PIP: A Database System for Great and Small Expectations*

ABSTRACT

We describe PIP, a probabilistic database system that combines the strengths of recent discrete systems such as MystiQ and MayBMS with the generality of the Monte-Carlo approach of MCDB. It supports both discrete and continuous probability distributions, powerful correlations definable by queries, expectations of aggregates and distinct-aggregates with or without group-by, and the computation of confidences. PIP uses c-tables to delay and minimize the amount of sampling work required. We study Monte-Carlo sampling and integration and present a Karp-Luby style importance sampling algorithm for continuous distributions. This technique is essential in scenarios where very small probabilities have to be approximated to a small relative error, as is often the case for the computation of conditional probabilities (as ratios of probabilities). We also provide experimental evidence for the competitiveness of our approach, comparing PIP with a reimplementation of the refined sample-first approach taken by MCDB.

1. INTRODUCTION

Uncertain data comes in many forms. For example, sensor readings with a margin of error or databases containing predictions describing a range of possible outcomes. Traditional database management systems (DBMS) are ill-equipped to manage uncertain data. For example, consider a predictive model for customer orders based on trends extrapolated from historical data. While a traditional DBMS can process queries on a sampled order database (one possible concrete database without uncertainty in it) if it is somehow instantiated from the model, it is not equipped to provide statistical information about quality of the estimates it generates as a result.

Probabilistic database management systems [4, 25, 12, 21, 22, 9, 24] aim at addressing this issue; the goal is to provide better support for querying uncertain data. The core problem in query processing in probabilistic databases is computing moments and histograms for uncertain values. This includes as special cases the computation of expectations of aggregates (for example, the expected aggregate revenues of a company for the next quarter) and the computation of probabilities of events definable by queries (e.g., the probability that the total sales will exceed a certain amount).

Technically, computing moments in the continuous case calls for the numerical integration of rather complicated functions. Assume that uncertainty is encoded via a number of random variables $X_1 \ldots X_k$ for which a probability density function (PDF) p is given. Computing the expectation of a query given by a function $q(\vec{X})$ is equivalent to computing the integral

$$E[q] = \int_{x_1=-\infty}^{\infty} \cdots \int_{x_k=-\infty}^{\infty} p(\vec{x}) \cdot q(\vec{x})\, d\vec{x}. \qquad (1)$$

Computing the probability of an event defined by a Boolean query can be thought of as a special case of the above where the query expressing events of interest defines a function q that maps to values 0 and 1 only.

In most practical cases, PDFs defined by queries are complicated functions for which no closed-form integrals are known. This is true even if very strong simplifying assumptions are made about the input data – for example, that the individual uncertain values are independent from each other and use well-studied distributions such as normal distributions: the queries create complex statistical dependencies in their own right.

Monte Carlo integration. There is one conceptually simple technique, however, that allows for the (approximate) numerical integration of even the most general functions, including those occurring in probabilistic data processing: Monte Carlo integration [15]. Conceptually, to compute an expectation, one simply approximates the integral by taking n samples $\vec{x}_1, \ldots, \vec{x}_n$ for $\vec{X}$ from p and taking the average of the q values,

$$\frac{1}{n} \cdot \sum_{i=1}^{n} q(\vec{x}_i). \qquad (2)$$

In general, however, even just taking a sample from a complicated PDF is difficult. Markov-chain Monte Carlo (MCMC, cf. e.g., [5]) is a class of extremely powerful techniques for doing this of which the Metropolis-Hastings technique is possibly the best-known example [14, 5]. MCMC is used pervasively in science, for instance in simulating physical processes or reconstructing genomes from short sequence data. It is also extensively used in a new generation of Arti-

*The system is named after Philip Pirrip, nicknamed Pip, the protagonist of Charles Dickens' novel Great Expectations. This is a temporary name for double-blind reviewing.

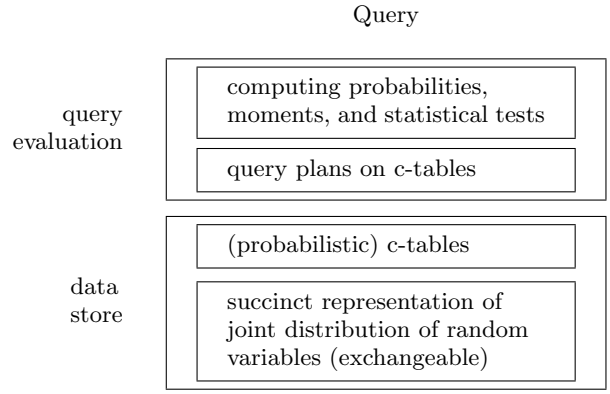

Figure 1: Pip Query Engine Architecture.

ficial Intelligence probabilistic inference systems whose aims are somewhat related to those of probabilistic database systems [19, 16].

Monte Carlo in the discrete case. Computing probabilities and expectations is by no means only difficult in the continuous case. In the discrete and finite case (each uncertain value can only take a finite number of possible values), the difficulty of computing the probabilities of events (e.g., that a particular tuple occurs in the query result) follows from the fact that the probability mass in discrete distributions is concentrated in *several* distinct places – similarly as in multimodal continuous distributions. As a consequence, Monte Carlo simulation with relatively few samples leads to low-quality approximations of probabilities and moments. In fact, it follows from [10, 11, 6] that approximating the probability of a tuple in a select-from-where query with duplicate elimination to within an error bound relative to the size of the probability value *must* take exponentially many Monte Carlo iterations already on a so-called *tuple-independent probabilistic database* [4] (arguably the simplest form of a discrete probabilistic database). In the discrete case, the exact computation of probabilities and moments is just a finite summation problem. However, in the general case there are exponentially many (inclusion-exclusion) terms to sum up. The problem is #P-hard and thus computationally infeasible [6, 4].

