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.
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 \textit{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 \textit{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 \textit{probability of} \( e \) \ textit{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, 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](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 values of 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 $r$:

```sql
SELECT weight, AVG(log_pxy_div_px_py) AS mutual_information
FROM (SELECT weight, LOG(pxy) - (LOG(px) + LOG(py)) AS log_pxy_div_px_py
      FROM (SELECT weight,
              PROBABILITY OF h_model.age = table.age AND h_model.bmi = table.bmi
              UNDER h_model GIVEN h_model.weight = table.weight AS px,
              PROBABILITY OF h_model.age = table.age
              UNDER h_model GIVEN h_model.weight = table.weight AS px,
              PROBABILITY OF h_model.bmi = table.bmi
              UNDER h_model GIVEN h_model.weight = table.weight AS py
      FROM (SELECT table.weight, table.age, table.bmi
            FROM health_data DUPLICATE 1000 TIMES
            GENERATIVE JOIN h_model
            GIVEN h_model.weight = health_data.weight) AS table)))
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

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

where $k$ is the number of patients with that specific weight, and $(\text{age}_i, \text{bmi}_i)$ is a sample from the model for that weight. To do so, lines 6–11 compute the probability densities $p(\text{age}_i, \text{bmi}_i)$ (lines 6–7), $p(\text{age}_i)$ (lines 8–9), and $p(\text{bmi}_i)$ (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 $\log p(\text{age}_i, \text{bmi}_i) - \log p(\text{age}_i)p(\text{bmi}_i)$ for each of the rows, naming this ratio $\log_{\text{pxy}} \div \text{px}_{\text{py}}$. Note that there are $1000k \log_{\text{pxy}} \div \text{px}_{\text{py}}$ 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 $\log_{\text{pxy}} \div \text{px}_{\text{py}}$ 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. 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, 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 $C$ of names col $\in C$ for the columns of tables and rowModels, as well as a countable set $I$ of identifiers $ID \in I$ for naming tables and rowModels.

---

1The 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 | $\sigma ::= \sigma_c \mid \sigma_d$ | $E ::= C^r_{\{col_1 : \sigma_1, \ldots, col_n : \sigma_n\}}$
Table/RowModel Type | $T ::= T[\text{ID}]\{col_1 : \sigma_1, \ldots, col_n : \sigma_n\}$ | $M ::= M[\text{ID}]\{col_1 : \sigma_1, \ldots, col_n : \sigma_n\}$
Table Expression | $t ::= \text{ID} \mid \text{RENAME t AS ID}$ | $t ::= \ldots$
 | $t_1 \text{ JOIN } t_2 \mid t \text{ WHERE } e$ | $t ::= \ldots$
 | $\text{SELECT } e \text{ AS COL from } t$ | $\text{GENERATE UNDER m LIMIT e}$
 | $t \text{ GENERATIVE JOIN } m$ | $t \text{ GENERATIVE JOIN } m$
RowModel Expression | $m ::= \text{ID} \mid m \text{ GIVEN } c^f \mid \text{RENAME m AS ID}$
Scalar Expression | $e ::= \text{ID}.\text{COL} \mid \text{OP}(e_1, \ldots, e_n)$ | $e ::= \ldots \mid \text{PROBABILITY of } c^f \text{ UNDER } m$
Event Expressions | $c^1 ::= c^1_1 \land c^1_2 \mid c^1_1 \lor c^1_2 \mid \text{ID.\text{COL}} \text{ OP } e$ | $c^0 ::= c^0_1 \land c^0_2 \mid \text{ID.\text{COL}} = e$
Event-0 Expressions | | |

**Primitive Domains:** $\text{OP} \in \{\text{OP}\}$, $\text{ID} \in I$, $\text{COL} \in C$, $\sigma_c \in \{\text{Real, PosReal, Ranged}(a, b), \ldots\}$, $\sigma_d \in \{\text{Int, Str, Nat, Bool}\}$

**Syntactic Sugar:** $e \text{ AS COL} \equiv e_1 \text{ AS col}_1, \ldots, e_n \text{ AS col}_n$.

*Fig. 3. Syntax of GenSQL.*

