PIP: A Database System for Great and Small Expectations

Oliver Kennedy and Christoph Koch
Department of Computer Science
Cornell University, Ithaca, NY, USA
{okennedy, koch}@cs.cornell.edu

***Abstract*— Estimation via sampling out of highly selective join queries is well known to be problematic, most notably in online aggregation. Without goal-directed sampling strategies, samples falling outside of the selection constraints lower estimation efficiency at best, and cause inaccurate estimates at worst. This problem appears in general probabilistic database systems, where query processing is tightly coupled with sampling. By committing to a set of samples before evaluating the query, the engine wastes effort on samples that will be discarded, query processing that may need to be repeated, or unnecessarily large numbers of samples.**

We describe PIP, a general probabilistic database system that uses symbolic representations of probabilistic data to defer computation of expectations, moments, and other statistical measures until the expression to be measured is fully known. This approach is sufficiently general to admit both continuous and discrete distributions. Moreover, deferring sampling enables a broad range of goal-oriented sampling-based (as well as exact) integration techniques for computing expectations, allows the selection of the integration strategy most appropriate to the expression being measured, and can reduce the amount of sampling work required.

We demonstrate the effectiveness of this approach by showing that even straightforward algorithms can make use of the added information. These algorithms have a profoundly positive impact on the efficiency and accuracy of expectation computations, particularly in the case of highly selective join queries.

I. Introduction

Uncertain data comes in many forms: Statistical models, scientific applications, and data extraction from unstructured text are all forms of uncertain data. Measurements have error margins while model predictions are often drawn from well known distributions. Traditional database management systems (DBMS) are ill-equipped to manage this kind of uncertainty. For example, consider a risk-management application that uses statistical models to evaluate the long term effects of corporate decisions and policies. This application may use a DBMS to store predictions and statistical measures (e.g., error bounds) of those predictions. However, arbitrary queries made on the predictions do not translate naturally into queries on the corresponding statistical measures. A user who requires error bounds on the sum of a join over several tables of predictions must first obtain a formula for computing those bounds, assuming a closed form formula even exists.

Probabilistic database management systems [3], [4], [21], [5], [11], [17], [18], [10], [20] aim at providing better support for querying uncertain data. Queries in these systems preserve the statistical properties of the data being queried, allowing users to obtain metrics about and representations of query results. The previously mentioned risk-management application, built on top of a probabilistic database, could use the database itself to obtain error bounds on the results of arbitrary queries over its predictions. By encoding the statistical model for its predictions in the database itself, the risk-management application could even use the probabilistic database to estimate complex functions over many correlated variables in its model. In effect, the application could compute all of its predictions within the probabilistic database in the first place.

Few systems are general enough to efficiently query probabilistic data defined over both discrete and continuous distributions. Those that are, generally rely on sampling to estimate desired values, as exact solutions can be hard to obtain. If a query contains a selection predicate, samples violating the predicate are dropped and do not contribute to the expectation. The more selective the predicate, the more samples are needed to maintain consistent accuracy. For example, a query may combine a model predicting customer profits with a model for predicting dissatisfied customers, perhaps as a result of a corporate decision to use a cheaper, but slower shipping company. If the query asks for profit loss due to dissatisfied customers, the query need only consider profit from customers under those conditions where the customer is dissatisfied (ie, the underlying model may include a correlation between ordering patterns and dependence on fast shipping).

Without knowing the likelihood that customer A is satisfied, the query engine must over-provision and waste time generating large numbers of samples, or risk needing to re-evaluate the query if additional samples are needed. This problem is well known in online aggregation, but ignored in general-purpose (i.e., both discrete and continuous) probabilistic databases.

Example 1.1: Suppose a database captures customer orders expected for the next quarter, including prices and destinations of shipment. The order prices are uncertain, but a probability distribution is assumed. The database also stores distributions of shipping durations for each location. Here are two c-tables defining such a probabilistic database:

Order	*Cust*	*ShipTo*	*Price*
	Joe	*NY*	X_1
	Bob	*LA*	X_3

Shipping	*Dest*	*Duration*
	NY	X_2
	LA	X_4

We assume a suitable specification of the joint distribution p of the random variables $X_1, \ldots, X_4$ occurring in this database.

Now consider the query

```
select expected_sum(O.Price)
from   Order O, Shipping S
where  O.ShipTo = S.Dest
and    O.Cust = 'Joe'
and    S.Duration >= 7;
```

asking for the expected loss due to late deliveries to customers named Joe, where the product is free if not delivered within seven days. This can be approximated by monte carlo sampling from p, where q represents the result of the sum aggregate query on a sample, here

$$q(\vec{x}) = \begin{cases} x_1 & \ldots \quad x_2 \geq 7 \\ 0 & \ldots \quad \textit{otherwise.} \end{cases}$$

In a naive sample-first approach, if $x_2 \geq 7$ is a relatively rare event, a large number of samples will be required to compute a good approximation to the expectation. Moreover, the profit x_1 is independent from the shipping time x_2. Despite this, samples for x_1 are still discarded if the constraint on x_2 is not met.

A. Contributions

Selective queries exemplify the need for contextual information when computing expectations and moments. This paper presents PIP, a highly extensible, general probabilistic database system built around this need for information. PIP evaluates queries on symbolic representations of probabilistic data, computing a complete representation of the probabilistic expression to be evaluated before an expectation or moment is taken. To our knowledge, PIP is the first probabilistic database system supporting continuous distributions to evaluate queries in this way.

PIP's approach encompasses and extends the strengths of discrete systems that use c-tables such as Trio [21], MystiQ [4], and MayBMS [1], as well as the generality of the sample-first approach taken by MCDB [10]. It supports both discrete and continuous probability distributions, statistical dependencies definable by queries, expectations of aggregates and distinct-aggregates with or without group-by, and the computation of confidences. The detailed technical contributions of this paper are as follows.

- We propose PIP, the first probabilistic database system based on c-tables to efficiently support continuous probability distributions.
- We show how PIP acts as a generalizable framework for exploiting information about distributions beyond simple sampling functionality (e.g., inverse cdfs) to enhance query processing speed and accuracy. We demonstrate this framework by implementing several traditional statistical optimizations within it.
- We propose a technique for identifying variable independence in c-table conditions and exploit it to accelerate sampling.
- We provide experimental evidence for the competitiveness of our approach. We compare PIP with a reimplementation of the refined sample-first approach taken by MCDB by using a common codebase (both systems are implemented on top of Postgres) to enable fair comparison. We show that PIP's framework performs considerably better than MCDB over a wide range of queries, despite applying only a relatively straightforward set of statistical optimizations. Even in the worst case, PIP remains competitive with MCDB (it essentially never does more work).

II. Background and Related Work

The estimation of probabilities of continuous distributions frequently devolves into the computation of complex integrals. PIP's architecture allows it to identify cases where efficient algorithms exist to obtain a solution. For more complex problems not covered by these cases, PIP relies on Monte Carlo integration [14], a conceptually simple technique that allows for the (approximate) numerical integration of even the most general functions. Conceptually, to compute the expectation of function $q(\vec{x})$, one simply approximates the integral by taking n samples $\vec{x}_1, \ldots, \vec{x}_n$ for $\vec{X}$ from their distribution $p(\vec{X})$ and taking the average of the function evaluated on all n values.

In general, even taking a sample from a complicated PDF is difficult. Constraints imposed by queries break traditional Monte Carlo assumptions of normalization on $p(\vec{X})$ and require that the sampling technique account for them or lose precision. A variety of techniques exist to address this problem, from straightforward rejection sampling, where constraint-violating samples are repeatedly discarded, to more heavy duty Markov-chain Monte Carlo (MCMC, cf. e.g., [6]) style techniques such as the Metropolis-Hastings algorithm [13], [6].