Importance sampling and error bounds. The exponentiality result just mentioned has been overcome by a nontrivial form of *importance sampling* originally due to Karp and Luby [10], which achieves relative error bounds in polynomial time (actually, with linearly many Monte-Carlo iterations). Intuitively, the sampling here is from a derived distribution over an interesting subspace of the original probability space. This technique, in more refined form [11, 3, 23], is implemented in the MystiQ and MayBMS systems [18, 13, 12]. It is only known to work for discrete probabilistic data.

Challenges of sampling-based query processing. To summarize, a Monte Carlo simulation/sampling-based approach is a natural design choice for probabilistic database systems. However, a Monte Carlo approach leaves us with the problems of efficiently creating samples (the techniques we have discussed so far call for samples that are entire databases, not just atomic values or tuples) and computing sufficiently many samples to yield satisfactory results. Additionally, we would like to know how many samples we need to get certain error bounds.

Recently, the paper [9] on the MCDB system 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 so called *VG-functions*. However, in its basic form, parallel evaluation over many sampled databases is very inefficient; as an optimization, samples are computed lazily and bundled by tuple.

Conditional tables (c-tables, [8]) 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 uses 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 [25] 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.

Using c-tables for query evaluation allows compositional transformation of probabilistic database representations, thus allowing a single c-table to sumultaneously represent all possible worlds (sample databases) at once. Samples can then be drawn as late in the query evaluation process as possible, after all relational algebra work is finished. This has many advantages over sample-first approaches.

- Evaluating relational algebra on c-tables is essentially as efficient as processing the same algebra queries on a single MCDB sample. By evaluating query operations on the c-table, we may be able to filter out tuples that would otherwise have to be sampled before being dropped in a sample-first approach. This allows us to sample more selectively for the needs of the query and produce higher-quality results with the same amount of work as in a sample-first approach. This is particularly important for highly selective queries or aggregation queries with group-by constructs, for which relatively low numbers of samples (as MCDB produces them) result in notoriously imprecise answers.
- Computing further samples in the c-tables approach is cheaper than in MCDB, because we do so only after most of the querying work has been done. Thus, using c-tables, multiple sample sets may be produced without incurring query processing overheads more than once.

  This is particularly important in online sampling, where we want to quickly produce a first result based on few samples and then improve on the result by computing more samples until the user is satisfied with the result.

EXAMPLE 1.1. Suppose a database captures customer orders expected for the next quarter, including prices and 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 drawing a number of samples from p and using formula 2 to approximate $E[q]$, where q represents the result of the sum aggregate query on a sample, here

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

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, samples also require an effort for customer Bob, which does not contribute to the result.

Now consider using c-tables. The result of the relational algebra part of the above query can be easily computed without looking at f as

R	Price	Condition
	X_1	$X_2 \leq 7$

This c-table compactly represents all relevant 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 X_2 is relevant while X_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 X_2 value and only sampling an X_1 value if $X_2 \geq 7$. Otherwise, we use 0 as the X_1 value. If $X_2 \geq 7$ 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 X_1 that are first computed and then discarded without seeing use considerably. If CDFs are available, we can of course do even better.

In Section 6 we compare the c-tables approach to a very basic, naive sample-first approach. Note however, that [9] describes a number of optimizations that MCDB uses to obtain some of the advantages gained by using c-tables. □

The c-tables approach has never been used to build a probabilistic database management system that supports continuous probability distributions. ORION [22] 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.

In addition, we have a vision of a powerful unified approach that combines transformations of succinct representations (c-tables) and Monte-Carlo techniques (beyond the Karp-Luby algorithm for handling duplicate elimination in discrete systems), which has never been put forward before.

The Pip System is a probabilistic database system based on probabilistic c-tables that combines the strong points of recent discrete systems such as MystiQ and MayBMS with the generality of the Monte-Carlo approach of MCDB. It supports both discrete and continuous probability distributions, powerful correlations 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 study query evaluation using probabilistic conditional tables, for relational algebra, aggregates, and the computation of expectations and probabilities.
- We present the architectural and language design decisions made in the Pip system, and show how the various techniques introduced combine into a unified whole.
- We study Monte-Carlo sampling and integration. We present a Karp-Luby style importance sampling algorithm for continuous distributions. This technique is essential in scenarios where very small probabilities have to be computed up to a small relative error, as is often the case for the computation of conditional probabilities (as ratios of probabilities). *The title of this paper in part relates to this: our system can compute expectations of great quality, even if small.*
- We provide experimental evidence for the competitiveness of our approach, comparing PIP with a reimplementation of the refined sample-first approach taken by MCDB. We use a common codebase for both systems based on Postgres to enable fair comparison. We show that PIP is in general competitive to MCDB (it essentially never does more work) and can reduce the amount of work needed to compute the same number of samples substantially; in other cases, delaying sampling using c-tables to a stage where more is known about what samples are needed; This added information can lead to the same number of samples giving better quality results in PIP than in MCDB.

The structure of this paper is as follows. Section 2 presents probabilistic c-tables, our database model, and the conceptual evaluation of queries on probabilistic c-tables. Section 3 studies sampling, Monte Carlo integration, and the continuous Karp-Luby algorithm. In Section 4, we present a high level view of Pip in the abstract. Section 5 discusses details of the implementation of PIP and our reimplementation of MCDB. Finally, in Section 6, we present the outcomes of our experiments with PIP and our MCDB reimplementation.