**Base types.** Cells of tables can have a base type $\sigma$, which is either a continuous type $\sigma_c$ or a discrete type $\sigma_d$. 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($N_1, \ldots, N_k$)** 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. $?\text{ID}$ is an optional identifier, allowing access to columns of a table or rowModel in a query. For instance, in $\text{SELECT ID.weight}$, the identifier $\text{ID}$ refers to a table and weight to a column of that table. The identifier can be optional, e.g. there is no default identifier for a table created after a join. The notation $\{\text{col}_1 : \sigma_1, \ldots, \text{col}_n : \sigma_n\}$ indicates that the table has columns $\text{col}_i$ of type $\sigma_i$. 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.** $\text{OP}$ is a set of primitive operations on base types including standard operations such as $+, *, <, >, =$ on integers and reals, $\land, \lor$ on Booleans, as well as constants for every value of a base type. For any $\text{OP} \in \text{OP}$, we write $\text{OP} : \sigma_1, \ldots, \sigma_n \to \sigma$ if $\text{OP}$ has arity $n$, takes arguments of base types $\sigma_1, \ldots, \sigma_n$ and returns a value of type $\sigma$. 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. $\text{Null} + 3 \mapsto \text{Null}$, $\text{Null} * 4.1 \mapsto \text{Null}$). 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. $x > 6 \land x < 3$, and those do not require a separate treatment. Events and events-0 are used in the **PROBABILITY of** and **GIVEN** constructs.
**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 \( \Delta \) 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. \( \Delta \) is a set of types, while \( \Delta \) is an ordered list of types. In the premise of a

---

²It 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 \texttt{PROBABILITY OF}, only the last element of \(\Delta\) will be accessible to an expression. We denote the empty context by \([\cdot]\). Intuitively, \(\Gamma\) contains the ambient tables in the database schema and any loaded models, and within \(\Gamma\), all identifiers \(\text{id}\) are assumed distinct. \(\Delta\) is the \textit{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 \texttt{PROBABILITY OF} expression has a \texttt{rowModel} in scope, it will query the model for the probability of an event under that model. If it’s a \texttt{GIVEN} expression, it will condition that model on an event.

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

\textbf{Notations used in the type system.} \(?\text{id}\) indicates an optional identifier and \text{id}. "\text{id}’ fresh" means that \text{id}’ is not in the contexts \(\Gamma, \Delta\) 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 \text{cols} and in \text{cols}’ should be disjoint. In the first typing rule for events, we write \(\forall \sigma_c. (\sigma_i, \text{op}) \neq (\sigma_c, =)\) to mean that \text{op} cannot be an equality on a continuous type. We recursively define the following two macros:

\[
\text{vars}(\text{id}.\text{col} \; \text{op} \; t) = \{\text{col}\} \quad \text{vars}(c \; \land \; c’) = \text{vars}(c) \cup \text{vars}(c’) \quad \text{vars}(c \; \lor \; c’) = \text{vars}(c) \cup \text{vars}(c’)
\]

\[
\text{condvars}(\text{m \; \text{given} \; c^0) = \text{vars}(c^0) \quad \text{condvars}(\text{m \; \text{given} \; c^1) = \text{condvars}(m) \quad \text{condvars}(\text{id}) = \emptyset
\]

\textbf{Restrictions imposed by the type system.} If the same identifier \text{id} appears twice in the premise of a typing rule, the two identifiers must equal, and two different identifiers \text{id} and \text{id}’ must be distinct. The \texttt{join} and \texttt{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 \texttt{ID GIVEN c^0}. Events-0 follow a linear typing system to avoid contradictory statements such as \texttt{ID.COL = 1.0 \land ID.COL = 2.0}. Events-0 in a \texttt{PROBABILITY OF} query on a conditioned model cannot refer to the conditioned columns of the model, which is enforced by the restriction \texttt{vars(c^0) \cap condvars(m) = \emptyset}.

\textbf{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:

\[
\begin{align*}
\bullet & \quad \texttt{SELECT } * \texttt{ FROM } t \rightsquigarrow \texttt{SELECT ID.COL}_1, \ldots, \text{ID.COL}_n \texttt{ FROM } t \\
