# Convergent Interactive Inference with Leaky Joins

Ying Yang
University at Buffalo
yyang25@buffalo.edu

Oliver Kennedy
University at Buffalo
okennedy@buffalo.edu

## ABSTRACT

One of the primary challenges in graphical models is inference, or re-constructing a marginal probability from the graphical model's factorized representation. While tractable for some graphs, the cost of inference grows exponentially with the graphical model's complexity, necessitating approximation for more complex graphs. For interactive applications, latency is the dominant concern, making approximate inference the only feasible option. Unfortunately, approximate inference can be wasteful for interactive applications, as exact inference can still converge faster, even for moderately complex inference problems. In this paper, we propose a new family of convergent inference algorithms (CIAs) that bridge the gap between approximations and exact solutions, providing early, incrementally improving approximations that become exact after a finite period of time. We describe two specific CIAs based on a cryptographic technique called linear congruential generators, including a novel incremental join algorithm for dense relations called Leaky Joins. We conclude with experiments that demonstrate the utility of Leaky Joins for convergent inference: On both synthetic and real-world probabilistic graphical models, Leaky Joins converge to exact marginal probabilities almost as fast as state of the art *exact* inference algorithms, while simultaneously achieving approximations that are almost as good as state of the art *approximation* algorithms.

## 1. INTRODUCTION

Probabilistic graphical models (PGMs) are a factorized encoding of joint (multivariate) probability distributions. Even large distributions can often be compactly represented as a PGM. A common operation on PGMs is inference, or reconstructing the marginal probability for a subset of the variables in the full joint distribution. Existing inference algorithms are either exact or approximate. Exact algorithms [9] like variable elimination and belief propagation produce exact results, but can be slow. On the other hand, approximate algorithms [44,45] like Gibbs sampling generate estimates within any fixed time bounds, but only converge asymptotically to exact results.

Over the past decade, a class of model database systems have begun to add support for probabilistic graphical models (PGMs) within database engines, allowing graphical models to be queried through SQL [13, 36, 42], combined with other data for joint analysis [20, 22], or used for analytics over messy data [29, 40, 41].

Model database systems typically employ approximate inference techniques, as model complexity can vary widely with different usage patterns and responsiveness is typically more important than exact results. However, exact inference can sometimes produce an exact result *faster* than it takes an approximate algorithm to converge, even for moderately complex inference problems. Furthermore, in interactive settings, the user may be willing to wait for more accurate results. In either case, the choice of whether or not use an exact algorithm must wait until the system has already obtained an approximation.

In this paper, we explore a family of *convergent inference algorithms* (CIAs) that simultaneously act as both approximate and exact inference algorithms: Given a fixed time bound, a CIA can produce a bounded approximate inference result, but will also terminate early if it is possible to converge to an exact result. Like a file copy progress bar, CIAs can provide a "result accuracy progress bar" that is guaranteed to complete eventually. Similar to online-aggregation [18] (OLA), CIAs give users and client applications more control over accuracy/time trade-offs and do not require an upfront commitment to either approximate or exact inference.

We propose two specific CIAs that use the relationship between inference and select-join-aggregate queries to build on database techniques for OLA [18]. Our algorithms specialize OLA to two unique requirements of graphical inference: dense data and wide joins. In classical group-by aggregate queries, the joint domain of the group-by attributes is sparse: Tables in a typical database only have a small portion of their active domain populated. Furthermore, classical database engines are optimized for queries involving a small number of large input tables. Conversely, in graphical inference, each "table" is small and dense and there are usually a large number of tables with a much more complicated join graph.

The density of the input tables (and by extension, all intermediate relations) makes it practical to sample directly from the output of a join, since each sample from the active domain of the output relation is likely to hit a row that is

present and non-zero. Hence, our first, naive CIA samples directly from the output of the select-join component of the inference query, using the resulting samples to predict the aggregate query result. To ensure convergence, we leverage a class of pseudorandom number generators called Linear Congruential Generators [31, 34] (LCGs). The cyclicity of LCGs has been previously used for coordinating distributed simulations [6]. Here, we use them to perform random sampling from join outputs *without* replacement, allowing us to efficiently iterate through the join outputs in a shuffled order. These samples produce bounded-error estimates of aggregate values. After the LCG completes one full cycle, every row of the join has been sampled exactly once and the result is exact.