2. PROBABILISTIC C-TABLES

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

2.1 C-tables

Conditional tables (c-tables) [8] are extensions of relational databases to handle uncertainty. A c-table over a set of variables is a relational table 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.

2.2 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 2. 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

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

and

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

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

Figure 2: Relational algebra on c-tables.

The relational algebra query is

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

We compute $C_{\sigma_{\text{Cust}=' Joe'}(\text{Order})} = \{\!|\, ((Joe, NY, X_1), \text{true}) \,|\!\}$, $C_{\sigma_{\text{Duration} \leq 7}(\text{Shipping})} = \{\!|\, ((NY, X_2), X_2 \geq 7), ((LA, X_4), X_4 \leq 7) \,|\!\}$, and $C_{\sigma_{\text{Cust}=' Joe'}(\text{Order}) \times \sigma_{\text{Duration} \leq 7}(\text{Shipping})} = \{\!|\, ((Joe, NY, X_1, NY, X_2), X_2 \leq 7), ((Joe, NY, X_1, LA, X_4), X_4 \leq 7) \,|\!\}$. The c-table for the overall result is as shown in Example 1.1. □

Note that a tuple can be removed from a c-table if its condition is inconsistent. Conditions can become inconsistent by combining contradictory conditions using conjunction, which may happen in the implementations of the operators selection, product, and difference.

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 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.
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.

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. 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.

2.3 Probabilistic c-tables; expectations

A *probabilistic c-table* 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

$$E[q] = \sum_{\vec{x}} \int_{y_1} \cdots \int_{y_n} p(\vec{x}, \vec{y}) \cdot q(\vec{x}, \vec{y}) \, d\vec{y}$$

and approximate it as

$$\frac{1}{n} \cdot \sum_{i=1}^{n} q(\vec{x}_i, \vec{y}_i)$$

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 $C_R = \{\!|\, (x_1, x_2 \leq 7) \,|\!\}$, the expected sum of prices is

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

□

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.

3. MONTE CARLO SAMPLING AND INTEGRATION

As has already been pointed out, both the computation of moments and probabilities reduces to numerical integration, and a dominant technique for doing this is Monte Carlo simulation. The approximate computation of expectation $E[\chi_\phi \cdot (h \circ t)]$,

$$\frac{1}{n} \cdot \sum_{i=1}^{n} p(\vec{x}_i) \cdot \chi_\phi(\vec{x}_i) \cdot h(t(\vec{x}_i)), \tag{3}$$

faces a number of difficulties. This section describes how these are addressed in PIP.

Basic sampling from p. It may be hard to sample from p. Pip implements MCMC (in the form of the Metropolis algorithm), a very powerful and general technique for computing samples, essentially based on random walks in a Markov Chain biased by p. Of course, this technique has a considerable overhead and is avoided where possible.

Independent random variables. In many special cases, there are better techniques, such as when the random variables are mutually independent (i.e., $p_{X_1 \ldots X_k}(x_1, \ldots, x_k) = \prod_{i=1}^{k} p_{x_i}(x_i)$). In that case, we sample independently from the various constituent distributions p_{X_i}. It should also be noted that constraint atoms may be treated as derived boolean random variables. When the value of the constraint is fixed (eg, when sampling under a selective condition) these atoms introduce dependencies between their component random variables.

Special sampling techniques. Efficient direct sampling (not using MCMC) is possible for a constituent distribution p_{X_i} if we have an inverse CDF P^{-1}. This allows us to draw a sample from p_{X_i} by drawing a sample x uniformly from [0,1] and computing $P^{-1}(x)$. We can precompute and materialize

(inverse) CDFs for later use using Monte Carlo integration itself.

Special and very efficient sampling methods are also known for certain well-studied distributions. For example, for the normal distribution, samples can be efficiently drawn using the Box-Muller transform (implemented in PIP), the Ziggurat algorithm, etc.

Sparseness of samples for selective conditions. 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. While we do not strictly employ rejection sampling here – samples for with χ_ϕ is zero count towards the number or samples n by which we average – information can get very sparse and the approximate expectations have a high relative error. (This is closely related to the most prominent problem in online aggregation systems [7, 20], and also in MCDB).

In the context of discrete probabilistic databases, the Karp-Luby algorithm addresses exactly this problem. It is a fully polynomial-time randomized approximation scheme for the probability of a DNF condition. As argued before, for positive relational algebra, assuming DNF conditions means no loss of generality or efficiency.

We now study some of these aspects in more detail. In particular, we develop a Karp-Luby style algorithm for the continuous case. The strength of this algorithm is that it only requires us to be able to approximate the probabilities of *conjunctive* conditions and sample from the joint distribution to efficiently approximate the probability of a DNF (which is important when we want to compute probabilities or expectations of aggregates after *duplicate elimination*).

3.1 Constrained Sampling

Though PIP requires that all distribution classes provide a general-purpose sampling routine, it is periodically necessary to sample a variable from a subset of its range. For example, consider a row containing the variable $Y \sim Normal(5, 10)$ and the condition atoms $(Y > -3)$ 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 values of Y that fall in the range $(-3, 2)$.

More generally, PIP requires the ability to sample values that are constrained by boolean formulas of condition atoms. This functionality is used both when computing expectations and as part of the Karp-Luby estimator. For conjunctions of atoms, this problem is equivalent to sampling from a contiguous subset of the sample space.