\bullet & \quad \texttt{PROBABILITY OF } * \rightsquigarrow \texttt{PROBABILITY OF } e \text{ for any query of the form } \texttt{SELECT PROBABILITY OF } * \texttt{ UNDER m \; \text{given} \; c \texttt{ FROM } t, \text{ where } e : \text{ID’.COL}_1’ = \text{ID}.\text{COL}_1 \land \ldots \land \text{ID’.COL}_n’ = \text{ID}.\text{COL}_n. \\
\bullet & \quad \text{m \; \text{given} } * \rightsquigarrow \text{m \; \text{given} } e \text{ within a } \texttt{SELECT \; FROM \; t} \text{ query. The event } e : \text{ID’.COL}_{i_1}’ = \text{ID}.\text{COL}_{i_1} \land \ldots \land \text{ID’.COL}_{i_k}’ = \text{ID}.\text{COL}_{i_k}, \text{ where the } \text{COL}_{i_j} \text{ are columns } t \text{ that do not appear in the } \texttt{SELECT} \text{ clause.} \\
\bullet & \quad \texttt{EXCEPT ID.COL} \text{ removes the column } \text{COL} \text{ from list of columns that } * \text{ selects.}
\end{align*}
\]

\footnote{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 $\llbracket e \rrbracket$ is a shorthand for $\llbracket \Gamma; \Delta \vdash e : t \rrbracket$.

**Base types (Fig. 5c).** We assign to each type $\sigma$ a measure space $\llbracket \sigma \rrbracket := (X_\sigma, \Sigma_\sigma, \nu_\sigma)$ consisting of a set $X_\sigma$, a sigma-algebra $\Sigma_\sigma$, and reference measure $\nu_\sigma$. $\mathbb{Z}$ 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 $\text{Null}$ 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 $\sigma$ 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 $\sigma_d$ such as $\text{Int}, \text{Nat}, \text{Bool}, \text{Str}$ is the counting measure. On continuous types $\sigma_c$ such as $\text{Real}$, the base measure is the Lebesgue measure $\lambda$. These are extended to base measures $\nu_\sigma$ on $\llbracket \sigma \rrbracket$ by using the dirac measure $\delta_{\{\star\}}$ on $\{\star\}$, e.g. the base measure on $\llbracket \mathbb{R} \rrbracket$ 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 $\prod_{1 \leq i \leq n} \llbracket \sigma_i \rrbracket$ by taking the product of the reference measures $\nu := \bigotimes_{1 \leq i \leq n} \nu_{\sigma_i}$.

**Table types (Fig. 5a).** Our semantics has two modes of interpreting table types, a “tupple mode” $\text{Tup}[\llbracket \cdot \rrbracket]$, and a “table mode” $\text{Tab}[\llbracket \cdot \rrbracket]$. $\text{Tab}[\llbracket \cdot \rrbracket]$ interprets tables as measures on bags of tuples, while $\text{Tup}[\llbracket \cdot \rrbracket]$ 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 $\text{Bag}[\llbracket T[?\text{ID} \{\text{COLS}\}] \rrbracket] = \text{Bag}(\text{Tup}[\llbracket T[?\text{ID} \{\text{COLS}\}] \rrbracket])$, where $\text{Bag}(X) = \{f : X \to \mathbb{N} | f(x) = 0 \text{ except for finitely many } x\}$. $\text{Bag}(X)$ is equipped with the least sigma-algebra containing the generating sets $\{b \in \text{Bag}(X) | b \text{ contains exactly } k \text{ elements in } A\}$ for measurable sets $A$ of $X$ [21].

**Contexts (Fig. 5b).** We interpret the global context $\Gamma$ with the table semantics $\text{Tab}[\llbracket \cdot \rrbracket]$ and the local context $\Delta$ with the tuple semantics $\text{Tup}[\llbracket \cdot \rrbracket]$. We write $\gamma$ for an element of $\text{Tab}[\llbracket \cdot \rrbracket]$, and see it as a finite map from identifiers to values. Likewise, we write $\delta$ for an element of $\Delta$. We write $\delta[\text{ID} \mapsto v]$ for the extended finite map mapping ID to $v$.

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

**Event expressions (Fig. 5g).** An event expression $\llbracket e : C^1 \{\text{COLS}\} \rrbracket(\gamma, \delta)$ is interpreted as a measurable subset $S$ of $\llbracket \text{COLS} \rrbracket$ (disjoint union of hyper-rectangles [73]). Depending on the expression, this set $S$ is used in different ways. We interpret the probability clause $\text{PROBABILITY OF } e^1 \text{ UNDER } m$ as $\int_{\llbracket \text{COLS} \rrbracket} \mathbb{I}_S d\mu$, where $\mu$ is the measure denoting the model $m$, i.e. $S$ is used in an indicator function $\mathbb{I}_S$. When used in a $\text{GIVEN}$ 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 $\text{Null}$. A similar situation to $\text{WHERE Null}$ arises for $\text{GIVEN}$, e.g. in $\text{GIVEN ID.COL \ op Null}$. Following the principle of least surprise, $\text{Null}$ acts by convention as a unit for conditioning, i.e. $\text{ID GIVEN ID.COL \ op Null}$ behaves the same as $\text{ID}$. To ensure this we interpret boolean expressions $\text{op}$ differently in the semantics of events, and write $\text{op}_t$ for the extended version of $\text{op}$ which sends $\star$ to $\text{true}$. The denotation of $\text{ID.COL \ op Null}$ 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

\[