Unfortunately, the domain of the join output for an inference query can be quite large and this naive approach converges slowly. To improve convergence rates, we propose a new online join algorithm called *Leaky Joins* that produces samples of a query's result in the course of normally evaluating the query. Systems for relational OLA (e.g., [15, 18, 21]) frequently assume that memory is the bottleneck. Instead, Leaky Joins are optimized for small input tables that make inference more frequently compute-bound than IO-bound. Furthermore, the density of the input (and intermediate) tables makes it possible to use predictable, deterministic addressing schemes. As a result, Leaky Joins can obtain unbiased samples efficiently without needing to assume a lack of correlation between attributes in the input.

The Leaky Joins algorithm starts with a classical bushy query plan. Joins are evaluated in parallel, essentially *"leaking"* estimates for intermediate aggregated values — marginal probabilities in our motivating use case — from one intermediate table to the next. One full cycle through a LCG is guaranteed to produce an exact result for joins with exact inputs available. Thus, initially only the intermediate tables closest to the leaves can produce exact results. As sampling on these tables completes a full cycle, they are marked as stable, sampling on them stops, and the tier above them is permitted to converge. In addition to guaranteeing convergence of the final result, we are also able to provide confidence bounds on the approximate results prior to convergence. As we show in our experiments, the algorithm satisfies desiderata for a useful convergent-inference algorithm: *computation performance* competitive with exact inference on simple graphs, and *progressive accuracy* competitive with approximate inference on complex graphs.

Our main motivation is to generate a new type of inference algorithm for graphical inference in databases. Nevertheless, we observe that Leaky Joins can be adapted to any aggregate queries over small but dense tables.

Specifically, our contributions include: (1) We propose a new family of Convergent Inference Algorithms (CIAs) that provide approximate results over the course of inference, but eventually converge to an exact inference result, (2) We cast the problem of Convergent Inference as a specialization of Online Aggregation, and propose a naive, *constant-space* convergent inference algorithm based on Linear Congruential Generators, (3) We propose Leaky Joins, a novel Online Aggregation algorithm specifically designed for Convergent Inference, (4) We show that Leaky Joins have time complexity that is no more than one polynomial order worse than classic exact inference algorithms, and provide an $\epsilon-\delta$ bound to demonstrate that the approximation accuracy is competitive with common approximation techniques, (5) We present experimental results on both synthetic and real-world graph data to demonstrate that (a) Leaky Joins gracefully degrade from exact inference to approximate inference as graph complexity rises. (b) Leaky Joins have exact inference costs competitive with classic exact inference algorithms, and approximation performance competitive with common sampling techniques, (6) We discuss lessons learned in our attempts to design a convergent inference algorithm using state-of-the-art incremental view maintenance systems [4, 25].

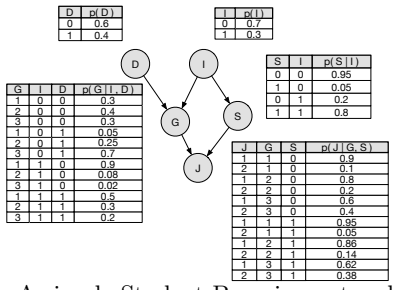


Figure 1: A simple Student Bayesian network

## 2. BACKGROUND AND RELATED WORK

In this section, we introduce notational conventions that we use throughout the paper and briefly overview probabilistic graphical models, inference and on-line aggregation.

### 2.1 Bayesian Networks

Complex systems can often be characterized by multiple interrelated properties. For example, in a medical diagnostics system, a patient might have properties including symptoms, diagnostic test results, and personal habits or predispositions for some diseases. These properties can be expressed for each patient as a set of interrelated random variables. We write sets of random variables in bold (e.g., $\mathbf{X} = \{X_i\}$). Denote by $p(\mathbf{X})$ the probability distribution of $X_i \in \mathbf{X}$ and by $p(x)$ the probability measure of the event $\{X_i = x\}$. Let $\mathbf{X} \backslash \mathbf{Y}$ denote the set of variables that belong to $\mathbf{X}$ but do not belong to $\mathbf{Y}$.

A Bayesian network (BN)[1] represents a joint probability distribution over a set of variables $\mathbf{X}$ as a directed acyclic graph. Each node of the graph represents a random variable $X_i$ in $\mathbf{X}$. The parents of $X_i$ are denoted by $pa(X_i)$, the children of $X_i$ are denoted by $ch(X_i)$.