Recently, a paper on the MCDB system[10] has promoted an integrated sampling-based approach to probabilistic databases. Conceptually, MCDB uses a *sample-first* approach: it first computes samples of entire databases and then processes queries on these samples. This is a very general and flexible approach, largely due to its modular approach to probability distributions via black box sample generators called VG Functions. Using Tuple-Bundles, a highly compressed representation of the sampled database instances, MCDB evaluates queries on these instances in parallel, sharing computation across instances where possible.

Conditional tables (c-tables, [9]) are relational tables in which tuples have associated conditions expressed as boolean expressions over comparisons of random variables and constants. C-tables are a natural way to represent the *deterministic skeleton* of a probabilistic relational database in a succinct and tabular form. That is, complete information about uncertain data is encoded using random variables, excluding only specifications of the joint probability distribution of the random variables themselves. This model allows representation of input databases with nontrivial statistical dependencies that are normally associated with graphical models.

For *discrete* probabilistic databases, a canon of systems has been developed that essentially use c-tables, without referring to them as such. MystiQ [4] uses c-tables internally for query

processing but uses a simpler model for input databases. Trio [21] uses c-tables with additional syntactic sugar and calls conditions *lineage*. MayBMS [1] uses a form of c-tables called U-relations that define how relational algebra representations of queries can encode the corresponding condition transformations.

ORION [18] is a probabilistic database management system for continuous distributions that can alternate between sampling and transforming distributions. However, their representation system is not based on c-tables but essentially on the world-set decompositions of [2], a factorization based approach related to graphical models. Selection queries in this model may require an exponential blow-up in the representation size, while selections are efficient in c-tables.

A. C-tables

A c-table over a set of variables is a relational table[1] extended by a column for holding a *local condition* for each tuple. A local condition is a Boolean combination (using "and", "or", and "not") of atomic conditions, which are constructed from variables and constants using $=$, $<$, $\leq$, $\neq$, $>$, and $\geq$. The fields of the remaining data columns may hold domain values or variables.

Given a variable assignment θ that maps each variable to a domain value and a condition ϕ, $\theta(\phi)$ denotes the condition obtained from ϕ by replacing each variable X occurring in it by $\theta(X)$. Analogously, $\theta(\vec{t})$ denotes the tuple obtained from tuple $\vec{t}$ by replacing all variables using θ.

The semantics of c-tables are defined in terms of possible worlds as follows. A possible world is identified with a variable assignment θ. A relation R in that possible world is obtained from its c-table C_R as

$$R := \{\!|\, \theta(\vec{t}) \mid (\vec{t}, \phi) \in C_R, \theta(\phi) \text{ is true} \,|\!\}.$$

That is, for each tuple $(\vec{t}, \phi)$ of the c-table, where ϕ is the local condition and $\vec{t}$ is the remainder of the tuple, $\theta(\vec{t})$ exists in the world if and only if $\theta(\phi)$ is true. Note that each c-table has at least one possible world, but worlds constructed from distinct variable assignments do not necessarily represent different database instances.

B. Relational algebra on c-tables

Evaluating relational algebra on c-tables (and without the slightest difference, on probabilistic c-tables, since probabilities need not be touched at all) is surprisingly straightforward. The evaluation of the operators of relational algebra on multiset c-tables is summarized in Figure 1. An explicit operator "distinct" is used to perform duplicate elimination.

Example 2.1: We continue the example from the introduction. The input c-tables are

$$C_{\text{Order}} = \{\!|\, ((Joe, NY, X_1), \textit{true}), ((Bob, LA, X_3), \textit{true}) \,|\!\}$$

[1]In the following, we use a multiset semantics for tables: Tables may contain duplicate tuples. Sets transformations are defined in comprehension notation $\{\!|\, \cdot \mid \cdot \,|\!\}$ with $\in$ as an iterator. Transformations preserve duplicates. We use $\uplus$ to denote bag union, which can be thought of as list concatenation if the multisets are represented as unsorted lists.

$$\begin{aligned}
C_{\sigma_\psi(R)} &= \{\!|\, (\vec{r}, \phi \wedge \psi[\vec{r}]) \mid (\vec{r}, \phi) \in C_R \,|\!\} \\
&\ldots\ \psi[\vec{r}] \text{ denotes } \psi \text{ with each reference to} \\
&\quad\ \text{a column } A \text{ of } R \text{ replaced by } \vec{r}.A. \\
C_{\pi_{\vec{A}}(R)} &= \{\!|\, (\vec{r}.\vec{A}, \phi) \mid (\vec{r}, \phi) \in C_R \,|\!\} \\
C_{R \times S} &= \{\!|\, (\vec{r}, \vec{s}, \phi \wedge \psi) \mid (\vec{r}, \phi) \in C_R, (\vec{s}, \psi) \in C_S \,|\!\} \\
C_{R \cup S} &= C_R \uplus C_S \\
C_{\text{distinct}(R)} &= \{\!|\, (\vec{r}, \bigvee \{\phi \mid (\vec{r}, \phi) \in C_R\}) \mid (\vec{r}, \cdot) \in C_R \,|\!\} \\
C_{R-S} &= \{\!|\, (\vec{r}, \phi \wedge \psi) \mid (\vec{r}, \phi) \in C_{\text{distinct}(R)}, \\
&\qquad \text{if } (\vec{r}, \pi) \in C_{\text{distinct}(S)} \text{ then } \psi := \neg\pi \\
&\qquad \text{else } \psi := \text{true} \,|\!\}
\end{aligned}$$

Fig. 1. Relational algebra on c-tables.

$$C_{\text{Shipping}} = \{\!|\, ((NY, X_2), \textit{true}), ((LA, X_4), \textit{true}) \,|\!\}.$$

The relational algebra query is

$$\pi_{\text{Price}}(\sigma_{\text{ShipTo=Dest}}(\sigma_{\text{Cust}='Joe'}(\text{Order}) \times \sigma_{\text{Duration} \geq 7}(\text{Shipping}))).$$

We compute

$$\begin{aligned}
C_{\sigma_{\text{Cust}='Joe'}(\text{Order})} &= \{\!|\, ((Joe, NY, X_1), \textit{true}) \,|\!\} \\
C_{\sigma_{\text{Duration} \geq 7}(\text{Shipping})} &= \\
&\quad \{\!|\, ((NY, X_2), X_2 \geq 7), ((LA, X_4), X_4 \geq 7) \,|\!\} \\
C_{\sigma_{\text{Cust}='Joe'}(\text{Order}) \times \sigma_{\text{Duration} \geq 7}(\text{Shipping})} &= \\
&\quad \{\!|\, ((Joe, NY, X_1, NY, X_2), X_2 \geq 7), \\
&\qquad ((Joe, NY, X_1, LA, X_4), X_4 \geq 7) \,|\!\}
\end{aligned}$$

C. Probabilistic c-tables; expectations

A *probabilistic c-table* (cf. [7], [11]) is a c-table in which each variable is simply considered a (discrete or continuous) *random variable*, and a joint probability distribution is given for the random variable. As a convention, we will denote the discrete random variables by $\vec{X}$ and the continuous ones by $\vec{Y}$. Throughout the paper, we will always assume without saying that *discrete random variables have a finite domain*.

We will assume a suitable function $p(\vec{X} = \vec{x}, \vec{Y} = \vec{y})$ specifying a joint distribution which is essentially a PDF on the continuous and a probability mass function on the discrete variables. To clarify this, p is such that we can define the expectation of a function q as

$$E[q] = \sum_{\vec{x}} \int_{y_1} \cdots \int_{y_n} p(\vec{x}, \vec{y}) \cdot q(\vec{x}, \vec{y})\, d\vec{y} \approx \frac{1}{n} \cdot \sum_{i=1}^{n} q(\vec{x}_i, \vec{y}_i)$$

given samples $(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)$ from the distribution p.

We can specify events (sets of possible worlds) via Boolean conditions ϕ that are true on a possible world (given by assignment) θ iff the condition obtained by replacing each variable x occurring in ϕ by $\theta(x)$ is true. The characteristic function χ_ϕ of condition (event) ϕ returns 1 on a variable assignment if it makes ϕ true and returns zero otherwise. The probability $\Pr[\phi]$ of event ϕ is simply $E[\chi_\phi]$.