The most straightforward approach to this problem is to perform rejection sampling; sample sets are repeatedly generated until one is found that satisfies the constraint formula. However, as the probability of satisfying all of the atoms drops, the number of rejected samples grows. Thus the cost of rejection sampling is inversely proportional to the probability of satisfying all the atoms.

If the inverse CDF is available for a given variable, it may be used to rapidly generate samples within specified bounds. The inverse CDF is effectively a mapping from the domain $(0, 1)$ to the corresponding value in the given distribution. Furthermore, the CDF increases monotonically.

$$[x > y] \rightarrow \left[CDF^{-1}(x) > CDF^{-1}(y)\right]$$

Thus, to obtain samples constrained to $(lower, upper)$, we sample $CDF^{-1}(X)$ where X is chosen uniformly from the range $(CDF(lower), CDF(upper))$. In the unlikely event that the inverse CDF is available, but the CDF is not, this technique may still be used. Instead of sampling from the range $(CDF(lower), CDF(upper))$, we instead sample $x \in (L, H)$ where L is initialized to 0, and H is initialized to 1. If $CDF^{-1}(x) \leq lower$ we set $L = x$ and try again. Similarly, if $CDF^{-1}(x) \geq upper$ we set $H = x$ and try again. In this way, we effectively learn the values of $CDF(lower)$ and $CDF(upper)$.

When the inverse CDF is not available, naive sampling is typically the most efficient approach. The probability of a set of variables satisfying a set of atoms is increased as the number of atoms is reduced. As in conjunctive integration, it is possible to improve the success rate by separately generating samples for each minimal independent subset. Because the subsets are smaller, it is less likely that a sample will need to be discarded. Furthermore, because fewer variables are being sampled for each subset, less computational effort is wasted generating invalid samples.

3.2 MCMC Sampling

A final alternative available to PIP, albeit a rarely used one, is the Metropolis algorithm [14]. Starting from an arbitrary point within the sample space, this algorithm performs a random walk. Steps are sampled from a multivariate normal distribution, and rejection sampling is used to weight the walk towards regions with higher probability densities. Samples taken at regular intervals during the random walk may be used as samples of the distribution.

Because the Metropolis algorithm uses relative probability densities, it is possible to use a density function that has not been normalized. Thus, regions of space that do not satisfy the clause are assigned 0 density; the random walk never enters these regions.

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. Consequently, the Metropolis algorithm is ideally suited for generating large numbers of samples when the CDF is not available and the probability of sampling a given value is small.

It necessary to parametrize the normal distribution used to select the next step in the random walk. Specifically, the standard deviation is highly dependent on the distribution being sampled from; If the value selected is too small, steps taken by the algorithm will be too small and the number of steps required to generate independent samples becomes large. If the value selected is too large, the algorithm will frequently attempt to leave the constraint bounds and thus many steps will be rejected.

PIP uses the burn-in phase to select an appropriate standard deviation for each dimension. Prior to sampling, PIP performs a dynamic standard deviation computation. Steps are limited to movement along one axis; thus only one variable contributes to the step direction. If the step is rejected, the standard deviation for that axis is lowered by a constant factor. If the step is accepted, the standard deviation is raised by a constant factor. These factors are selected such that the standard deviation converges to a point where the acceptance-rejection ratio falls in the commonly accepted ideal range of $0.1 - 0.4$[17]. The dynamic standard devia-

tion computation does not replace the burn-in phase, but instead reduces the number of burn-in steps required.

3.3 Computing Confidences

Computing the confidence of a row in the C-Table; ie, computing the probability that the conjunction ϕ of the row's condition atoms, is equivalent to computing

$$\sum_{\vec{x}} \int_{\vec{y}} p(\vec{x}, \vec{y}) \cdot \chi_\phi(\vec{x}, \vec{y}) \, d\vec{y}.$$

This integral may be naively estimated via Monte Carlo sampling as described in Section 2.3. The difficulty is twofold: First, it must be possible to sample from p. Second, enforcing χ_ϕ requires rejection sampling, which can be very inefficient if ϕ is selective.

Exploiting independence. To minimize the number of variables being integrated at one time, PIP first subdivides constraint atoms into minimal independent subsets. Two constraint subsets are independent if their member atoms have no variables in common. When determining subset independence, composite random variables (for instance, defined by artithmetic expressions over random variables) are treated as the set of all of their component variables. By definition, atoms in each subset are independent. Thus, the probability of each subset may be computed independently as well; the overall probability is the product of the independent probabilities. For example, consider the one row c-table

R	ϕ_2
	$(X > 4) \wedge ([X \cdot Z] > Y) \wedge (A < 6)$

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

Because condition atoms describing discrete variables are all of the form $Var = Val$, discrete variables are handled trivially. Inconsistent values have already been removed, so the probability for the entire subgroup is the probability of the listed variable assignment.

The simplest subset of continuous atoms is one that references only one variable. In this case, the atoms in the set provide constant upper or lower bounds, and the integration problem may be solved by evaluating the variable's CDF at the tightest upper and lower bounds. If the CDF is not available or if it is not possible to derive tight bounds on the CDF, PIP can still integrate via Monte Carlo sampling. In the one-variable case, numerical integration of the variable's PDF could also be used where an extremely precise answer is required.

Sampling using inverse CDFs. With more than two variables in the independent subset, Monte Carlo sampling becomes the most effective way of estimating the subset's probability. As has already been noted, Monte Carlo techniques perform poorly if the value being computed is small. However, in some cases PIP may be able to use a variable's CDF to reduce the sampling area.