A Bayesian network compactly encodes a joint probability distribution using the Markov condition: Given a variable's parents, the variable is independent of all of its non-descendants in the graph. Thus, the full joint distribution is given as:

$$P(\mathbf{X}) = \prod_i P(X_i | pa(X_i))$$

Every random variable $X_i$ is associated with a conditional probability distribution $P(X_i|pa(X_i))$. The joint probability distribution is factorized into a set of $P(X_i|pa(X_i))$ called factors denoted by $\phi_i$ or factor tables if $X_i$ is discrete. Denote by $scope(\phi_i)$ the variables in a factor $\phi_i$. Finally, we use $attrs(\phi_i) = scope(\phi_i) \cup \{p_{\phi_i}\}$ to denote the attributes

[1]Although our focus here is inference on directed graphical models (i.e. Bayesian networks), the same techniques can be easily adapted for inference in undirected graphical models.

of the corresponding factor table: the variables in the factor's scope and the probability of a given assignment to $X_i$ given fixed assignments for its parents. A full BN can then be expressed as the 2-tuple $\mathcal{B} = (\mathcal{G}(\mathbf{X}), \Phi)$, consisting of the graph and the set of all factors.

EXAMPLE 1. *Consider four random variables **I**ntelligence, **D**ifficulty, **G**rade, **S**AT, and **J**ob in a Student Bayesian network. The four variables I, D, S, J have two possible values, while G has 3. A relation with $2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 48$ rows is needed to represent this joint probability distribution. Through the Markov condition, the graph can be factorized into the smaller Bayesian network given in Figure 1. For a graph with a large number of variables with large domains, factorization can reduce the size significantly.*

## 2.2 Inference

Inference in BNs usually involves computing the posterior marginal for a set of query variables $\mathbf{X}_q$ given a set of evidence, denoted by $E$. For example, $E = \{X_1 = x_1, X_3 = x_3\}$ fixes the values of variables $X_1$ and $X_3$. Denote by $\mathbf{X}_E$ the set of observed variables (e.g., $\mathbf{X}_E = \{X_1, X_3\}$). The posterior probability of $\mathbf{X}_q$ given E is

$$P(\mathbf{X}_q|E) = \frac{P(\mathbf{X}_q, E)}{P(E)} = \frac{\sum_{\mathbf{X}\setminus\{\mathbf{X}_q,\mathbf{X}_E\}} P(\mathbf{X})}{\sum_{\mathbf{X}\setminus\mathbf{X}_E} P(\mathbf{X})}.$$

The marginalization of $X_1 \ldots X_i$ over a joint probability distribution is equivalent to a select-join-aggregate query computed over the ancestors of $X_1 \ldots X_i$:

```
SELECT X_1,...,X_i,
           SUM(p_1 * ... * p_N) AS prob
FROM factor_1 NATURAL JOIN ...
                  NATURAL JOIN factor_N
WHERE E_1 = e_1 AND ... AND E_k = e_k
GROUP BY X_1,...,X_i;
```

Applying evidence to a graphical model is computationally straightforward and produces a strictly simpler graphical model. As a result, without loss of generality, we ignore evidence and focus exclusively on straightforward inference queries of the form $\mathrm{P}(\mathbf{X}_q)$.

### 2.2.1 Exact Inference

**Variable Elimination.** Variable elimination mirrors aggregation push-down [10], a common query optimization technique. The idea is to avoid the exponential blowup in the size of intermediate, joint distribution tables by pushing aggregation down through joins over individual factor tables. As in query optimization, join ordering plays a crucial role in variable elimination, as inference queries often have join graphs with high hypertree width. Intermediate materialized aggregates in VE are typically called separators (denoted $S$), intermediate (materialized) joins are called cliques ($C$), variables aggregated away between a clique and the following separator are called clique variables (denoted $var(C)$), and their inputs are called clique factors.