The expected sum of a function h applied to the tuples of a table R,

```
select expected_sum(h(*)) from R;
```

can be computed as

$$E\Big[\sum_{\vec{t}\in R} h(\vec{t})\Big] = E\Big[\sum_{(t,\phi)\in C_R} \chi_\phi \cdot h(t)\Big] = \sum_{(t,\phi)\in C_R} E\Big[\chi_\phi \cdot (h \circ t)\Big]$$

(the latter by linearity of expectation). Here $t(\vec{x},\vec{y})$ denotes the tuple t, where any variable that may occur is replaced by the value assigned to it in $(\vec{x},\vec{y})$.

Example 2.2: Returning to our running example, for $C_R = \{\!|\, (x_1, x_2 \geq 7)\,|\!\}$*, the expected sum of prices is*

$$\sum_{(t,\phi)\in C_R} \int_{x_1} \cdots \int_{x_4} p(\vec{x}) \cdot \chi_\phi(\vec{x}) \cdot t(\vec{x}).\text{Price}\; d\vec{y} = \int_{x_1} \cdots \int_{x_4} p(\vec{x}) \cdot \chi_{X_2 \geq 7}(\vec{x}) \cdot x_1 \; d\vec{y}.$$

Counting and group-by. Expected count aggregates are obviously special cases of expected sum aggregates where h is a constant function 1. We generally consider expected sum aggregates with grouping by (continuously) uncertain columns to be of doubtful value. Group-by on nonprobabilistic columns (i.e., which contain no random variables) poses no difficulty in the c-tables framework: the above summation simply proceeds within groups of tuples from C_R that agree on the group columns. In particular, by delaying any sampling process until after the relational algebra part of the query has been evaluated on the c-table representation, we find it easy to create as many samples as we need for each group in a goal-directed fashion. This is a considerable strong point of the c-tables approach used in PIP.

III. Design of the PIP System

Representing the uncertain components of a query's output symbolically as a c-table makes a wide variety of integration techniques available for use in evaluating the statistical characteristics of the expression. If our risk-management application assumes a general model of customer profit and customer satisfaction that relies on queries to create correlations between them, the sampler can detect this lack of dependency, estimate profit and probability of dissatisfaction separately, and combine the two afterwards. Even with relatively straightforward integration techniques, additional knowledge of this form has a profoundly positive impact on the efficiency and accuracy with which expectations of query results can be computed.

Accuracy is especially relevant in cases where the integral has no closed form and exact methods are unavailable. This is the case in a surprising range of practical applications, even when strong simplifying assumptions are made about the input data. For example, even if the input data contains only independent variables sampled from well-studied distributions (e.g., the normal distribution), it is still possible for queries to create complex statistical dependencies in their own right. It is well known, at least in the case of discrete and finite probability distributions, that relational algebra on block-independent-disjoint tables can construct any finite probability distribution [15], [9].

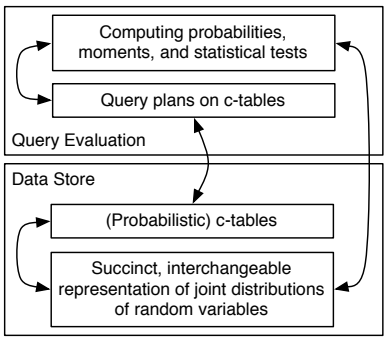


Fig. 2. Pip Query Engine Architecture

A. Symbolic Representation

PIP represents probabilistic data values symbolically via random variables defined in terms of parametrized probability distribution classes. PIP supports several generic classes of probability distributions (e.g., Normal, Uniform, Exponential, Poisson), and may be extended with additional classes. Variables are treated as opaque while they are manipulated by traditional relational operators. The resulting symbolic representation is a c-table. As the final stage of the query, special operators defined within PIP compute expectations and moments of the uncertain data, or sample the data to generate histograms.

These expectation operators are invoked with a lossless representation of the expression to be evaluated. Because the variables have been treated as opaque, the expectation operator can obtain information about the distribution a variable corresponds to. Similarly, the lossless representation allows on-the-spot generation of samples if necessary; There is no bias from samples shared between multiple query runs.

Developers can (but need not) provide supplemental information (e.g., functions defining the PDF and the CDF) about distributions they extend PIP with. The operator can exploit this additional information to accelerate the sampling process, or potentially even sidestep it entirely. For example, if a query asks for the probability that a variable will fall within specified bounds, the expectation operator can compute it with at most two evaluations of the variable's CDF.

Because the symbolic representation PIP uses is lossless, intermediate query results or views may be materialized. Expectations of values in these views or subsequent queries based on them will not be biased by estimation errors introduced by materializing the view. This is especially useful when a significant fraction of query processing time is devoted to managing deterministic data (eg, to obtain parameters for the model's variables). Not only can this accelerate processing of commonly used subqueries, but it makes online sampling feasible; the sampler need not evaluate the entire query from scratch to generate additional samples.

Example 3.1: Consider the running example in the context of c-tables. The result of the relational algebra part of the

example query can be easily computed as

R	*Price*	*Condition*
	Y_1	$Y_2 \geq 7$

without looking at p. This c-table compactly represents all data still relevant after the application of the relational algebra part of the query, other than p, which remains unchanged. Sampling from R to compute

```
select expected_sum(Price) from R;
```

is a much more focused effort. First, we only have to consider the random variables relating to Joe; but determining that random variable Y_2 is relevant while Y_4 is not requires executing a query involving a join. We want to do this query first, before we start sampling.

Second, assume that delivery times are independent from sales volumes. Then we can approximate the query result by first sampling an Y_2 value and only sampling an Y_1 value if $Y_2 \geq 7$. Otherwise, we use 0 as the Y_1 value. If $Y_2 \geq 7$ is relatively rare (e.g., the average shipping times to NY are very slow, with a low variance), this may reduce the amount of samples for Y_1 that are first computed and then discarded without seeing use considerably. If CDFs are available, we can of course do even better.

B. Random Variables

At the core of PIP's symbolic representation of uncertainty is the random variable. The simplest form of a random variable in PIP consists of a unique identifier, a subscript (for multi-variate distributions), a distribution class, and a set of parameters for the distribution.

For example, we write

$$[Y \Rightarrow Normal(\mu, \sigma^2)]$$

to represent a normally distributed random variable X with mean μ and standard deviation σ^2. Multivariate distributions may be specified using array notation. For instance,

$$[Y[n] \Rightarrow MVNormal(\mu, \sigma^2, N)]$$

Thus, when instantiating a random variable, users specify both a distribution for the variable to be sampled from, and a parameter set for that distribution. As a single variable may appear simultaneously at multiple points within the database, the unique identifier is used to ensure that sampling process generates consistent values for the variable within a given sample.

Rather than storing random variables directly, PIP employs the equation datatype, a flattened parse tree of an arithmetic expression, where leaves are random variables or constants. Because an equation itself describes a random variable, albeit a composite one, we refer to equations and random variables interchangably. Observe that while we limit our implementation to arithmetic operators, any non-recursive expression may be similarly represented; The equation datatype can be used to encode any target expression accepted by PostgreSQL.

Random variable equations can be combined freely with constant expressions, both in the target clause of a select statement, and in the where clause. All targets including random variable equations are encoded as random variable equations themselves. All where clauses including random variable equations are encoded as c-table conditions, the *context* of the row. As this allows a variable to appear several times in a given context, the variable's identifier is used as part of the seed for the pseudorandom number generator used by the sampling process.

C-table conditions allow PIP to represent per-tuple uncertainty. Each tuple is tagged with a condition that must hold for the variable to be present in the table. C-table conditions are expressed as a boolean equation of *atoms*, arbitrary inequalities of random variables. The independent probability, or *confidence* of the tuple is the probability of the condition being satisfied.