\begin{align*}
\text{Tuple} [T?\{\text{cols}\}] \{\text{col}_1: \sigma_1, \ldots, \text{col}_n : \sigma_n\} &= \|\Pi_{1\leq i \leq n}^{\sigma_i}\| \\
\text{Matrix} [M?\{\text{cols}\}] \{\text{col}_1: \sigma_1, \ldots, \text{col}_n : \sigma_n\} &= \mathcal{P}_{\text{dom}}(\Pi_{1\leq i \leq n}^{\sigma_i}) \\
\text{Table} [T?\{\text{cols}\}] \{\text{col}_1: \sigma_1, \ldots, \text{col}_n : \sigma_n\} &= \mathcal{P}_{\text{bag}}(\Pi_{1\leq i \leq n}^{\sigma_i}) \\
\text{Matrix} [M?\{\text{cols}\}] \{\text{col}_1: \sigma_1, \ldots, \text{col}_n : \sigma_n\} &= \mathcal{P}_{\text{adm}}(\Pi_{1\leq i \leq n}^{\sigma_i})
\end{align*}
\]
We use functional programming notation for the mathematical functions `filter`, `map`, `map2`, `fold`. Given a bag `S` and a function `f : S → ℝ`, we define the bag `filter f S := {x ∈ S | f(x) ≠ 0}`. Likewise, we define `map f S := {f(x) | x ∈ S}` and `map2 f S T := {f(x, y) | x ∈ S, y ∈ T}`. A function `f : X × Y → Y` is commutative if `f(x1, f(x2, y)) = f(x2, f(x1, y))` for all `x1, x2, y`. Given a commutative function `f : X × Y → Y`, we further define `fold f : Y × Bag(X) → Y` by `fold f y0{x1, . . . , xn} = f(x1, f(x2, . . . f(xn, y0) . . . ))`.

RowModel expressions (Fig. 5f). The semantics of rowModels is more involved, as conditioning statements `given e` 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 `νA, νB`, a measure `μ` on `A × B` admits an `(A, B)` *disintegration* if we can write `μ = νA ⊗ κ` for some measure kernel `κ` such that for all `a ∈ A`, `κ(a)` has a density `p(− | a)` w.r.t. `νB`. A valid decomposition `(A, B)` for `Π_{1≤i≤n}σ_i` is given by `A = Π_{j∈J}[σ_j]` for some `J ⊆ {1, . . . , n}` and `B = Π_{j∈[1...n] \ J}[σ_j]`. A measure `μ ∈ P(Π_{1≤i≤n}σ_i)` is admissible if it admits an `(A, B)` disintegration for all valid decompositions `(A, B)` of `Π_{1≤i≤n}σ_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 `Tab[1D] := (μ, p, {κA}A, {ρ}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 `κA` is an `(A, B)`-disintegration of `μ`. For all `a ∈ A`, `p_A(− | a)` is a density for `κ_A(a)` w.r.t. the reference measure `νB`. If `v` is a partial assignment of the variables in `B`, we also write `p_A(v | a)` for the marginal density of `κ_A(a)` at `v` obtained from `p_A(− | a)` by integrating out the missing variables in `v`. We denote by `P_{adm}(X)` the set of such quadruples `(μ, p, {κA}A, {ρ}A)`, where `μ` is a measure on `X`. Given `m ∈ P_{adm}(X)`, we write `m.meas` for its first component `μ`, `m.pdf` for the density `p`, `m.A` for the kernel `κA`, and `m.A.pdf` for the density `p_A`. Using this notation, given an event-0 `e` denoting a projection `π` and value `v`, the expression `m.pdf(v)` gives a marginal density of the model `m` at `v`; i.e. `m.pdf(v)` is a version of the density of `π.m.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 densitites, which is enforced in the semantics of rowModel expressions. We write `P_{dens}(X)` for the set of pairs `(μ, p)` where `μ` is a measure on `X := X_1 × . . . × X_n` and `p` is either a density of `μ` w.r.t. the reference measure, or of the form `λ(x_1, . . . , x_n).q(x_i1, . . . , x_in)` for some `i_1, . . . , i_k`, and where `q` is a marginal density of `μ` on `X_{i_1} × . . . × 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 ∈ P_{adm}(X)`, `π : X → Y` a projection function, and `v ∈ Y`, we define `Dis(m, π, v) := (m.(π(X))(v) \ δ_v, m.(π(X)).pdf(v))`.

For conditioning on events, given `m ∈ P_{dens}(X)` and a measurable `S ⊆ X`, we define