EXAMPLE 2. *The marginal probability distribution of J in Figure 1 can be expressed by $p(J) = \sum_{D,I,S,G} p(D, I, S, G, J)$. We choose to first marginalize out D by constructing $C_D$'s separator $S_D$:*

$$S_D[G, I] = \sum_D C_D[D, G, I] = \sum_D \phi_D[D] \bowtie_D \phi_G[D, G, I]$$

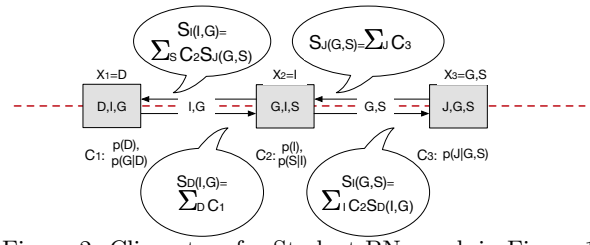


Figure 2: Clique tree for Student BN graph in Figure 1

*Next, we marginalize out I by computing*

$$S_I[G, S] = \sum_I C_I[G, I, S] = \sum_I \boldsymbol{S}_D[G, I] \bowtie_I \phi_I[I] \bowtie \phi_S[I, S]$$

*The marginalization of G and S follows a similar pattern, leaving us with $C_q = S_S[J] = p(J)$.*

The limiting factor in the computational cost of obtaining a separator is enumerating the rows of the clique. Assuming that the distribution over each variable $X_i$ has N possible outcomes ($|dom(X_i)| = N$), the cost of computing separator $S$ with clique $C$ will be $O(N^{|scope(C)|})$. Tree-width in graphical models (related to query hypertree width) is the size of the largest clique's scope ($\max_C(|scope(C)|)$), making variable elimination exponential-cost in the graph's tree-width.

**Belief Propagation.** Belief propagation generalizes variable elimination by allowing information to flow in both directions along the graph, messages are sent along each cluster's separator by summing out all uncommon variables between the two clusters. The process creates, for each variable in the graph, its full conditional probability given all other variables in the graph. Figure 2 shows the message passing process for the graph in Figure 1. Although belief propagation is more efficient for performing multiple simultaneous inference operations in parallel, for singleton tasks it is a factor of two slower than variable elimination. Thus, in this paper we use variable elimination as a representative example of exact inference.

### 2.2.2 Approximate Inference

**Markov Chain Monte Carlo Inference.** MCMC is a family of sampling techniques that generate sequences of samples. Intuitively, the first element in the sequence is drawn from the prior and successive samples are drawn from distributions that get increasingly closer to the posterior. For example, we might draw one assignment of values in $\mathbf{X}$ with each variable $X_i$ following the conditional probability distribution $p(X_i|pa(X_i))$ in the topological order of the graph. Then, we iteratively re-sample one variable's value at a time according to its factor table, given the current assignments for its parents and children. The longer we continue re-sampling, the less the sample is biased by its initial value. We use **Gibbs sampling** as a representative MCMC inference algorithm.

**Loopy Belief Propagation.** Loopy belief propagation is the same as belief propagation, but operates on a loopy cluster graph instead of a clique tree. This change makes the cluster smaller than those in clique tree and makes message passing steps less expensive. There is a trade-off between cost and accuracy in loopy-belief propagation, as join graphs that allow fast propagation may create a poor approximation of the result. This tradeoff is antithetical to our goal

of minimizing heuristic tradeoffs, making loopy-belief propagation a poor fit for our target applications. As a result we focus on Gibbs sampling as a representative example of approximate inference.

## 2.3 Online Aggregation

Starting with work by Hellerstein et.al. [18], Olken [30], and others, a large body of literature has been developed for so-called online aggregation (OLA) or approximate query processing (AQP) systems. Such systems replace pipeline-blocking operators like join and aggregate with sampling-based operators that permit approximate or partial results to be produced immediately, and iteratively refined the longer the user is willing to wait. The work most closely related to our own efforts is on OLA [16, 17, 18]. OLA systems use query evaluation strategies that estimate and iteratively refines the output of aggregate-joins. Given enough time, in most systems, the evaluation strategy eventually converges to a correct result. As in random sampling, $\epsilon - \delta$ bounds can be obtained, for example using Hoeffding's inequality. A key challenge arising in OLA is how to efficiently generate samples of source data. Sampling without replacement allows the algorithm to converge to the correct result once all samples have been exhausted, but has high space requirements, as it is necessary to keep track of the sampling order. Conversely, sampling with replacement is not guaranteed to ever converge to the correct answer. One of our key contributions in this paper is a specialization of OLA to graphical models called Cyclic Sampling, which permits sampling without replacement using only constant-space. Numerous other systems have since adapted and improved on the idea of OLA. Aqua [1] uses key constraints for improved stratified sampling. BlinkDB [2, 3] and Derby [23] maintain prematerialized stratified samples to rapidly answer approximate queries. GLADE [33] and DBO [15, 21] exploit file buffering to opportunistically generate samples of a query result in the course of normal query evaluation.