Given tables in which all conditions are conjunctions of atomic conditions and the query does not employ duplicate elimination, then all conditions in the output table are conjunctions. Thus it makes sense to particularly optimise this scenario [1]. In the case of positive relational algebra with the duplicate elimination operator (i.e., we trade duplicate elimination against difference), we can still efficiently maintain the conditions in DNF, i.e., as a simple disjunction of conjunctions of atomic conditions.

Without loss of generality, the model can be limited to conditions that are conjunctions of constraint atoms. Generality is maintained by using bag semantics to encode disjunctions; Disjunctive terms are encoded as separate rows, and the DISTINCT operator is used to coalesce terms. This restriction provides several benefits. First, constraint validation is simplified; A pairwise comparison of all atoms in the clause is sufficient to catch the inconsistencies listed above. As an additional benefit, if all atoms of a clause define convex and contiguous regions in the space $\vec{x}, \vec{y}$, these same properties are also shared by their intersection.

C. Condition Inconsistency

Conditions can become *inconsistent* by combining contradictory conditions using conjunction, which may happen in the implementations of the operators selection, product, and difference. If such tuples are discovered, they may be freely removed from the c-table.

A condition is consistent if there is a variable assignment that makes the condition true. For general boolean formulas, deciding consistency is computationally hard. But we do not need to decide it during the evaluation of relational algebra operations. Rather, we exploit straightforward cases of inconsistency to clean-up c-tables and reduce their sizes. We rely on the later Monte Carlo simulation phase to enforce the remaining inconsistencies.

1) The consistency of conditions not involving variable values is always immediately apparent.
2) Conditions $X_i = c_1 \wedge X_i = c_2$ with constants $c_1 \neq c_2$ are always inconsistent.
3) Equality conditions over continuous variables $Y_j = (\cdot)$, with the exception of the identity $Y_j = Y_j$, are not inconsistent but can be treated as such (the probability

Algorithm 3.2: Checking the consistency of a condition

1) $consistencyCheck$(ConditionSet C)
2) foreach *Discrete* condition $[X = c_1] \in C$
3) if $\exists [X = c_2] \in C$ s.t. $c_1 \neq c_2$, return **Inconsistent.**
4) foreach *Continuous* variable group K *(See Section IV-A)*
5) initialize bounds map S_0 s.t. $S_0[X] = [-\infty, \infty]\ \forall X \in K$
6) while $S_t \neq S_{t-1}$ *(incrementing $t > 0$ each iteration)*
7) let $S_t = S_{t-1}$
8) foreach equation $E \in K$
9) if at most 1 variable in E is unbounded
(use the bounded variables to shrink variable bounds)
10) foreach X, $S_t[X] = S_{t-1}[X] \cap tighten_N(X, E, S_t)$
(Where N = the degree of E)
(computing bounds from some equations may be slow)
11) if $tighten_N$ has not been defined, skip E.
12) if $\exists X$ s.t. $S_t[X] = \emptyset$, return **Inconsistent.**
13) if no Eqns were skipped return **Consistent.** else return *Consistent.*

1) $tighten_1$(Variable X, Equation E, Map S)
(example of variable bounding for degree 1 polynomial)
2) Express E in normal form $aX + bY + cZ + \ldots > 0$
3) if $a > 0$, return $[-(b \cdot \max(S[Y]) + c \cdot \max(S[Z]) + \ldots)/a, \infty]$
4) if $a < 0$, return $[-\infty, -(b \cdot \max(S[Y]) + c \cdot \max(S[Z]) + \ldots)/a]$

Strong consistency guarantees returned by this algorithm are marked in **bold**, while weak ones are marked in *italics*.

mass will always be zero). Similarly, conditions $Y_j \neq (\cdot)$, with the exception of $Y_j \neq Y_j$, can be treated as true and removed or ignored.

4) Other forms of inconsistency can also be detected where it is efficient to do so.

With respect to discrete variables, inconsistency detection may be further simplified. Rather than using abstract representations, every row containing discrete variables may be exploded into one row for every possible valuation. Condition atoms matching each variable to its valuation are used to ensure mutual exclusion of each row. Thus, discrete variable columns may be treated as constants for the purpose of consistency checks. As shown in [1], deterministic database query optimizers do a satisfactory job of ensuring that constraints over discrete variables are filtered as soon as possible.

This sampling algorithm is presented in Algorithm 3.2. Due to space constraints only $tighten_1$ is presented in this paper, but all polynomial equations may be handled using a similar, albeit more complex enumeration of coefficients. The PIP implementation currently limits users to simple algebraic operators, thus all variable expressions are polynomial. However, since consistency checking is optional, more complex equations can simply be ignored.

D. Distributions

As variables are defined in terms of parametrized distribution classes, PIP's infrastructure is agnostic to the implementation of the underlying distributions. When defining a distribution, programmers need only include a mechanism for sampling from that distribution, much like [10]'s VG Functions. However, PIP is not limited to simple sampling functionality. If it is possible to efficiently compute or estimate the distribution's probability density function (PDF), cumulative distribution function (CDF), and/or inverse cumulative distribution function (CDF^{-1}), these may be included to improve PIP's efficiency.

Distribution specific values like the PDF, CDF and inverse CDF are used to demonstrate what can be achieved with PIP's framework. Further distribution-specific values like weighted-sampling, mean, entropy, and the higher moments can be used by more advanced statistical methods to achieve even better performance. The process of defining a variable distribution is described further in Section V.

Though PIP abstracts the details of a variable's distribution from query evaluation, it distinguishes between discrete and continuous distributions. As described in Section II, existing research into c-tables has demonstrated efficient ways of querying variables sampled from discrete distributions. PIP employs similar techniques when it is possible to do so.

IV. Sampling and Integration

Conceptually, query evaluation in PIP is broken up into two components: Query and Sampling. PIP relies on Postgres to evaluate queries; As described in Section II, a query rewriting pass suffices to translate c-tables relational algebra extensions into traditional relational algebra. Details on how query rewriting is implemented are provided in Section V.

As the query is being evaluated, special sampling operators in the query are used to transform random variable expressions into histograms, expectations, and other moments. Both the computation of moments and probabilities in the general case reduces to numerical integration, and a dominant technique for doing this is Monte Carlo simulation. The approximate computation of expectation

$$E[\chi_\phi \cdot (h \circ t)] = \frac{1}{n} \cdot \sum_{i=1}^{n} p(\vec{y_i}) \cdot \chi_\phi(\vec{y_i}) \cdot h(t(\vec{y_i})) \qquad (1)$$

faces a number of difficulties. In particular, samples for which χ_ϕ is zero do not contribute to an expectation. If ϕ is a very selective condition, most samples do not contribute to the summation computation of the approximate expectation. (This is closely related to the most prominent problem in online aggregation systems [8], [16], and also in MCDB).

Example 4.1: Consider a row containing the variable

$$[Y \Rightarrow Normal(\mu = 5, \sigma^2 = 10)]$$

and the condition predicate $(Y > -3)$ *and* $(Y < 2)$. *The expectation of the variable Y in the context of this row is not* 5. *Rather the expectation is taken only over samples of Y that fall in the range* $(-3, 2)$, *(coming out to approximately* 0.17*).*

A. Sampling Techniques

a) Rejection Sampling: One straightforward approach to this problem is to perform rejection sampling; sample sets are repeatedly generated until a sufficient number of viable (satisfying) samples have been obtained.

However, without scaling the number of samples taken based on $E[\chi_\phi]$, information can get very sparse and the approximate expectations will have a high relative error. Unfortunately, as the probability of satisfying the constraint drops, the work required to produce a viable sample increases. Consequently, any mechanism that can improve the chances of satisfying the constraint is beneficial.

b) Sampling using inverse CDFs: As an alternative to generator functions, PIP can also use the inverse-transform method [12]. If available, the distribution's inverse-CDF function is used to translate a uniform-random number in the range $[0, 1]$ to the variable's distribution.

