Fig. 12c shows the posterior predictive over the ensemble of models (line) and for each individual model (dots) for a BMI above 30 given exercise and health status.

```
SELECT * FROM gene_expression_data

GENERATE UNDER gene_expression_model
LIMIT 1000
```

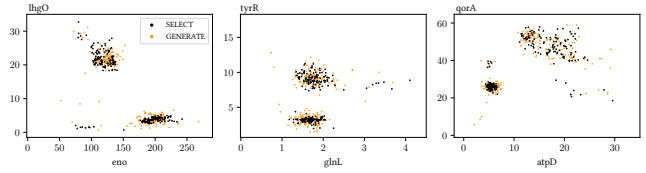
(a) Overall population



(b) 6 genes from the overall population.

```
SELECT * FROM gene_expression_data
WHERE Ara. = "not added" AND IPTG = "added"

GENERATE UNDER gene_expression_model
  GIVEN Ara. = "not added" AND IPTG = "added"
LIMIT 1000
```

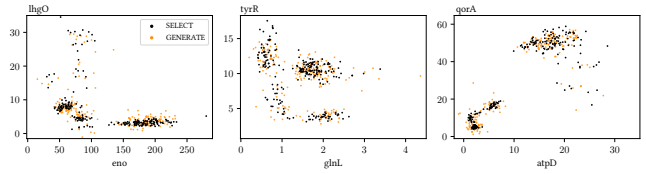
(c) Add IPTG but not Arabinose



(d) 6 genes from (c).

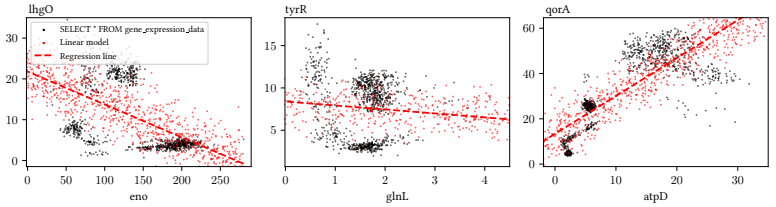
```
SELECT * FROM gene_expression_data
WHERE Ara. = "added" AND IPTG = "added"

GENERATE UNDER gene_expression_model
  GIVEN Ara. = "added" AND IPTG = "added"
LIMIT 1000
```

(e) Add both IPTG and Arabinose

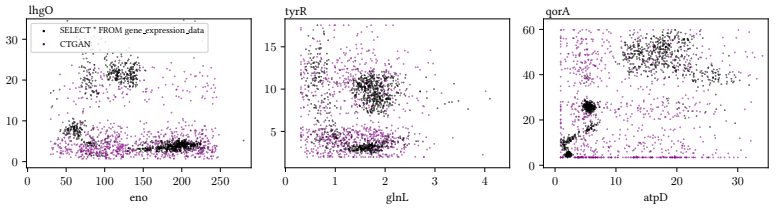


(f) 6 genes from (e).

Fig. 13. Case study: Conditional synthetic data generation for a virtual wet lab.



(a) Real and generated data from bivariate linear models (compare to Fig. 13b).



(b) Comparison of real and generated data from CTGAN (compare to Fig. 13b).

Fig. 14. Linear models and conditional generative adversarial networks (CTGAN [88]) produce less accurate synthetic virtual wet lab data as compared to the synthetic data from GenSQL shown in Fig. 13b. In (b), the default model and inference parameters in the open source implementation of CTGAN is used.

Conditional synthetic data generation for virtual wet lab. Figure 13 shows synthetic gene expression data generated using GenSQL, given a dataset from an experiment testing genetic circuits [56] in *Escherichia coli*. The synthetic data aligns with the overall population characteristics (Figs. 13a and 13b) and accurately reflects the outcomes of actual experimental interventions (Figs. 13c to 13f). In synthetic biology, the prospect of implementing genetic circuits has fundamental implications for medical device engineering [53], bio-sensing [82] and environmental biotechnology [89]. These circuits require input which is typically provided by adding inducer substances to

the culture mediums where the organisms are grown. Our figures show the effect of adding two such inducer substances, Arabinose and IPTG on 6 different host genes.