As discussed in Section 3.1, inverted CDFs allow efficient sampling of bounded variables. For each variable in the subset where both a CDF and an inverted CDF are available, PIP computes the variable's upper and lower bounds from the atoms in the subset. This includes both direct constraints of the form $X > C$, where X is bounded on the bottom by C, and pairwise constraints of the form $X + Y < C_1$ and $X - Y < C_2$, where X is bounded on the top by $\frac{C_1 + C_2}{2}$.

Applied to all the variables in the subset that have both a CDF and an inverted CDF, this process creates a hyper-rectangular bounding box in the sample space. Because rectangular bounds are independent, the probability of a sample falling within the bounds can be computed independently for each variable as above. Finally, Monte Carlo integration is performed, but only within the subspace. Because sampling is constrained to the bounded area, Monte Carlo integration avoid rejection sampling, and requires fewer samples for a higher precision.

This process is only possible if rectangular bounds on the sample space exist. However, it is most useful when the probability density contained within the atoms is small. Though it is possible to define non-convex constraint atoms, all linear atoms are convex. Furthermore, the space defined by the conjunction of a set of convex atoms is itself convex. Thus, anecdotally it is possible to define a bounding box containing no less than half of the area defined by the constraints.

There are cases where bounds are insufficient. For example, concave atoms are not likely to admit effective rectangular bounds. Similarly, even though a bounding box covers no less than half of the volume of a contiguous convex constraint area, the bulk of the probability mass may still lie outside of the constrained sample area. In such cases, a recursive technique may be applied.

The bounding box is first subdivided into smaller regions. Monte Carlo integration is performed on the region twice, but with only a small number of iterations apiece. If the two results agree to within the desired precision, integration stops and the average of the results is multiplied by the probability of a sample falling into the sampling region. If the two results differ significantly, the region is further subdivided and the algorithm recurses on each sub-region.

Because the recursive cutoff is determined by the estimated accuracy of the result, this algorithm will not recurse on regions that are entirely within or outside of the constrained sample area. Consequently, the majority of samples generated by the algorithm will be put towards estimating relatively high values where Monte Carlo integration is most effective.

3.4 Karp-Luby Algorithm in the Continuous

The number of condition atoms in a given row is linear in the query complexity; every renaming operator adds a fixed number of condition atom columns to the table, while a join simply merges the condition columns of both input tables. Thus, the size of each conjunctive clause is fixed with respect to the amount of input data and typically relatively small. However, there is no such bound on the number of conjunctive clauses in a DNF. Thus, while it is reasonable for PIP to perform conjunctive integration in memory, PIP must use the disk to store intermediate state when computing integrals of DNF formulas.

Two techniques may be used to estimate the general integral of a set of conjunctive clauses of atoms. Under naive Monte Carlo integration, PIP first generates a set of samples and performs a linear scan of the conjunctive clauses to determine how many samples are true in at least one conjunctive clause. The number of samples required is determined both by the scale of the answer and the desired precision. Though straightforward, this approach is inefficient if the value being computed is small.

1. Perform a linear scan of the clauses C of the disjunction, computing $P[\vec{\phi_C}]$. As part of the same scan, compute $bag\_sum = \sum_C P[\vec{\phi_C}]$.

2. Perform another linear scan, this time generating $total\_samples \cdot \frac{P[clause]}{bag\_sum}$ samples $\vec{S}$.

3. Perform a final linear scan over $\vec{C} \bowtie \vec{S}$. Compute the array $Sat[S] = \sum_C \chi_{\vec{\phi_C}}(S)$

4. The average value $\frac{1}{|\vec{S}|} \sum_S \frac{1}{Sat[S]}$ is an estimator for the ratio of the integral to the bag sum.

Figure 3: The Karp-Luby Estimator

An alternative approach is a Karp-Luby style estimator, as used in the discrete case in [18, 13]. This approach first computes the independent probability of each conjunctive clause. The sum of these probabilities, termed the bag sum, is computed and stored. Though the bag sum is related to the integral, the two are not equal unless the conjunctive clauses are all mutually exclusive. If there is overlap between clauses, the bag sum will exceed the value of the integral. The Karp-Luby estimator computes this overlap, generating an estimate of the ratio of the integral to the bag sum.

After computing the bag probabilities and the bag sum, the Karp-Luby estimator generates samples for each clause, constrained to the subspace it defines. Each clause is used to produce a number of samples proportional to its contribution to the bag sum. The generated samples are compared agaainst all clauses. The number of clauses that satisfy a given sample are counted, and the average of the inverse of the counts is the ratio of the integral to the bag sum. This value is multiplied by the bag sum to produce an estimate of the integral. This algorithm is summarized in Figure 3

There is an evident tradeoff between these two processes. naive sampling is faster when constrained sampling cannot be efficiently performed. However, even in this case, the Karp-Luby estimator provides a more consistent precision; it generates a consistent number of "useful" samples, even though doing so is more expensive. These processes are summarized in Figure 4.

4. DESIGN OF THE PIP SYSTEM

The goal of PIP is to evaluate queries on variables sampled from both discrete and continuous distributions as well as to provide tools to aid in the statistical analysis of those results. Uncertainty in PIP is represented via the pVar datatype. Every instance of this datatype represents a random variable. When instantiating a pVar, users specify both a distribution for the variable to be sampled from, and a parameter set for that distribution.