This technique makes constrained sampling more efficient. If the uniform-random input is selected from the range $[CDF(a), CDF(b)]$, the generated value is guaranteed to fall in the range $[a, b]$. Even if precise constraints can not be obtained, this technique still reduces the volume of the sampling space, increasing the probability of getting a useful sample.

c) Exploiting independence: Prior to sampling, PIP subdivides constraint predicates into minimal independent subsets; sets of predicates sharing no common variables. When determining subset independence, variables representing distinct values from a multivariate distribution are treated as the set of all of their component variables. For example, consider the one row c-table

R	ϕ_2
	$(Y_1 > 4) \wedge ([Y_1 \cdot Y_2] > Y_3) \wedge (A < 6)$

In this case, the atoms $(Y_1 > 4)$ and $([Y_1 \cdot Y_2] > Y_3)$ form one minimal independent subset, while $(A < 6)$ forms another.

Because these subsets share no variables, each may be sampled independently. Sampling fewer variables at a time not only reduces the work lost generating non-satisfying samples, but also decreases the frequency with which this happens.

d) Metropolis: A final alternative available to PIP, is the Metropolis algorithm [13]. Starting from an arbitrary point within the sample space, this algorithm performs a random walk weighted towards regions with higher probability densities. Samples taken at regular intervals during the random walk may be used as samples of the distribution.

The Metropolis algorithm has an expensive startup cost, as there is a lengthy 'burn-in' period while it generates a sufficiently random initial value. Despite this startup cost, the algorithm typically requires only a relatively small number of steps between each sample. We can estimate the work required for both Metropolis and Naive rejection sampling.

$$W_{metropolis} = C_{burn\ in} + [\#\ samples] \cdot C_{steps\ per\ sample}$$

$$W_{naive} = \frac{1}{1 - P[reject]} \cdot [\#\ samples]$$

By generating a small number of samples for the subgroup, PIP can generate a rough estimate of $P[reject]$ and decide which approach is less expensive.

B. Row-Level Sampling Operators

Evaluating a query on a probabilistic table (or tables) produces as output another probabilistic table. Though the raw probabilistic data has value, the ultimate goal is to compute statistical metrics: expectations, moments, etc. To achieve this goal, PIP provides a set of sampling operators: functions that convert probabilistic data into deterministic metrics.

Example 4.2: Consider the c-tables

R	*A*	ϕ_1
	5	$(Y_1 > 4)$

S	*B*	ϕ_2
	X	$(Y_2 > 2)$

The query

```
select A * B as C from R, S;
```

produces the result table

T	*C*	ϕ
	$5 \cdot X$	$(Y_1 > 4) \wedge (Y_2 > 2)$

A human may find it useful to know that for a given possible world T contains one row with C equal to $5 \cdot Y_1$ in worlds described by variables $Y_1 > 4$ and $Y_2 > 2$, and is empty in all other worlds. However, analysis of large volumes of data in this form requires the ability to summarize and/or aggregate. Analyzing a table containing several hundred or more rows of this form becomes easier if a histogram is available.

Sampling operators take in an expression to be evaluated and the expression's context, or boolean formula of constraints associated with the expression's row and output a histogram, expectation, or higher moment for the expression under the constraints of its context. As described in Section II, without loss of generality, it is possible to express the context as a set of conditions to be combined conjunctively.

This paper focuses predominantly on sampling operators that follow *per-row sampling semantics*. Under these semantics, each row is sampled independently. The metric in question is taken for only over the volume of probability space defined by the expression's context for each row. For example, in the case of Monte-Carlo sampling, samples are generated for each row, but only samples satisfying the row's context are considered. All other samples are discarded. If the context is unsatisfiable, a value of NAN will result.

The choice to focus on per-row sampling operators is motivated by efficiency concerns. If the results table is larger than main memory, the sampling process can become IO-bound. Per-row sampling operators require only a single pass (or potentially a second pass if additional precision is required) over the results. While we do consider table-wide sampling semantics out of necessity for some aggregates, the development of additional table-wide techniques is beyond the scope of this paper.

Note that the resulting metrics are still probabilistic data. For example, the expectation of a given cell is computed in the context of the cell's row; The expectation is computed only over those worlds where the row's condition holds, as in all other worlds the row does not exist. The conf operator is used to compute the probability of satisfying the row's condition, a value known as the row's confidence. If the conf operator is present, all conditions applying to the row are removed from the result and the resulting table is deterministic.

PIP's sampling process, including all techniques described above in Section IV-A is summarized in Algorithm 4.3. Despite the limited number of sampling techniques employed, this algorithm demonstrates the breadth of state available to the PIP framework at sample-time. Independent group sampling requires set of constraints. CDF sampling requires

Algorithm 4.3: The expectation operator; Given an expression and context condition, compute the expression's expectation given that the condition is true and optionally also compute the probability $P[C]$ of the condition being true. If sampling is necessary, both values are computed with ϵ, δ precision.

1) $expectation$(Expression E, Condition C, Bool $getP$, Goal $\{\epsilon, \delta\}$)
2) let $\bar{X}$ = the set of variables in E
3) let $target = \sqrt{2} \cdot \text{erf}^{-1}(1-\epsilon)$
4) let $N = 0; Sum = 0; SumSq = 0$
5) foreach variable group $K \in C$ s.t. $\exists X \in K \wedge X \in \bar{X}$
6) let $Count[K] = 0$; $Sampler[X]$ =Natural $\forall X \in K$
7) $consistencyCheck(K)$ *(See Alg 3.2)*
(save the bounds map S generated by consistencyCheck())
8) If inconsistent, return (NAN, 0)
(if a given variable has bounds, try to sample within those bounds)
9) foreach X s.t. $S[X] \neq [-\infty, \infty] \wedge X$ has CDF/CDF^{-1} functions
10) $Sampler[X]$ = CDF
11) (keep sampling until we have enough samples for ϵ, δ precision.)
12) while $\left[target \cdot \left|(\frac{Sum}{N})^2 - \frac{SumSq}{N}\right| + \frac{Sum}{N}\right] < [\delta \cdot Sum]$ AND $N < 1/delta$
13) N = N+1
(We only need samples for variables in both E and C)
14) foreach variable group $K \in C$ s.t. $\exists X \in K \wedge X \in \bar{X}$
15) if $Sampler[X]$ = Metropolis for any $X \in K$
16) run metropolis over saved state for group K
17) else do...
18) Count[K] = Count[K]+1
19) if $(Count[K] - N)/Count[K] >$ Metropolis Threshold
20) if all X have a PDF function
21) Sampler[X] = Metropolis $\forall X \in K$
22) scan for start point to initialize Metropolis state for K
23) if unable to find a start point return (NAN, 0)
24) continue to next K *(line 14)*
25) else return (NAN, 0)
26) generate one instance of all $X \in K$ using $Sampler[X]$
27) ... while samples do not satisfy K
28) update $Sum = Sum + E[\bar{X}]$, $SumSq = SumSq + E[\bar{X}]^2$
29) let $Prob = \prod_K \frac{N}{Count[K]}$ $\forall K$ not sampled via Metropolis
(We've gotten some work towards $P[C]$ for free already)
(If the we actually care about it, we might have more work)
30) if $getP$ = True
(Variables in C but not E haven't been processed yet)
(Also, Metropolis doesn't give us a probability)
31) foreach variable group $K \in C$ s.t. $\nexists X \in K \wedge [X \in \bar{X} \vee Sampler[X] =$ Metropolis]
32) if $|K|_{Vars} = 1$ AND $X \in K$ has a CDF function
33) use CDF to integrate $P[K]$
34) else integrate by sampling as above *(w/o metropolis)*
35) update $Prob = Prob \cdot P[K]$
36) return $(\frac{Sum}{N}, Prob)$ (P is ignored if $getP$ = False)

distribution-specific knowledge. Metropolis sampling requires similar knowledge, and also employs bounds on $P[reject]$ to make efficiency decisions. All of this information is available to the expectation operator, making it the ideal place to implement these, as well as more advanced optimization techniques.