Producing standard RNA sequencing data can be costly [48], especially for new, engineered organisms that are not mass-produced. When it is produced, RNA sequencing will yield measurements for gene expressions for thousands of annotated host genes [9]. These genes are highly interrelated, and knowledge of the relations is only partially available [44]. Thus, the application of generative models to these data presents a challenging high-dimensional modeling problem, further compounded by the inherent non-linearity in the data, as illustrated in Figs. 14a and 14b.

The most popular approach to modeling gene expression data is linear regression [22], as models are easy to interpret and readily available in data analysis libraries. For non-numerical data, linear regression requires analysis-specific re-coding of discrete values. That aside, the low capacity of the model means that it fails to faithfully reproduce in the actual wet lab data, as shown in Fig. 14a. Conditional generative adversarial networks(CTGAN) [88], though more complex, are also an appropriate baseline because they are domain-general and effective at modeling multivariate, heterogeneous data. However, GANs are hard to interpret and as RNA sequencing data acquisition is so costly, the number of available training examples (943) renders it unsuitable for CTGANs. Fig. 14b depicts this model class failing to accurately model the gene expression data.

6 RELATED WORK

Probabilistic databases. Probabilistic databases systems [79, 81] develop efficient algorithms for inference queries on discrete distributions over databases, often based on variants of weighted model counting, for which hardness complexity results were shown and algorithms were developed for tractable cases and efficient approximations. Cambronero et al. [13] integrate probabilities into a relational database system to support imputation, while Hilprecht et al. [39] use probabilistic circuits to improve query performance. Jampani et al. [42] use probabilistic databases to support random data generation and simulation. Cai et al. [12] provides Gibbs sampling support in the space of database tables to a SQL-like language, enabling bayesian machine learning workload such as linear regression or latent Dirichlet allocation. These languages are typically extensions to SQL or relational algebra but with limited support for probabilistic models, which they tradeoff for performance. Schaechtle et al. [76] presents a preliminary design for extending SQL to support probabilistic models of tabular data. Our work differs in that it presents (1) a formalization of the system; (2) a denotational semantics; (3) soundness guarantees for the system; (4) a unified interface that probabilistic models implement; (5) a lowering transform and target lowering language; (6) an extensive performance evaluation; and (7) two new case studies on real-world data.

Semantics of probabilistic databases. Bárány et al. [3] and Grohe et al. [34] give a semantic account to probabilistic databases by giving a probabilistic semantics and guarantees to an extension of Datalog. Dash and Staton [21] give a monadic account and denotational semantics for measurable queries in probabilistic databases. Their semantics of SQL-like expressions inspired the semantics of our table expressions. Grohe and Lindner [35] established a formal framework for reasoning about infinite probabilistic databases. Benzaken and Contejean [5] formalized the semantics of SQL in Coq while Borya [11] formalized relational algebra and a SQL-like syntax using a model checker.

Probabilistic program synthesis. GenSQL has been designed with the possibility to leverage powerful probabilistic program synthesis techniques based on Bayesian [1, 50, 69] or non-Bayesian [17, 30, 36, 41, 57] probabilistic model discovery. The AMI provides a unifying approach to expressing powerful Bayesian inference workflows in these probabilistic programs using a high-level SQL-like language. Extending the interface to handle synthesized models of time series [70, 72] and/or relational data [71] is a promising avenue for future work.

Probabilistic programming systems. While we used a Clojure version [16] of CrossCat [50] in our experiments, our system supports any probabilistic program that satisfies the rowModel interface. We can thus reuse models written in the variety of PPLs developed in the literature, such as models written in languages supporting approximate inference [8, 14, 20, 26, 54, 75, 85] and exact inference [27, 40, 73, 90]. Our model interface is inspired by the SPPL interface [73] and the CGPM interface [68]. Gordon et al. [33] propose a probabilistic programming system using a functional syntax similar to the stochastic lambda calculus, specialized to inference over relational databases, implemented on top on Infer.net. It can perform inference tasks such as linear regression and querying for missing values which enable data imputation, classification, or clustering. Borgström et al. [10] present a probabilistic DSL and semantics for regression formulas in the style of the formula DSL in R. Domain-specific PPLs for tabular data have also been designed to solve tasks such as data cleaning [46, 63].