# 3. CONVERGENT INFERENCE

Running-time for variable elimination $O(N^{|scope(C)|})$, is dominated by tree-width, and strongly depends on the elimination ordering (already an $NP$-hard problem [26]). Since the running time grows exponentially in the size of largest clique cluster $C_{max}$, the running complexity can have high variance depending on the order. Because the cost is exponential, even a small increase in complexity can change the runtime of variable elimination from seconds to hours, or even days. In short, predicting whether an exact solution is feasible is hard enough that most systems simply rely exclusively on approximation algorithms. On other hand, approximate inference may get asymptotically close to an answer, but it will never fully converge. Thus, most applications that benefit from exact results must rely on either human intuition to decide.

The goal and first contribution of this paper is to introduce *convergent inference*, a specialized form of approximate inference algorithm that is guaranteed to eventually converge to an exact result. In this section, we develop the idea of convergent inference and propose several convergent inference algorithms, or CIAs. A CIA eventually produces an exact result, but can be interrupted at any time to quickly produce a bounded approximation. More precisely, a CIA should satisfy the following conditions: (1) After a fixed period of time $t$, a CIA can provide approximate results with $\epsilon - \delta$ error bounds, such that $P\left(|P_t - P_{exact}| < \epsilon\right) > 1 - \delta$; and (2) A CIA can will obtain the exact result $P_{exact}$ in a bounded time $t_{exact}$.

Ideally, we would also like a CIA to satisfy two additional conditions: (3) The time complexity required by a good CIA to obtain an exact result should be competitive with variable elimination; and (4) The quality of the approximation produced by a good CIA should be competitive with the approximation produced by a strictly approximate inference algorithm given the same amount of time.

We first introduce a fundamental algorithm called cyclic sampling which performs pseudo-random sampling without replacement from the joint probability distribution of the graph. This algorithm is guaranteed to converge, but requires an exponential number of samples to do so. We then present an improved CIA based on classical aggregate-join query processing that relies on a novel "leaky join" operator. Finally, we discuss lessons learned in a failed attempt to combine cyclic sampling with state-of-the-art techniques for incremental view maintenance.

All three of our approaches draw on the relationship between graphical inference and aggregate query processing. However, though the problems are similar, we re-emphasize that there are several ways in which graphical inference queries violate assumptions made in classical database query-processing settings. First, conditional probability distributions are frequently dense, resulting in many-many relationships on join attributes. Correlations between variables are also common, so graphical models often have high tree widths. By comparison, the common case for join and aggregation queries is join graphs with a far smaller number of tables, simpler (e.g., foreign key) predicates, and typically low tree widths.

Finally, we note that although we use graphical models as a driving application, similar violations occur in other database applications (e.g., scientific databases [39]). The algorithms we present could be adapted for use in these settings as well.

## 3.1 Cyclic Sampling

We first discuss a naive form of convergent inference called cyclic sampling that forms the basis for each of our approaches. Recall that each intermediate table (the separator) is an aggregate computed over a join of factor tables (the clique), and that the domain of the clique is (or is very nearly) a cartesian product of the attributes in its scope. In principle, one could compute an entire separator table by scanning over the rows of its clique.

We note that input factor tables and the separator tables are typically small enough to remain in memory. Thus, using array-indexed storage for the clique tables is feasible and the cost of accessing one row of the clique is a constant. Consequently, efficient random sampling on the joint probability distribution is possible, and the marginal probabilities of interest can be incrementally approximated as in OLA [18].

The key insight of cyclic sampling is that if this random sampling is performed without replacement, it will eventually converge to an exact result if we reach a point where each row of the clique has been sampled exactly once. Unfortunately, sampling without replacement typically has space complexity linear in the number of items to be sampled,

which is exponential in the number of variables. Fortunately for graphical inference, a there exists a class of so-called cyclic pseudorandom number generators that iteratively construct pseudorandom sequences of non-repeating integers in constant space.