C. Aggregate Sampling Operators

Aggregate operators (eg. sum, avg, stddev) applied to c-tables introduce a new form of complexity into the sampling process: the result of an aggregate operator applied to a c-table is difficult to represent and sample from. Even if the values being aggregated is a constant, each row's context must be evaluated independently. The result is 2^n possible outputs, each with a linear number of conditions in the number of rows. If the values being aggregated are variable expressions, the result is an identical number of outputs, each containing data linear in the size of the table.

This added complexity, coupled with the frequency with which they appear at the root of a query plan, makes them an ideal point at which to perform sampling. As aggregates compute expectations over entire tables, the probability of a given row's presence in the table can be included are included in the aggregate's expectation. This behavior is termed *per-table sampling semantics*.

We begin with the simplest form of aggregate expectation, that of an aggregate that obeys linearity of expectation ($E[f(\vec{y})] = f(E[\vec{y}])$), such as sum(). Such aggregates are straightforward to implement: per-row expectations of $f(\vec{y})\chi(\vec{y})$ are computed, and aggregated (e.g., summed up). Of note however, is the effect that the operator has on the variance of the result. In the case of sum(), each expectation can be viewed as a Normally distributed random variable with a shared, predetermined variance. By the law of large numbers, the sum of a set of N random variables with equal standard deviation σ has a variance of $\frac{\sigma}{\sqrt{N}}$. In other words, when computing the expected sum of N variables, we can reduce the number of samples taken for each individual element by a factor of $\frac{1}{\sqrt{N}}$.

If the operator does not obey linearity of expectation (e.g., the max aggregate), the aggregate implementation is made more difficult. Any aggregate may still be implemented naively by evaluating it in parallel on a set of sample worlds instantiated prior to evaluation. This is a worst-case approach to the problem; it may be necessary to perform a second pass over the results if an insufficient number of sample worlds are generated. However, more efficient special case aggregates, specifically designed to compute expectations are possible.

For example, consider the max() aggregate. If the target expression is a constant, this aggregate can be implemented extremely efficiently. Given a table sorted by the target expression in descending order, PIP estimates the probability that the first element in the table (the highest value) is present. The aggregate expectation is initialized as the product of this probability and the first element. The second term is maximal only if the first term is not present; when computing the probability of the second term, we must compute the probability of all the second term's constraint atoms being fulfilled while at least one of the first atom's terms is not fulfilled. Though the complexity of this process is exponential in the number of rows, the probability of each successive row being maximal drops exponentially.

Example 4.4: To illustrate this, consider the table (annotated with probabilities)

R	A	ϕ	$P[\phi]$
	5	$X \geq 7$	*0.7*
	4	$Y \geq 7$	*0.8*
	1	$Z \geq 7$	*0.3*
	0	$Q \geq 7$	*0.6*

Based on the probabilities listed above,

$$E[\max(A)] = 5 \cdot 0.7 + 4 \cdot 0.8 + 1 \cdot 0.3 + 0 \cdot 0.6$$

However, if the desired precision is 0.1*, we can stop scanning*

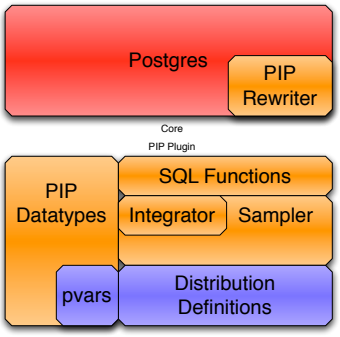


Fig. 3. The PIP Postgres plugin architecture

after the second record since the maximum any later record can change the result is $1-(1-0.7)*(1-0.8)=0.056$.

V. Implementation

In order to evaluate the viability of PIP's c-tables approach to continuous variables, we have implemented an initial version of PIP as an extension to the PostgreSQL DBMS as shown in Figure 3.

A. Query Rewriting

Much of this added functionality takes advantage of PostgreSQL's extensibility features, and can be used "out-of-the-box". For example, we define the function

CREATE_VARIABLE(*distribution*[**,***params*]**)**

which is used to create continuous variables[2]. Each call allocates a new variable, or a set of jointly distributed variables and initializes it with the specified parameters. When defining selection targets, operator overloading is used to make random variables appear as normal variables; arbitrary equations may be constructed in this way.

To complete the illusion of working with static data, we have modified PostgreSQL itself to add support for C-Table constructs. Under the modified PostgreSQL when defining a datatype, it is possible to declare it as a CTYPE; doing so has the following three effects:

- CTYPE columns (and conjunctions of CTYPE columns) may appear in the WHERE and HAVING clauses of a SELECT statement. When found, the CTYPE components of clause are moved to the SELECT's target clause.

  ```
  select *
  from   inputs
  where  X>Y and Z like '%foo'
  ```

  is rewritten to

  ```
  select *, X>Y
  from   inputs
  where  Z like '%foo'
  ```

- SELECT target clauses are rewritten to ensure that all CTYPE columns in input tables are passed through. The exception to this is in the case of special probability-removing functions. If the select statement contains one or more such functions (typically aggregates, or the conf operator), CTYPE columns are not passed through.

  ```
  select X,Y
  from   inputs
  ```

  is rewritten to

  ```
  select X,Y,inputs.phi1,inputs.phi2,...
  from   inputs
  ```

- In the case of aggregates, the mechanism by which CTYPE columns may be passed through is unclear. Thus If the select statement contains an aggregate and one or more input tables have CTYPE columns, the query causes an error unless the aggregate is labeled as a probability-removing function.
- UNION operations are rewritten to ensure that the number of CTYPE columns in their inputs is consistent. If one input table has more CTYPE columns of a given type than the other, the latter is padded with NULL constraints.

  ```
  -- left(X,phi1), right(X,phi1,phi2)
  select *
  from   left UNION right
  ```

  is rewritten to

  ```
  select *
  from   (select *,NULL AS phi2 FROM left
         ) UNION right
  ```

[2]For discrete distributions, PIP uses a repair-key operator similar to that used in [11]

R_{ctable}	A	B	ϕ
	$X*3$	5	$X>Y \wedge Y>3$
	Y	3	$Y<3 \vee X<Y$

⇓⇓⇓

R_{int}	A (VarExp)	B (integer)	ϕ_1 (CTYPE)	ϕ_2 (CTYPE)
	$X*3$	5	$X>Y$	$Y>3$
	Y	3	$Y<3$	NULL
	Y	3	$X<Y$	NULL

Fig. 4. Internal representation of C-Tables

Note that these extensions are not required to access PIP's core functionality; they exist to allow users to seamlessly use deterministic queries on probabilistic data.

PIP takes advantage of this by encoding constraint atoms in a CTYPE datatype; Overloaded $>$ and $<$ operators return a constraint atom instead of a boolean if a random variable is involved in the inequality, and the user can ignore the distinction between random variable and constant value (until the final statistical analysis).

B. Defining Distributions

PIP's primary benefit over other c-tables implementations is its ability to admit variables chosen from arbitrary continuous distributions. These distributions are specified in terms of general distribution classes, a set of C functions that describes the distribution. In addition to a small number of functions used to parse and encode parameter strings, each PIP distribution class defines one or more of the following functions.

- `Generate(Parameters, Seed)` uses a pseudorandom number generator to generate a value sampled from the distribution. The seed value allows PIP to limit the amount of state it needs to maintain; multiple calls to Generate with the same seed value produce the same sample, so only the seed value need be stored.
- `PDF(Parameters, x)` evaluates the probability density function of the distribution at the specified point.
- `CDF(Parameters, x)` evaluates the cumulative distribution function at the specified point.
- `InverseCDF(Parameters, Value)` evaluates the inverse of the cumulative distribution function at the specified point.