BayesDB. Although BayesDB [52] was motivated by similar goals as GenSQL, GenSQL introduces novel semantics concepts and soundness theorems that BayesDB did not. GenSQL also improves upon BayesDB in terms of expressiveness and performance, as shown in Section 5. For example, GenSQL queries can be nested and interleaved with SQL, and also combine results from multiple models. GenSQL also provides an exact inference engine for a broad class of sum-product probabilistic programs [73]. BayesDB, on the other hand, has interesting features that GenSQL does not yet support such as iterating over model and columns (e.g. to find pairs of columns with the highest mutual information) [67] and similarity search between rows [65]. BayesDB also has a "meta-modeling" DSL [66] for composing probabilistic programs from various sources.

Automated Machine Learning. Several systems [6, 23, 43, 45, 58, 80] have been developed to automate the use of discriminative machine learning methods for analyzing tabular data. Unlike GenSQL, they do not support the use of generative probabilistic programs for tabular data satisfying a unified interface (for sampling, conditioning, and evaluating probabilities or densities) which enables a single model to be reused across many different tasks.

7 CONTRIBUTIONS

GenSQL specializes probabilistic programming languages to applications with tabular data. It is differentiated from general purpose PPLs in three main ways:

- **Through the AMI, GenSQL enables multi-language workflows.** Users from different domains and with different expertise should be able to use probabilistic models for their queries without having to learn all the details of the PPL in which the model is written. The AMI enables this separation of concerns by providing a well-specified interface. It enables the integrating probabilistic models of tabular data in different languages, as it can be implemented in either a general-purpose or domain-specific PPL (Appendix C). There is no standard way to jointly query models in different PPLs or use the result of a query in one language against a model in another language. As different PPLs focus on different workloads, users of GenSQL can work with several models written in different PPLs. GenSQL thus provides a natural multi-language workflow, and our experiments already use multiple backends (Gen.clj, SPPL, and ClojureCat).
- **GenSQL enables declarative querying.** No current PPL offers a simple declarative syntax for evaluating complex queries (e.g., containing elaborate joins and nested selects) interleaving calls on probabilistic models and database tables. A number of PPLs provide declarative syntax for specifying and conditioning models, but the user must decide which operations on what conditional distributions to evaluate and then manually combine the results of these operations. GenSQL relieves the users of such concerns, reducing the chances of programming errors.

• **GenSQL enables reusable performance optimizations.** Widely used database management systems (DBMS) have been optimized by many engineer-hours of effort over several decades. These optimizations are highly reusable because they are independent of the application domain and specific languages that the DBMS interfaces with. GenSQL enables analogous optimizations for workloads that interleave ordinary database queries with probabilistic inference and generative modeling. GenSQL's optimizations can carry over to many domains and workflows, avoiding the need for project-specific performance optimizations involving probabilistic models of tabular data.

We see two interesting avenues for GenSQL to impact database applications and design.

Integration of GenSQL with database management systems (DBMS). First, GenSQL could serve as a query language, allowing users to query generative models of tabular data directly from the DBMS. One use case of rapidly increasing practical importance is querying synthetic data, generated on the fly to meet user-specified privacy-utility trade-offs, instead of querying real data that cannot be shared due to privacy constraints. Other potential applications for synthetic data include testing, performance tuning, and sensitivity analysis of end-to-end data analysis workflows. In all these settings, GenSQL implementations could also draw on performance engineering innovations from DBMS engines, optimized further using the generative models themselves (e.g., to reduce variance for stratified sampling approximations to SQL aggregates [2]).

Modularized development of queries and models. GenSQL introduces abstractions that isolate query developers and query users from model developers. This separation of concerns is analogous to the physical data independence property achieved by relational databases [19]. Most database users do not need to know the details of how data is stored and indexed to be able to query it efficiently, but some experts do understand how to tune indices to ensure that databases meet the necessary performance constraints. Most GenSQL users need not be experts on the details of the algorithms, modeling assumptions, and software pipelines that produced the underlying generative models. Expert statisticians and generative modelers can still ensure the models are of sufficient quality and tune trade-offs between performance, maintenance costs, and accuracy, improving models without invalidating user workflows. With GenSQL, both typical users and experts can more easily and interactively query generative models to test their validity, both qualitatively and quantitatively. This division of responsibility between users, generative modelers, and probabilistic programming system developers could potentially help our society more safely and productively broaden the deployment of generative models for tabular data.