A cyclic pseudorandom number generator generates a sequence of non-repeating numbers in the range $[0, m)$ for some given period $m$ with members that exhibit minimal pairwise correlation. We use Linear Congruential Generators (LCGs) [31, 34], which generate a sequence of semi-random numbers with a discontinuous piecewise linear equation defined by the recurrence relation:

$$X_n = (aX_{n-1} + b) \mod m \quad (1)$$

Here $X_n$ is the $n$th number of the sequence, and $X_{n-1}$ is the previous number of the sequence. The variables $a$, $b$ and $m$ are constants: $a$ is called the multiplier, $b$ the increment, and $m$ the modulus. The key, or seed, is the value of $X_0$, selected uniformly at random between 0 and $m$. In general, a LCG has a period no greater than $m$. However, if $a$, $b$, and $m$ are properly chosen, then the generator will have a period of exactly $m$. This is referred to as a maximal period generator, and there have been several approaches to choosing constants for maximal period generators [24,28]. In our system, we follow the Hull-Dobell Theorem [38], which states that an LCG will be maximal if

1. $m$ and the offset $b$ are relatively prime.
2. $a - 1$ is divisible by all prime factors of $m$.
3. $a - 1$ is divisible by 4 if m is divisible by 4.

LCGs are fast and require only constant memory. With a proper choice of parameters $a$, $b$ and $m$, a LCG can produce maximal period generators and pass formal tests for randomness. Parameter selection is outlined in Algorithm 1.

**Algorithm 1** `InitLCG(totalSamples)`

**Require:** *totalSamples*: the total number of samples
**Ensure:** $a$, $b$, $m$: LCG Parameters
1: $m \leftarrow totalSamples$
2: $S \leftarrow$ prime factors of $m$
3: **for each** $s$ **in** $S$ **do**
4: $\quad a \leftarrow a \times s$
5: **if** $m = 0 \mod 4$ **and** $a \neq 0 \mod 4$ **then**
6: $\quad a \leftarrow 4a + 1$
7: $b \leftarrow$ any coprime of $m$ smaller than $m$

The cyclic sampling process itself is shown in Algorithm 2. Given a Bayesian network $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,$\Phi$), and a marginal probability query Q = $p(\mathbf{X}_q)$, we first construct the LCG sampling parameters (lines 1-5): $a$, $b$ and $m$ according to Hull-Dobell Theorem for the total number of samples in the joint probability distribution p($\mathbf{X}$). The sampling process (starts at Line 16) constructs an index by obtaining the next value from the LCG (line 7) and decomposes the index into an assignment for each variable (line 10). We calculate the joint probability of p($\mathbf{X}$) for this assignment (line 13) add this probability to the corresponding result row, and increment the sample count.

At any point, the algorithm may be interrupted and each individual probability in $p(\mathbf{X}_q)$ may be estimated from the accumulated probability mass $p(x)$:

$$p(\mathbf{X}_q) = \frac{\prod_{X_j \in \mathbf{X}} |dom(X_j)|}{count_x} \cdot p(x)$$

Cyclic sampling promises to be a good foundation for CIA, but must satisfy the two constraints. First, it needs to provide an epsilon-delta approximation in a fixed period of time. In classical OLA and some approximate inference algorithms, samples generated are independently and identically distributed. As a result, Hoeffding's inequality can provide accuracy guarantees. In CIAs, samples are generated without replacement. We need to provide an $\epsilon - \delta$ approximation under this assumption. Second, cyclic sampling needs to eventually converge to an exact inference result. This requires that we sample all the items in the joint probability distribution exactly once and the samples should be sampled randomly.

**Algorithm 2** `CyclicSampling`($\mathcal{B}$, $Q$)

**Require:** A bayes net $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,$\Phi$)
**Require:** A conditional probability query: Q=$P(\mathbf{X_q})$
**Ensure:** Probabilities for each $\vec{q}$: $\{p_{\vec{q}} = P(\mathbf{X_q}) = \vec{x}_q)\}$
**Ensure:** The number of samples for each $\vec{q}$: $\{count_{\vec{q}}\}$
1: $totalSamples \leftarrow 1$
2: **for each** $\phi_i$ **in** $\Phi$ **do**
3: $\quad totalSamples = totalSamples * |dom(X_i)|$
4: $index_1 \leftarrow$ `rand_int()` $\mod totalSamples$
5: $a, b, m \leftarrow$ `InitLCG`($totalSamples$)
6: **for each** $k \in 0 \ldots totalSamples$ **do**
7: $\quad assignment \leftarrow index_k$
8: $\quad$ `/* De-multiplex the variable assignment */`
9: $\quad$ **for each** $j \in 0 \ldots n$ **do**
10: $\quad\quad x_j \leftarrow assignment \mod |dom(X_j)|$
11: $\quad\quad assignment \leftarrow assignment \div |dom(X_j)|$
12: $\quad$ `/* probability is a product of the factors */`
13: $\quad prob \leftarrow \prod_i \phi_i \left(\pi_{scope(\phi_i)}(\langle x_1, \ldots, x_n \rangle)\right)$
14: $\quad$ `/* assemble return values */`
15: $\quad \vec{q} \leftarrow \pi_{\mathbf{X_q}}(\vec{x})$; $p_{\vec{q}} \leftarrow p_{\vec{q}} + prob$; $count_{\vec{q}} \leftarrow count_{\vec{q}} + 1$
16: $\quad$ `/* step the LCG */`
17: $\quad index_{k+1} \leftarrow (a * index_k + b) \mod m$