In addition to representing uncertainty in values for individual cells in a table, PIP represents per-tuple uncertainty with c-tables. 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 pVars. The independent probability, or *confidence* of the tuple is the probability of the condition being satisfied.

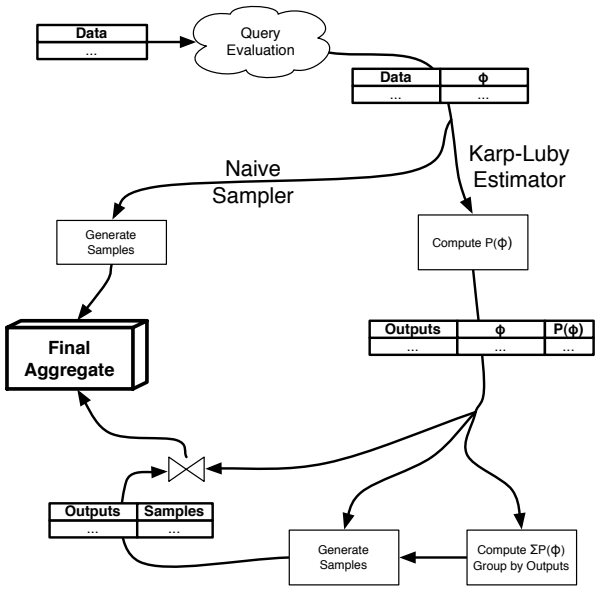

Figure 4: General Integration

4.1 Random Variables

A pVar, or probabilistic variable, represents a nondeterministic value in the database. Every pVar is created from by specifying a distribution and a set of parameters for that distribution. For example, we write $[X => Normal(\mu, \sigma^2)]$ to represent a pVar named X that follows a Normal distribution with a mean of μ and a standard deviation of σ^2. In this way, PIP is agnostic to the distribution which a variable is sampled from; arbitrary problem-specific distributions may be created and seamlessly integrated into PIP's infrastructure.

When defining a distribution, programmers need only include a mechanism for sampling from that distribution. However, if it is possible to efficiently compute or estimate the distribution's probability density function (PDF), cumulative distribution function (CDF), or inverse cumulative distribution function (CDF^{-1}), these may be included to increase PIP's efficiency. The process of defining a variable distribution is described further in Section 5.

Though PIP abstracts the details of the distribution from its query evaluation, it distinguishes between discrete and continuous distributions. As described in Section 2, 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.

Multiple pVars may be combined into an algebraic formula to create a composite pVar. Composite pVars may be used to define joint distributions, to build more complex distributions, or may be created as a side effect of query evaluation. Though composite pVars are interchangeable with regular pVars in general, there are several instances where they receive special treatment. In particular, it is necessary to ensure that within a given sample all instances of a given pVar assume the same value; the value of a composite pVar is obtained only after its component pVars have been sampled.

Finally, constraint atoms are conditional formulas of pVars. Such formulas may be equalities if all variables in the fomula

are discrete, otherwise they must be an inequality; the probability of a continuous variable adopting an exact value is zero. Given a constraint atom C, its probability $P[C]$ is the probability of selecting variable assignments that satisfy the formula. Constraint atoms are used to express a tuple's condition. By adding constraint atom columns to a table, a tuple's presence in the table becomes dependent on the constraint atom being true.

Multiple atoms may be joined by boolean operators to create a constraint formula. PIP expresses constraint formulas as DNF equations. Multiple atoms in a tuple are evaluated conjunctively; the tuple is present if and only if all the atoms are true. Multiple tuples sharing the same key value or values are used to express disjunctions; the key value is present in the output if any one tuple with that key is present.

4.2 Variable Definition

Before evaluating a query on random variables, the variables must first be defined. This process takes one of two forms. PIP uses a repair-key operator similar to that used in [12] to define discrete distributions. Conceptually, this operator identifies tuples that share key values and ensures that only one of the two tuples is present in the database at any given moment.

Mutual exclusion is ensured by adding a constraint atom of the form $Var = Val$, where each key is assigned its own discrete variable, and all tuples sharing a key have distinct values. Repair-key may be parametrized with an initial probability value for each tuple. The sum of these probabilities may not exceed one for a given key. If the sum is less than one, the remainder indicates the probability that none of the tuples are in the table.

Continuous variables may be created inline with the create-variable operation. This function takes a distribution class and parameters, and outputs a new variable. For example, the following query outputs a variable delivery time for a given order.

```
select orders.order_id, orders.item_id,
       create variable ('Normal', params.mean,
         params.std_dev) AS delivery_time
from   orders, params
where  orders.item_id = params.item_id;
```

4.3 Query Evaluation

Query evaluation proceeds in two phases: Query and Sampling. During the query phase, PIP evaluates a query rewritten to employ the c-tables relational algebra extensions described in Section 2. Selection clauses not involving pVars are handled traditionally, while those involving one or more pVars instead tag the output tuple with an equivalent condition clause. For example, the query:

```
select *
from   input_table
where  fixed_column = 3
 and   4 > variable_column;
```

is rewritten to

```
select *, constraint('4 > variable_column')
from   input_table
where  fixed_column = 3;
```

Queries are also modified to ensure that these newly created constraint columns are not projected away.

All selections that generate composite columns including at least one pVar are replaced by composite pVars; evaluation of these columns is effectively delayed until the sampling phase. For example, the query:

```
select fixed_column + variable_column
from   input_table
```

is rewritten to

```
select composite(+, fixed_column, variable_column)
from   input_table
```

The rewritten query is evaluated by a deterministic database engine and produces a *nondeterministic table*; the query's output contains constant values, pVars and condition atoms. Note that discrete variable assignments are expressed entirely through constraint atoms; discrete variables are stored in the database as constants. Because of this, the complexity of evaluating queries over discrete values is pushed entirely into the underlying deterministic database engine. Thus, in the query phase deterministic variables are treated as constants with respect to continuous variables.