DATA-AVAILABILITY STATEMENT

An artifact providing a version of GenSQL, and reproducing our experiments, is available [77].

ACKNOWLEDGMENTS

MIT contributors would like to acknowledge support from DARPA, under the Machine Common Sense (MCS) program (Award ID: 030523-00001) Synergistic Discovery and Design (SD2) program (Contract No. FA8750-17-C-0239), and Compositionally Organized Learning To Reason About Novel Experiences (COLTRANE) grant (Contract No. 140D0422C0045); and unrestricted gifts from Google, an anonymous donor, and the Siegel Family Foundation. F. Saad acknowledges support from the National Science Foundation (Award ID: 2311983). The authors wish to thank the anonymous referees for their valuable comments and suggestions. We have also benefited greatly from conversations with and feedback from Pierre Glaser, Younesse Kaddar, João Loula, Timothy J. O'Donnell, and Fabian Zaiser.

REFERENCES

[1] Ryan P. Adams, Hanna Wallach, and Zoubin Ghahramani. 2010. Learning the Structure of Deep Sparse Graphical Models. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 1–8.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Stoica Ion. 2013. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/2465351.2465355

[3] Vince Bárány, Balder Ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative Probabilistic Programming with Datalog. *ACM Transactions on Database Systems* 42, 4 (2017), 1–35. https://doi.org/10.1145/3132700

[4] Luc Bauwens, Michel Lubrano, and Jean-Francois Richard. 2000. *Bayesian Inference in Dynamic Econometric Models*. Oxford University Press, Oxford, UK.

[5] Véronique Benzaken and Evelyne Contejean. 2019. A Coq Mechanised Formal Semantics For Realistic SQL Queries: Formally Reconciling SQL and Bag Relational Algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, New York, NY, USA, 249–261. https://doi.org/10.1145/3293880.3294107

[6] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. 2015. Hyperopt: A Python Library for Model Selection And Hyperparameter Optimization. *Computational Science & Discovery* 8, 1 (2015), 014008. https://doi.org/10.1088/1749-4699/8/1/014008

[7] Patrick Billingsley. 1995. *Probability and Measure* (3rd ed.). John Wiley & Sons, New York.

[8] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 1 (2019), 973–978.

[9] Frederick R. Blattner et al. 1997. The Complete Genome Sequence of Escherichia Coli K-12. *Science* 277, 5331 (1997), 1453–1462. https://doi.org/10.1126/science.277.5331.1453

[10] Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo, Adam Scibior, and Marcin Szymczak. 2016. Fabular: Regression Formulas as Probabilistic Programming. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 271–283. https://doi.org/10.1145/2837614.2837653

[11] Joachim Borya. 2023. *Formalisation of Relational Algebra and a SQL-like Language with the RISCAL Model Checker*. Technical Report 23-06. Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz. https://doi.org/10.35011/risc.23-06

[12] Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J. Haas, and Christopher Jermaine. 2013. Simulation of Database-Valued Markov Chains Using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 637–648. https://doi.org/10.1145/2463676.2465283

[13] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. 2017. Query Optimization for Dynamic Imputation. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1310–1321. https://doi.org/10.14778/3137628.3137641

[14] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

[15] CDC. 2022. BMI Percentile Calculator: Body Mass Indexes in the United States. https://dqydj.com/bmi-percentile-calculator-united-states

[16] Nicholas G. Charchut. 2020. *Implementation of a Cross-Platform Automated Bayesian Data Modeling System*. Master's thesis. Massachusetts Institute of Technology, Cambridge, MA.