PIP requires that all distribution classes define a Generate function. All other functions are optional, but can be used to improve PIP's performance if provided; The supplemental

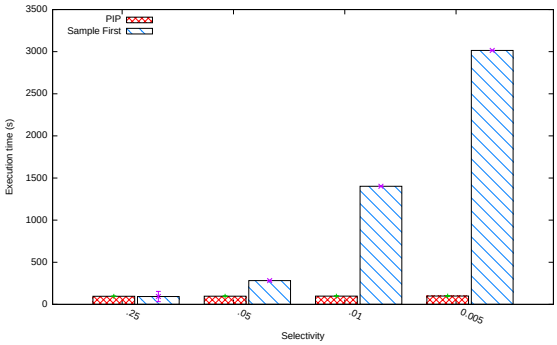


Fig. 5. Time to complete a 1000 sample query, accounting for selectivity-induced loss of accuracy.

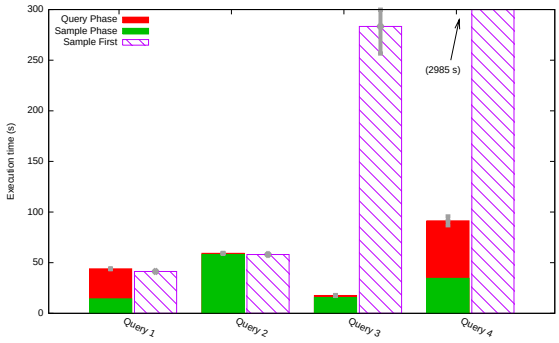


Fig. 6. Query evaluation times in PIP and Sample-First for a range of queries. Sample-First's sample-count has been adjusted to match PIP's accuracy.

functions need only be included when known methods exist for evaluating them efficiently.

C. Sampling Functionality

PIP provides several functions for analyzing the uncertainty encoded in a c-table. The two core analysis functions are conf() and expectation().

- `conf()` performs a conjunctive integration to estimate the probability of a specific row being present in the output, in effect computing the expectation $E[1]$. It identifies and extracts all lineage atoms from the row being processed and then performs the conjunctive integration over them as normal.
- `aconf()`, a variant of conf(), is used to perform general integration. This function is an aggregate that computes the joint probability of at least one aggregated row being present in the output.
- `expectation()` computes the expectation of a variable by repeated sampling. If a row is specified when the function is called, the sampling process is constrained by the constraint atoms present in the row.
- `expected_sum()`, `expected_max()` are aggregate variants of expectation. As with expectation() they can be parametrized by a row to specify constraints.
- `expected_sum_hist()`, `expected_max_hist()` are similar to the above aggregates in that they perform sampling. However, instead of outputting the average of the results, it instead outputs an array of all the generated samples. This array may be used to generate histograms and similar visualizations.

Aggregates pose a challenge for the query phase of the PIP evaluation process. Though it is theoretically possible to create composite variables that represent aggregates of their inputs, in practice it is infeasible to do so. The size of such a composite is not just unbounded, but linear in the size of the input table. A variable symbolically representing an aggregate's output could easily grow to an unmanageable level. Instead, PIP limits random variable aggregation to the sampling phase.

VI. Evaluation

As a comparison point for PIP's ability to manage continuous random variables, we have constructed a sample-first probabilistic extension to Postgres that emulates MCDB's tuple-bundle concept using ordinary Postgres rows. A sampled variable is represented using an array of floats, while the tuple bundle's presence in each sampled world is represented using a densely packed array of booleans. In lieu of an optimizer, test queries were constructed by hand so as to minimize the lifespan of either array type.

Using Postgres as a basis for both implementations places them on an equal footing with respect to DBMS optimizations unrelated to probabilistic data. This makes it possible to focus our comparison solely on the new, probabilistic functionality of both systems. However, to make the distinction from MCDB (which is a separate development not based on Postgres) explicit, we refer to our implementation of the MCDB approach as Sample-First.

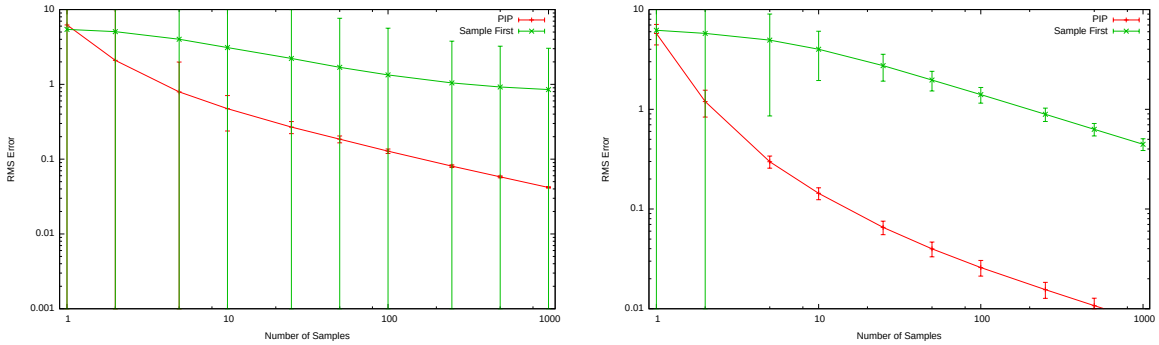
We evaluated both the PIP C-Tables and the Sample-First infrastructure against a variety of related queries. Tests were run over a single connection to a modified instance of PostgreSQL 8.3.4 with default settings running on a 2x4 core 2.0 GHz Intel Xeon with a 4MB cache. Unless otherwise specified queries were evaluated over a 1 GB database generated by the TPC-H benchmark, all sampling processes generate 1000 samples apiece, and results shown are the average of 10 sequential trials with error bars indicating one standard deviation.

First, we demonstrate PIP's performance on a simple set of queries ideally suited to the strengths of Sample-First. These two queries (identical to Q1 and Q2 from [10]) involve parametrizing a table of random values, applying a simple set of math operations to the values, and finally estimating the sum of a large aggregate over the table.

The first query computes the rate at which customer purchases have increased over the past two years. The percent increase parametrizes a Poisson distribution that is used to predict how much more each customer will purchase in the coming year. Given this predicted increase in purchasing, the query estimates the company's increased revenue for the coming year.

In the second query, past orders are used to compute the mean and standard deviation of manufacturing and shipping times. These values parametrize a pair of Normal distributions that combine to predict delivery dates for each part ordered today from a Japanese supplier. Finally, the query computes the maximum of these dates, providing the customer with an estimate of how long it will take to have all of their parts delivered.

The results of these tests are shown as query Q_1 and Q_2, respectively, in Figure 6. Note that performance times for PIP are divided into two components: query and sample, to



(a) (b)

Fig. 7. RMS error across the results of 30 trials of (a) a simple group-by query Q_4 with a selectivity of 0.005, and (b) a complex selection query Q_5 with an average selectivity of 0.05.

distinguish between time spent evaluating the deterministic components of a query and building the result c-table, and time spent computing expectations and confidences of the results. The results are positive; the overhead of the added infrastructure is minimal, even on queries where Sample-First is sufficient. Furthermore, especially in Q2, the sampling process comprises a relatively small portion of the query; additional samples can be generated without incurring the nearly 1 minute query time.

The third query Q_3 in Figure 6 combines a simplified form of queries Q_1 and Q_2. Rather than aggregating, the query compares the delivery times of Q_2 to a set of "satisfaction thresholds." This comparison results in a (probabilistic) table of dissatisfied customers that is used in conjunction with Q_1's profit expectations to estimate profit lost to dissatisfied customers. A query of this form might be run on a regular basis, perhaps even daily. As per this usage pattern, we pre-materialize the component of this query unlikely to change on a daily basis: the expected shipping time parameters.

Though PIP and Sample-First both take the same amount of time to generate 1000 samples under this query, the query's selectivity causes Sample-First to disregard a significant fraction of the samples generated; for the same amount of work, Sample-First generates a less accurate answer. To illustrate this point, see Figure 7(a). This figure shows the RMS error, normalized by the correct value in the results of a query for predicted sales of 5000 parts in the database, given a Poisson distribution for the increase in sales and a popularity multiplier chosen from an exponential distribution. As an additional restriction, the query considers only the extreme scenario where the given product has become extremely popular (resulting in a selectivity of $e^{-5.29} \approx 0.005$).