It is also important to note that the nondeterministic table is a **lossless** encoding of the relationship between the variables used in the query and the output table. There is no bias to be perpetuated in further queries on the output table. Consequently, the output of one query phase may be saved as a materialized view and used for arbitrary further computation.

5. 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. PIP's extended functionality is provided by a set of user-defined functions written in C.

5.1 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 variables. Each call allocates a new variable and initializes it with the specified parameters. When specifying selection targets, operator overloading is used to make random variables appear as normal variables; arbitrary equations may be constructed in this way.

However, additional work is needed to complete the illusion. In addition to the PIP plugin, 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 two 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. For example, if (X¿Y) resolves to a CTYPE variable,

  ```
  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.

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).

5.2 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.

1. $Generate(Parameters, Seed)$ 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.

2. $PDF(Parameters, x)$ evaluates the probability density function of the distribution at the specified point.

3. $CDF(Parameters, x)$ evaluates the cumulative distribution function at the specified point.

4. $InverseCDF(Parameters, Value)$ 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 specified. Consequently, the supplemental functions need only be provided when it is possible to evaluate them efficiently. Depending on the user's needs however, it may be reasonable to provide estimates instead of exact values. For example, the CDF of a Normal distribution is a complex integral, but may be efficiently estimated by interpolating between precomputed values.

Future implementations could conceivably generalize the sampling process. A sample may be generated using any of the four functions: The Metropolis-Hastings algorithm can sample from an arbitrary PDF, the inverse CDF evaluated on a uniform random value produces a sample, and a binary search may be used to evaluate the inverse CDF given the CDF.

5.3 Sampling Functionality

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

conf() performs a conjunctive integration to estimate the probability of a specific row being present in the output. 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. Two variants of aconf() exist: a naive one-pass version, and a Karp-Luby variant that builds a transient table on which it performs two more aggregation passes.

expected() computes the expectation of a variable by repeated sampling. If a row may be specified when the function is called, the sampling process is constrained by the constraint atoms present in the row.

stddev() computes the standard deviation of the expectation by repeated sampling. As with expected() a row may be used to constrain the range of the variable.

histogram() is similar to expected() in that it performs repeated sampling, or constrained sampling if appropriate. However, instead of outputting the average of the results, it instead outputs an array of all the requested samples.

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 composite aggregate variable could easily grow to an unmanageable level. Instead, PIP limits random variable aggregation to the sampling phase.

Many aggregates such as sum() and average() are linear; the expectation of the sum is the sum of the expectation. PIP implements these by combining Postgres' existing aggregate operator with PIP's expectation operator. PIP also implements histogram variants of both operators. Nonlinear aggregates such as max(), min(), and stddev() are special-cased.

6. 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 loosely 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.

We evaluated both the Pip C-Tables and the Sample-First infrastructure against three 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. All queries were evaluated over a 1 GB database generated by the TPC-H benchmark. Unless otherwise specified, all sampling processes generate 1000 samples apiece. Unless otherwise specified, results shown are the average of 10 measurements run sequentially.

The first set of tests evaluate Pip's performance on queries similar to those used in [9]. The results of these tests are shown in Figure 5. Performance times for Pip are divided into two components: query and sample, to distinguish be-

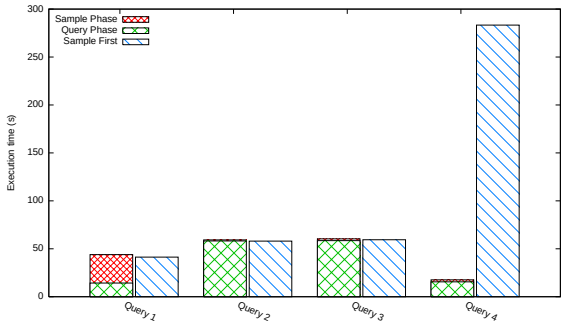

Figure 5: Query evaluation times in Pip and Sample-First

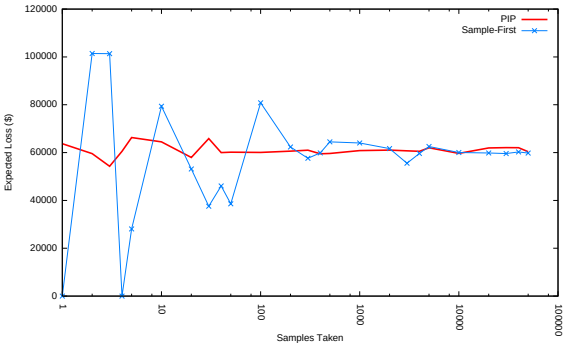

Figure 6: Variance as a function of samples in Pip and Sample-First. Each data point is an estimate generated by a single run at the indicated number of samples

tween 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 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 and third queries, 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. The second query computes the maximum of these dates, while the third compares the times against arbitrary customer satisfaction thresholds to generate a list of potential "dissatisfied" customers.

As expected, Pip's performance on these queries is comparable to the sample-first approach. In the general case, Pip performs effectively the same computations as Sample-First, save that Pip delays sampling slightly longer. In particular, queries 2 and 3 demonstrate how little time sampling can take for some queries; additional samples can be computed without incurring the nearly 1 minute query time.

The final timing test combines a simplified version of the prior two queries to simulate a risk management query that might be evaluated daily. Given per-customer revenue forecasts, estimates for product manufacturing and shipping times, customer satisfaction thresholds, compute the expected revenue lost to customers dissatisfied with today's service. For added realism, this query uses a precomputed table of part production and shipping times (which changes rarely enough that it can be updated monthly). This query is shown in Figure 7.

This query includes two distinct, independent sampling components: the expectation of revenue from a customer, and the probability that a customer will be dissatisfied. 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. Because worlds where a customer is satisfied are not relevant to the query, an arbitrarily large number of customer revenue values will be discarded. Pip separates the computation of expectations and confidences and does not suffer from this problem.

For this comparison, 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.

Figure 6 demonstrates one extreme case of this in a comparison between Karp-Luby estimates and Sample-First estimates. The results shown are for repeated executions of a query similar to query 4, save with a filter that removes all but approximately 10 clients. In queries that do not involve a large linear aggregate, the sample-first approach disqualifies a sufficient number of possible worlds that subsequent expectation computations falter. Conversely the Karp-Luby estimator has sufficient information that it can employ a precomputed CDF lookup table to compute each row's bag probabilities. Because of this and the fact that it generates more "useful" samples, its results have a much lower variance with far fewer samples required.

We have shown that it is possible to apply the c-tables approach to probabilistic databases with only minimal overhead, even when it is used to represent arbitrary variable distributions. The additional information provided by the c-table adds a degree of flexibility that allows Pip to outperform Sample-First approaches in a number of instances.

7. 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] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross. "An Optimal Algorithm for Monte Carlo Estimation". *SIAM J. Comput.*, **29**(5):1484–1496, 2000.

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

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