**Computation Cost.** Let $n$ be the number of random variables in $\mathcal{B}$, as a simplification assume w.l.o.g. that each random variable has domain size *dom*. Calculating the parameters for LCG takes constant time. The sampling process takes $O(|N|)$ time, where $|N|$ is the total number of samples in the reduced joint probability distribution $P(\mathbf{X})$. $N$ can be as large as $dom^n$, that is exponential in the size of the graph.

**Confidence Bound.** Classical approximate inference and OLA algorithms use random sampling with replacement, making it possible to use well known accuracy bounds. For example one such bound, based on Hoeffding's inequality [19] establishes a tradeoff between the number of samples needed $n$, a probabilistic upper bound on the absolute error $\epsilon$, and an upper bound on probably that the bound will be violated $\delta$. Given two values, we can obtain the third.

Hoeffding's inequality for processes that sample with replacement was extended by Serfling et al. [37] for sampling without replacement. Denote by N the total number of samples, P(x) is the true probability distribution after seeing all the samples N. Denote by $P_n(x)$ the approximation of P(x)

after n samples. From [37], and given that probabilities are in general bounded as $0 \leq p \leq 1$, we have that:

$$P_n\left(P_n(x) \notin [P(x)-\epsilon, P(x)+\epsilon]\right) \leq \delta \equiv exp\left[\frac{-2n\epsilon^2}{1-(\frac{n-1}{N-1})}\right] \quad (2)$$

In other words, after $n$ samples, there is a $(1-\delta)$ chance that our estimate $P_n(x)$ is within an error bound $\epsilon$.

## 3.2 Leaky Joins

In Cyclic Sampling, samples are drawn from an extremely large joint relation and require exponential time for convergence. To address this limitation, we first return to Variable Elimination as described in Section 2.2.1. Recall the clique tree representation in Figure 2 for the BN in Figure 1, where the marginal for the goal variable ($J$) is produced by clique cluster $C_3$. Each clique focuses on a single clique variable $X_i$, and the clique cluster is a product of the separator table to the clique's left and all remaining factor tables containing $X_i$. As a result, each factor $\phi$ in $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,$\Phi$) belongs to exactly one clique cluster. Variable Elimination (the process below the red line) mirrors classical blocking aggregate-join evaluation, computing each separator table (aggregate) fully and passing it to the right.

The key idea is to create a clique tree as in Variable Elimination, but to allow samples to gradually "leak" through the clique tree rather than computing each separator table as a blocking operation. To accomplish this, we propose a new *Leaky Join* relational operator. A single Leaky Join computes a group-by aggregate over one or more Natural Joins, "online" using cyclic sampling as described above. As the operator is given more cpu-time, its estimate improves. Crucially, Leaky Joins are composable. During evaluation, all Leaky Joins in a query plan are updated in parallel. Thus the quality of a Leaky Join operator's estimate is based not only on how many samples it has produced, but also on the quality of the estimates of its input tables.

Algorithm 3 gives an evaluation strategy for inference queries using Leaky Joins. Abstractly, an evaluation plan consists of a set of intermediate tables for each clique $C_i \in \mathbf{C}$ (i.e., each intermediate join), and for each separator $S_i \in \mathbf{S}$ (i.e., each intermediate aggregate). Queries are evaluated volcano-style, iterating over the rows of each clique and summing over the product of probabilities as described in Section 2.2. As in Cyclic Sampling, the iteration order is randomized by a LCG (lines 9-13). For each clique $C_i$, the algorithm samples a row $\vec{x}$ (lines 9-12), computes the marginal probability for that row (line 14), and adds it to its running aggregate for the group $\vec{q}$ that $\vec{x}$ belongs to (line 20). It is necessary to avoid double-counting samples in the second and subsequent cycles of the LCG. Consequently, the algorithm updates the separator using the difference $\delta_{prob}$ between the newly computed marginal and the previous version (lines 16-19).