```
cond(m, S) :=
\begin{cases}
    \left( \lambda S \cdot \frac{m.meas(S \cap S')}{m.meas(S)}, \lambda x. \frac{I_S(x)m.pdf(x)}{m.meas(S)} \right) & \text{if } m.meas(S) > 0 \\
    (\delta_{\{\ast, \ldots, \ast\}}, I_{\{\{\ast, \ldots, \ast\}}) & \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 $M[id] \{cols\}$, the AMI requires the existence of the following three methods:

$$
\begin{align*}
\text{simulate}_{id} &: (C^0 \{cols'\}, C^1 \{cols\}) \rightarrow T[id] \{cols\} \\
\text{logpdf}_{id} &: (C^0 \{cols'\}, C^1 \{cols\}, C^0' \{cols''\}) \rightarrow \text{Real} \\
\text{prob}_{id} &: (C^0 \{cols'\}, C^1 \{cols\}, C^1' \{cols\}) \rightarrow \text{Ranged}(0, 1).
\end{align*}
$$

where $cols' \subseteq cols$. These methods should behave as follows:

- $\text{simulate}_{id}(c^0, c^1)$ returns a sample from a model with identifier ID, conditioned on the event-0 $c^0$ and event $c^1$.
- $\text{logpdf}_{id}(c^0, c^1, c^0')$ returns the (marginal if $cols'' \subseteq cols$) log-density of the model ID conditioned on the event-0 $c^0$ and event $c^1$, at the point $c^0$.
- $\text{prob}_{id}(c^0, c^1, c^1')$ returns the probability of the event $c^1$ 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 $c^1$ 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 $c^1$).
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 acting on bags, and primitives for the AMI. It also contains a version of events and transforms a query. The rationale is that a table identifier that the program transform preserves the semantics of the lowered query (Theorem 4.2).