```
create table ‘shipping_params’ as
  select
    avg   (l_shipdate    - o_orderdate) as ship_mu,
    avg   (l_receiptdate - l_shipdate ) as arrv_mu,
    stddev(l_shipdate    - o_orderdate) as ship_sigma,
    stddev(l_receiptdate - l_shipdate ) as arrv_sigma,
    l_partkey as p_partkey
  from orders,lineitem
  where o_orderkey = l_orderkey
  group by partkey;
alter table params add constraint "p_partkey_pkey"
  primary key (p_partkey);
-- BEGIN TIMING QUERY --
create temporary table q4_shipping as
  select o_orderkey AS orderkey, o_custkey AS custkey,
    CREATE VARIABLE(‘Normal’,ship_mu,ship_sigma)
    CREATE VARIABLE(‘Normal’,arrv_mu,arrv_sigma)
  from  orders,lineitem,shipping_params
  where p_partkey = l_partkey;
   and  o_orderdate = today()
   and  o_orderkey  = l_orderkey;
create temporary table q4_annoyed as
  select custkey from q4_shipping where ship > 120
union all
  select custkey from q4_shipping where arrv > 90;
create temporary table q4_order_increase as
  select o_orderkey, o_custkey,
     CREATE VARIABLE(‘Poisson’, increase) *
       l_extended_price * (1.0 - l_discount) as rev
  from (select newc / oldc as increase, custkey
    from (select o_custkey as custkey,
             sum(o_orderdate.year-1996.0) AS newc,
             sum(1997.0-o_orderdate.year) AS oldc
      where o_orderdate.year = 1997
        or  o_orderdate.year = 1996
      group by custkey
     ) as counts
   ) as increase_per_cust,
    orders
  where custkey = o_custkey
 ) as var_increase_per_customer,
 (select lineitem.*,
  from  nation,supplier, lineitem, partsupp
  where n_name = ’japan’ and n_nationkey = s_nationkey
   and  s_suppkey = ps_suppkey
   and  ps_partkey = l_partkey
   and  ps_suppkey = l_suppkey
 ) as items_from_japan;
-- BEGIN TIMING SAMPLE --
select avg(confidence),
       expected_sum_naive(rev, q4_annoyed)
from   q4_annoyed,
       (select o_custkey as custkey, rev
        from   q4_revenue_gains
       ) as revenues
where  revenues.custkey = q4_annoyed.custkey;
```

Figure 7: Timing Query 4

[6] E. Grädel, Y. Gurevich, and C. Hirsch. “The Complexity of Query Reliability”. In *Proc. PODS*, pages 227–234, 1998.

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

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

[9] 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.

[10] R. M. Karp and M. Luby. “Monte-Carlo Algorithms for Enumeration and Reliability Problems”. In *Proc. FOCS*, pages 56–64, 1983.

[11] R. M. Karp, M. Luby, and N. Madras. “Monte-Carlo Approximation Algorithms for Enumeration Problems”. *J. Algorithms*, **10**(3):429–448, 1989.

[12] 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, 2008. To appear.

[13] C. Koch and D. Olteanu. “Conditioning Probabilistic Databases”. In *Proc. VLDB*, 2008.

[14] 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.

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

[16] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. “BLOG: Probabilistic Models with Unknown Objects”. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, Cambridge, MA, 2007.

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2007.

[18] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. ICDE*, pages 886–895, 2007.

[19] M. Richardson and P. Domingos. “Markov Logic Networks”. *Machine Learning*, **62**:107–136, 2006.

[20] 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.

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

[22] 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.

[23] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[24] 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.

[25] J. Widom. “Trio: a system for data, uncertainty, and lineage”. In C. Aggarwal, editor, *Managing and Mining Uncertain Data*. Springer-Verlag, 2008. To appear.