RMS error was computed over 30 trials using the algebraically computed correct value as a mean, and then averaged over all 5000 parts. Note that PIP's error is over two orders of magnitude lower than the sample-first approach for a comparable number of samples. This corresponds to the selectivity of the query; as the query becomes more selective, the sample-first error increases. Furthermore, because CDF sampling is used to restrict the sampling bounds, the time taken by both approaches to compute the same number of samples is equivalent.

A similar example is shown in Figure 7(b). Here, a model is constructed for how much product suppliers are likely to be able to produce in the coming year based on an Exponential distribution, and for how much product the company expects to sell in the coming year as in Q1. From this model, the expected underproduction is computed, with a lower bound of 0; the selection criterion considers only those worlds where demand exceeds supply. For the purposes of this test, the model was chosen to generate an average selectivity of 0.05. Though the comparison of 2 random variables necessitates the use of rejection sampling and increases the time PIP spends generating samples, the decision to drop a sample is made immediately after generating it; PIP can continue generating samples until it has a sufficient number, while the Sample-First approach must rerun the entire query.

Note the relatively large variance in the RMS error of the Sample-First results these figures, particularly the first one. Here, both the selectivity and the price for each part vary with the part. Thus, some parts become more important while others become harder to sample from. In order to get a consistent answer for the entire query Sample-First must provision enough samples for the worst case, while PIP can dynamically scale the number of samples required for each term.

Returning to Figure 6, Queries Q_3 and Q_4 have been run with PIP at a fixed 1000 samples. As Sample-First drops all but a number of samples corresponding to the selectivity of the query, we run Sample-First with a correspondingly larger number of samples. For Query 3, the average selectivity of 0.1 resulted in Sample-First discarding 10% of its samples. To maintain comparable accuracies, Sample-First was run at 10,000 samples.

We expand on this datapoint in Figure 5 where we evaluate Q_4, altered to have varying selectivities. The sample-first tests are run with $\frac{1}{selectivity}$ times as many samples as PIP to compensate for the lower error, in accordance with Figure 7(a). Note that selectivity is a factor that a user must be aware of when constructing a query with sample-first while PIP is able to account for selectivity automatically, even if rejection sampling is required.

It should also be noted that both of these queries include

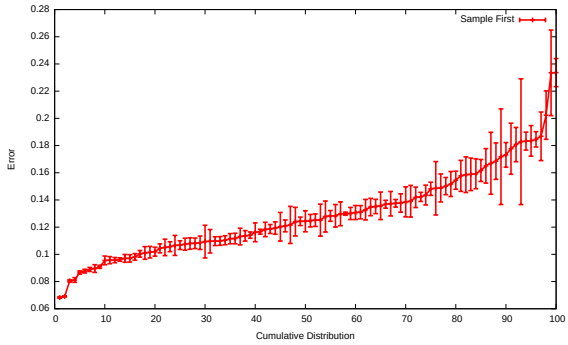


Fig. 8. Sample-First error as a fraction of the correct result in a danger-estimation query on the NSIDC's Iceberg Sighting Database. PIP was able to obtain an exact result.

two distinct, independent variables involved in the expectation computation. A studious user may note this fact and hand optimize the query to compute these values independently. However, without this optimization, a sample-first approach will generate one pair of values for each customer for each world. As shown in the RMS error example, an arbitrarily large number of customer revenue values will be discarded and the query result will suffer. In this test, customer satisfaction thresholds were set such that an average of 10% of customers were dissatisfied. Consequently sample-first discarded an average of 10% of its values. To maintain comparable accuracies, the sample-first query was evaluated with 10,000 samples while the PIP query remained at 1000 samples.

As a final test, we evaluated both PIP and our Sample-First implementation on the NSIDC's Iceberg Sighting Database[19] for the past 4 years. 100 virtual ships were placed at random locations in the North Atlantic, and each ship's location was evaluated for its proximity to potential threats; Each iceberg in the database was assigned a normally distributed position relative to its last sighting, and an exponentially decaying danger level based on time since last sighting. Recently sighted icebergs constituted a high threat, while historic sightings represented potential new iceberg locations. The query identified icebergs with greater than a 0.1% chance of being located near the ship and estimated the total thret posed by all potentially nearby icebergs. The results of this experiment are shown in Figure VI. PIP was able to employ CDF sampling and obtain an exact result within 10 seconds. By comparison, the Sample-First implementation generating 10,000 samples took over 10 minutes and produced results deviating by as much as 25% from the correct result on average.

VII. Conclusion

We have shown that symbolic representations of uncertainty like C-Tables can be used to make the computation of expectations, moments, and other statistical measures in probabilistic databases more accurate and more efficient. The availability of the expression being measured enables a broad range of sampling techniques that rely on this information and allows more effective selection of the appropriate technique for a given expression.

We have shown that the use of symbolic representations can be exploited to significantly reduce query processing time and significantly improve query accuracy for a wide range of queries, even with only straightforward algorithms. For the remaining queries, we have shown that the overhead created by the symbolic representation has only a negligible impact on query processing time. This, combined with PIP's extensibility, make it a powerful platform for evaluating a wide range of queries over uncertain data.

References

[1] L. Antova, T. Jansen, C. Koch, and D. Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.

[2] L. Antova, C. Koch, and D. Olteanu. "10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information". In *Proc. ICDE*, 2007.

[3] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. "Evaluating Probabilistic Queries over Imprecise Data". In *Proc. SIGMOD*, pages 551–562, 2003.

[4] N. Dalvi and D. Suciu. "Efficient query evaluation on probabilistic databases". *VLDB Journal*, **16**(4):523–544, 2007.

[5] A. Deshpande and S. Madden. "MauveDB: supporting model-based user views in database systems". In *Proc. SIGMOD*, pages 73–84, 2006.

[6] W. Gilks, S. Richardson, and D. Spiegelhalter. *Markov Chain Monte Carlo in Practice: Interdisciplinary Statistics*. Chapman and Hall/CRC, 1995.

[7] T. J. Green and V. Tannen. "Models for Incomplete and Probabilistic Information". In *International Workshop on Incompleteness and Inconsistency in Databases (IIDB)*, 2006.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[9] T. Imielinski and W. Lipski. "Incomplete information in relational databases". *Journal of ACM*, **31**(4):761–791, 1984.

[10] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. "MCDB: A Monte Carlo approach to managing uncertain data". In *Proc. ACM SIGMOD Conference*, pages 687–700, 2008.

[11] C. Koch. "MayBMS: A system for managing large uncertain and probabilistic databases". In C. Aggarwal, editor, *Managing and Mining Uncertain Data*, chapter 6. Springer-Verlag, Feb. 2009.

[12] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 1982.

[13] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H.Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6), June 1953.

[14] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(335), 1949.

[15] C. Ré and D. Suciu. Materialized views in probabilistic databases: for information exchange and query optimization. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 51–62. VLDB Endowment, 2007.

[16] F. Rusu, F. Xu, L. L. Perez, M. Wu, R. Jampani, C. Jermaine, and A. Dobra. The dbo database system. In *SIGMOD Conference*, pages 1223–1226, 2008.

[17] P. Sen and A. Deshpande. "Representing and Querying Correlated Tuples in Probabilistic Databases". In *Proc. ICDE*, pages 596–605, 2007.

[18] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. "Orion 2.0: native support for uncertain data". In *Proc. ACM SIGMOD Conference*, pages 1239–1242, 2008.

[19] N. Snow and I. D. C. D. C. for Glaciology. International ice patrol (iip) iceberg sightings database. Digital Media, 2008.

[20] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. "BayesStore: Managing large, uncertain data repositories with probabilistic graphical models". In *VLDB '08, Proc. 34th Int. Conference on Very Large Databases*, 2008.

[21] J. Widom. "Trio: a system for data, uncertainty, and lineage". In C. Aggarwal, editor, *Managing and Mining Uncertain Data*. Springer-Verlag, Feb. 2009.