[17] Sarah Chasins and Phitchaya M. Phothilimthana. 2017. Data-driven synthesis of full probabilistic programs. In *Proceedings of the 29th International Conference on Computer Aided Verification*. Springer, Cham, 279–304. https://doi.org/10.1007/978-3-319-63387-9_14

[18] Nicolas Chopin, Omiros Papaspiliopoulos, et al. 2020. *An Introduction to Sequential Monte Carlo*. Springer, Cham. https://doi.org/10.1007/978-3-030-47845-2

[19] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[20] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

[21] Swaraj Dash and Sam Staton. 2021. Monads for Measurable Queries in Probabilistic Databases. arXiv:2112.14048

[22] Sebastian Eck and Wolfgang Stephan. 2008. Determining the Relationship of Gene Expression and Global mRNA Stability in Drosophila Melanogaster And Escherichia Coli Using Linear Models. *Gene* 424, 1-2 (2008), 102–107. https://doi.org/10.1016/j.gene.2008.07.033

[23] 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:2003.06505
[24] D. H. Fremlin. 2001. *Measure Theory*. Vol. 2. Torres Fremlin, Colchester, England.
[25] Jonah Gabry, Daniel Simpson, Aki Vehtari, Michael Betancourt, and Andrew Gelman. 2019. Visualization in Bayesian Workflow. *Journal of the Royal Statistical Society Series A: Statistics in Society* 182, 2 (2019), 389–402. https://doi.org/10.1111/rssa.12378
[26] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 1682–1690.
[27] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Proceedings of the 28th International Conference on Computer Aided Verification*. Springer, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4
[28] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. 2014. *Bayesian Data Analysis* (3rd ed.). CRC Press, Boca Raton. https://doi.org/10.1201/9780429258411
[29] Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. 2020. Bayesian Workflow. arXiv:2011.01808
[30] Robert Gens and Pedro Domingos. 2013. Learning the Structure of Sum-Product Networks. In *Proceedings of the 30th International Conference on Machine Learning*. PMLR, Norfolk, MA, USA, 873–880.
[31] Alan Genz and Frank Bretz. 2009. *Computation of Multivariate Normal and t Probabilities*. Lecture Notes in Statistics, Vol. 195. Springer, Cham. https://doi.org/10.1007/978-3-642-01689-9
[32] Michele Giry. 2006. A Categorical Approach To Probability Theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference*. Springer, Cham, 68–85. https://doi.org/10.1007/BFb0092872
[33] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgström, and John Guiver. 2014. Tabular: A Schema-Driven Probabilistic Programming Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 321–334. https://doi.org/10.1145/2535838.2535850
[34] Martin Grohe, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Peter Lindner. 2022. Generative Datalog with Continuous Distributions. *J. ACM* 69, 6 (2022), 1–52. https://doi.org/10.1145/3559102
[35] Martin Grohe and Peter Lindner. 2022. Infinite Probabilistic Databases. *Logical Methods in Computer Science* 18, 1, Article 34 (2022), 43 pages. https://doi.org/10.46298/LMCS-18(1:34)2022
[36] Roger Grosse, Ruslan Salakhutdinov, William Freeman, and Joshua B. Tenenbaum. 2012. Exploiting compositionality to explore a large space of model structures. In *Proceedings of the 28th Annual Conference on Uncertainty in Artificial Intelligence*. AUAI Press, Puyallup, WA, USA, 306–315.
[37] Allen Hatcher. 2002. *Algebraic Topology*. Cambridge University Press, Cambridge, UK.
[38] Miguel A. Hernán and James M. Robins. 2006. Estimating Causal Effects from Epidemiological Data. *Journal of Epidemiology & Community Health* 60, 7 (2006), 578–586. https://doi.org/10.1136/jech.2004.029496
[39] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005. https://doi.org/10.14778/3384345.3384349
[40] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 140 (2020), 31 pages. https://doi.org/10.1145/3133904
[41] Irvin Hwang, Andreas Stuhlmüller, and Noah D Goodman. 2011. Inducing Probabilistic Programs by Bayesian Program Merging. arXiv:1110.5667
[42] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Perez, Chris Jermaine, and Peter J. Haas. 2011. The Monte Carlo Database System: Stochastic Analysis Close to the Data. *ACM Transactions on Database Systems* 36, 3 (2011), 1–41. https://doi.org/10.1145/2000824.2000828
[43] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, New York, NY, USA, 1946–1956. https://doi.org/10.1145/3292500.3330648
[44] Ingrid M Keseler, Socorro Gama-Castro, Amanda Mackie, Peter E Midford, Alan J Wolfe, Julio Collado-Vides, Ian T Paulsen, and Peter D Karp. 2021. The EcoCyc Database in 2021. *Frontiers in Microbiology* 12 (2021), 711077. https://doi.org/10.3389/fmicb.2021.711077
[45] Erin LeDell and Sebastien Poirier. 2020. H2O AutoML: Scalable Automatic Machine Learning. In *Proceedings of the 7th ICML Workshop on AutoML*. AutoML-Conf, 16 pages.
[46] Alexander Lew, Monica Agrawal, David Sontag, and Vikash Mansinghka. 2021. PClean: Bayesian Data Cleaning at Scale With Domain-Specific Probabilistic Programming. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 1927–1935.
[47] Alexander K Lew, Matin Ghavamizadeh, Martin C Rinard, and Vikash K Mansinghka. 2023. Probabilistic Programming with Stochastic Probabilities. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1708–1732. https:

//doi.org/10.1145/3591290
[48] Brian K. Lohman, Jesse N. Weber, and Daniel I. Bolnick. 2016. Evaluation Of Tagseq, A Reliable Low-Cost Alternative for RNA Seq. *Molecular Ecology Resources* 16, 6 (2016), 1315–1321. https://doi.org/10.1111/1755-0998.12529
[49] David JC MacKay et al. 1998. Introduction to Gaussian Processes. *NATO ASI series F computer and systems sciences* 168 (1998), 133–166.
[50] Vikash Mansinghka, Patrick Shafto, Eric Jonas, Cap Petschulat, Max Gasner, and Joshua B. Tenenbaum. 2016. CrossCat: A Fully Bayesian Nonparametric Method for Analyzing Heterogeneous, High Dimensional Data. *Journal of Machine Learning Research* 17, 138 (2016), 1–49.
[51] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 603–616. https://doi.org/10.1145/3192366.3192409
[52] Vikash K. Mansinghka, Richard Tibbetts, Jay Baxter, Pat Shafto, and Baxter Eaves. 2015. BayesDB: A Probabilistic Programming System for Querying the Probable Implications of Data. arXiv:1512.05006
[53] Maysam Mansouri and Martin Fussenegger. 2022. Therapeutic Cell Engineering: Designing Programmable Synthetic Genetic Circuits in Mammalian Cells. *Protein & Cell* 13, 7 (2022), 476–489. https://doi.org/10.1007/s13238-021-00876-1
[54] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Sontag Kolobov. 2005. BLOG: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA, USA, 1352–1359.
[55] Kevin P. Murphy. 2022. *Probabilistic Machine Learning: An Introduction*. MIT Press, Cambridge, MA.
[56] Alec A. K. Nielsen, Bryan S. Der, Jonghyeon Shin, Prashant Vaidyanathan, Vanya Paralanov, Elizabeth A. Strychalski, David Ross, Douglas Densmore, and Christopher A. Voigt. 2016. Genetic circuit design automation. *Science* 352, 6281 (2016), aac7341. https://doi.org/10.1126/science.aac7341
[57] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 208–217. https://doi.org/10.1145/2737924.2737982
[58] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. 2016. Automating Biomedical Data Science Through Tree-Based Pipeline Optimization. In *Applications of Evolutionary Computation*. Springer, Cham, 123–137. https://doi.org/10.1007/978-3-319-31204-0_9
[59] Judea Pearl. 1988. *Probabilistic Reasoning In Intelligent Systems: Networks Of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, USA.
[60] Fabian Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning research* 12 (2011), 2825–2830.
[61] MIT Probabilistic Computing Project. 2024. Gen.clj. https://github.com/probcomp/Gen.clj
[62] Tom Rainforth. 2018. Nesting Probabilistic Programs. In *Proceedings of the 34th Annual Conference on Uncertainty in Artificial Intelligence*. AUAI Press, Puyallup, WA, USA, 10 pages. arXiv:1803.06328
[63] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631
[64] Christian P. Robert and George Casella. 2004. *Monte Carlo Statistical Methods* (2 ed.). Springer, Cham. https://doi.org/10.1007/978-1-4757-4145-2
[65] Feras Saad, Leonardo Casarsa, and Vikash Mansinghka. 2017. Probabilistic Search for Structured Data via Probabilistic Programming and Nonparametric Bayes. arXiv:1704.01087
[66] Feras Saad and Vikash Mansinghka. 2016. Probabilistic Data Analysis with Probabilistic Programming. arXiv:1608.05347
[67] Feras Saad and Vikash Mansinghka. 2017. Detecting Dependencies in Sparse, Multivariate Databases using Probabilistic Programming and Non-Parametric Bayes. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 632–641.
[68] Feras Saad and Vikash K. Mansinghka. 2016. A Probabilistic Programming Approach to Probabilistic Data Analysis. In *Advances in Neural Information Processing Systems*, Vol. 29. Curran Associates, Inc., Red Hook, NY, USA, 2011–2019.
[69] 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, Article 37 (2019), 29 pages. https://doi.org/10.1145/3290350
[70] Feras A. Saad and Vikash K. Mansinghka. 2018. Temporally-Reweighted Chinese Restaurant Process Mixtures for Clustering, Imputing, and Forecasting Multivariate Time Series. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 755–764.
[71] Feras A. Saad and Vikash K. Mansinghka. 2021. Hierarchical Infinite Relational Model. In *Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence*. PMLR, Norfolk, MA, USA, 1067–1077.