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 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
Given simulate

GIVEN

18

64

JOIN

GENERATIVE JOIN

id

WHERE

RENAME

64

GIVEN

UNDER

GIVEN

D.3

GIVEN

Huot, Ghavami, Lew, Schaechtle, Freer, Shelby, Rinard, Saad, Mansinghka

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

Fig. 7. The lowering transformation \( \mathcal{T}\{\cdot\}\).
in the exact semantics. As an example, the approximate semantics of $\text{\textbf{map}} (x.t_1) t_2$\text{\textbf{approx}} in the context $\gamma$ and at the “random seed” $\omega \in \Omega$ is given at the $n$-th approximation by

$$t_2 \text{\textbf{approx}}(y, \omega)_n \Rightarrow (\lambda S. \text{\textbf{return}} \{ \lambda x'. [t_1]_{\text{\textbf{approx}}}(y \mapsto x', \omega)_n \mid y \in S \}).$$

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 $\gamma$, 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 $\{\mu^n_\text{id}\}_{0}^{\infty}$ such that

$$\text{\textbf{Bag}} \prod_{i=1}^{k} \{\sigma_i\};$$

- for $\text{id}$, $\gamma$, $c^1$ and $c^0$ as above, and $c^1_2 : C^1[\{\sigma_1, \ldots, \sigma_k\}]$, there exists a sequence of real random variables $\{\nu^n_\text{id}\}_{0}^{\infty}$ such that

$$(\nu^n_\text{id})_{\text{\textbf{approx}}}(\gamma)_n \approx (\nu^n_\text{id})_{\text{\textbf{approx}}}(\gamma)_n$$

which takes values in $[0, 1]$ \text{\textbf{P}}-almost surely;

- for $\text{id}$, $\gamma$, $c^1$ and $c^0$ as above, and $c^0_2 : C^0[\{\sigma_1, \ldots, \sigma_k\}]$, there exists a sequence of real random variables $\{\lambda^n_\text{id}\}_{0}^{\infty}$ such that

$$(\lambda^n_\text{id})_{\text{\textbf{approx}}}(\gamma)_n \approx (\lambda^n_\text{id})_{\text{\textbf{approx}}}(\gamma)_n.$$

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 $(\gamma, \delta)$:

$$\text{\textbf{SELECT}} \text{id}.\text{col} \text{\textbf{FROM}} \text{id} \text{\textbf{WHERE}} \text{id}.\text{col} \leq (\text{\textbf{PROBABILITY}} \text{\textbf{OF}} \text{id}'.\text{col}' = 7).$$

If the value of the term $\text{\textbf{PROBABILITY}} \text{\textbf{OF}} \text{id}'.\text{col}' = 7$ is approximated, even if we can make this approximation arbitrarily accurate, the output of the query need not converge. For example, if the table $\text{id}$ contains a row in which the value of $\text{col}$ is exactly $7$ 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 $\gamma$ in which each random variable is replaced by its limit as $n \rightarrow \infty$. 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 $\Gamma, [[t]] : T[?\text{id}]\{\text{cols}\}$ be a safe query and suppose the AMI methods have asymptotically sound implementations. Then, for every evaluation of the context $\gamma$, \text{\textbf{P}}-almost surely

$$\lim_n (\{[t]_n\}_{\text{\textbf{approx}}}(\gamma) = [t]([\lim \gamma_n]).$$

## 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.

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

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 \texttt{PROBABILITY OF ID\_x > 42 UNDER ID GIVEN ID\_y = 17} to the semantically equivalent query \texttt{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
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 $P(BMI \mid \star) < 0.01$ for all BMI values within the 5th (20.3) and 95th (38.4) percentiles in the US.

(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) Conditional probabilities 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 mislabeling 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.
GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables

---

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

(a) Overall population

```
SELECT * FROM gene_expression_data
GENERATE UNDER gene_expression_model
LIMIT 1000
```

(b) 6 genes from the overall population.

(c) Add IPTG but not Arabinose

```
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
```

(d) 6 genes from (c).

(e) Add both IPTG and Arabinose

```
SELECT * FROM gene_expression_data
WHERE Ara. = "added" AND IPTG = "added"
GENERATE UNDER gene_expression_model
GIVEN Ara. = "added" AND IPTG = "added"
LIMIT 1000
```

(f) 6 genes from (e).

---

**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.


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 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. 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.** 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


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