[72] Feras A. Saad, Brian J. Patton, Matthew D. Hoffmann, Rif A. Saurous, and Vikash K. Mansinghka. 2023. Sequential Monte Carlo Learning for Time Series Structure Discovery. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, Norfolk, MA, USA, Article 1226, 17 pages.

[73] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 804–819. https://doi.org/10.1145/3453483.3454078

[74] Feras A. K. Saad. 2022. *Scalable Structure Learning, Inference, and Analysis with Probabilistic Programs*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[75] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic Programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55. https://doi.org/10.7717/peerj-cs.55

[76] Ulrich Schaechtle, Cameron Freer, Zane Shelby, Feras Saad, and Vikash Mansinghka. 2022. Bayesian AutoML for Databases via the InferenceQL Probabilistic Programming System. In *Proceedings of the 1st Conference on Automated Machine Learning (Late-Breaking Workshop)*. AutoML-Conf, 8 pages.

[77] Ulrich Schaechtle and Mathieu Huot. 2024. PLDI artifact evaluation for "PSQL: A Probabilistic Programming System for Automating Inference over Databases and Probabilistic Models. Zenodo. https://doi.org/10.5281/zenodo.10949799

[78] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 130–144. https://doi.org/10.1145/3009837.3009852

[79] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Springer, Cham. https://doi.org/10.1007/978-3-031-01879-4

[80] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization Of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, New York, NY, USA, 847–855. https://doi.org/10.1145/2487575.2487629

[81] Guy Van den Broeck and Dan Suciu. 2017. Query Processing on Probabilistic Data: A Survey. *Foundations and Trends in Databases* 7, 3-4 (2017), 197–341. https://doi.org/10.1561/1900000052

[82] Baojun Wang, Mauricio Barahona, and Martin Buck. 2013. A Modular Cell-Based Biosensor Using Engineered Genetic Logic Circuits to Detect and Integrate Multiple Environmental Signals. *Biosensors and Bioelectronics* 40, 1 (2013), 368–376. https://doi.org/10.1016/j.bios.2012.08.011

[83] WHO. 2023. World Health Organization: Obesity. https://www.who.int/health-topics/obesity

[84] Darren J. Wilkinson. 2018. *Stochastic Modelling for Systems Biology*. CRC Press, Boca Raton, FL, USA.

[85] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A New Approach To Probabilistic Programming Inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 1024–1032.

[86] Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart Russell. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. In *Proceedings of the 35th International Conference on Machine Learning*. PMLR, Norfolk, MA, USA, 5343–5352.

[87] xCures. 2019. BEAT19 Behavior, Environment And Treatments for COVID-19. https://classic.clinicaltrials.gov/ct2/show/results/NCT04321811?view=results

[88] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling Tabular Data using Conditional GAN. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc., Red Hook, NY, USA, 11 pages.

[89] Yubin Xue, Pei Du, Amal Amin Ibrahim Shendi, and Bo Yu. 2022. Mercury Bioremediation in Aquatic Environment by Genetically Modified Bacteria with Self-Controlled Biosecurity Circuit. *Journal of Cleaner Production* 337 (2022), 130524. https://doi.org/10.1016/j.jclepro.2022.130524

[90] Fabian Zaiser, Andrzej Murawski, and Chih-Hao Luke Ong. 2023. Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., Red Hook, NY, USA, 2427–2462.