In order to determine progress towards convergence, the algorithm also tracks a sample count for each row of the clique and separator tables (lines 15, 17), as well as the total, aggregate count for each separator table (line 21). Informally, this count is the number of distinct tuple lineages represented in the current estimate of the probability value. For a given clique table $C_i$ the maximum value of this count is the product of the sizes of the domains of all variables eliminated (aggregated away) in $C_i$ (line 3). For example,

**Algorithm 3** `EvaluateLeakyJoins`($\mathcal{B}$, $Q$)

**Require:** A bayes net $\mathcal{B} = (\mathcal{G}(\mathbf{X}), \Phi)$
**Require:** An inference query $Q = P(\mathbf{X_q})$
**Ensure:** The result separator $S_{target} = P(\mathbf{X_q})$
1: $\langle \mathbf{S}, \mathbf{C} \rangle \leftarrow$ `assemblePlan`$(\mathcal{B}, \mathbf{X}_q)$
2: **for each** $i \in 1 \ldots |\mathbf{S}|$ **do**
3: $samples_i \leftarrow 0$; $maxSamples_i = |dom(desc(S_i))|$
4: $a_i, b_i, m_i \leftarrow$ `initLCG`$(|dom(C_i)|)$
5: $index_i \leftarrow$ `rand_int`() $\mod m_i$
6: Fill $S_i$ and $C_i$ with $\langle\ prob : 0.0\ ,\ count : 0\ \rangle$
7: **while** there is an $i$ with $samples_i < maxSamples_i$ **do**
8: **for each** $i$ where $samples_i < maxSamples_i$ **do**
9: `/* Step the LCG */`
10: $index_i \leftarrow (a_i * index_i + b_i) \mod |dom(\psi_i)|$
11: `/* Demux index as Alg. 2 lines 9-11 */`
12: Get $\vec{x}$ from $index_i$
13: `/* Get the joint probability as Alg. 2 line 13 and get the joint sample count similarly */`
14: $prob \leftarrow \prod_{\phi \in factors(C_i)} \left(\phi\left[\pi_{scope(\phi)}(\vec{X})\right].prob\right)$
15: $count \leftarrow \prod_{\phi \in factors(C_i)} \left(\phi\left[\pi_{scope(\phi)}(\vec{X})\right].count\right)$
16: `/* Compute update deltas */`
17: $\langle \delta_{prob}, \delta_{count} \rangle = \langle\ prob, count\ \rangle - C_i[\vec{x}]$
18: `/* Apply update deltas */`
19: $C_i[\vec{x}] = C_i[\vec{x}] + \langle \delta_{prob}, \delta_{count} \rangle$
20: $\vec{q} = \pi_{scope(S_i)}(\vec{x})$; $S_i[\vec{q}] = S_i[\vec{q}] + \langle \delta_{prob}, \delta_{count} \rangle$
21: $samples_i = samples_i + \delta_{count}$

in Figure 2, no variables have been eliminated in $C_1$ so each row of $C_1$ contains at most one sample. By $C_2$, the variable $D$ has been eliminated, so each row of $C_2$ can represent up to $|dom(D)| = 2$ samples. Similarly each row of $C_3$ represents up to $|dom(D)| \cdot |dom(I)| = 4$ samples. The algorithm uses the sample count as a termination condition. Once a table is fully populated, sampling on it stops (line 8). Once all tables are fully populated, the algorithm has converged (line 7).

In short, Leaky Joins work by trickling samples down through each level of the join graph. The cyclic sampler provides flow control and acts as a source of randomization, allowing all stages to produce progressively better estimates in parallel. With each cycle through the samples, improved estimates from the join operator's input are propagated to the next tier of the query.

EXAMPLE 3. *As an example of Leaky Joins, consider a subset of the graph in Figure 1 with only nodes $D, I, G$ and an inference query for $p(G)$. Using classical heuristics from variable elimination, the Leaky Joins algorithm elects to eliminate $D$ first and then $I$, and as a result assembles the intermediate clique cluster $\boldsymbol{C}$ and clique separator $\boldsymbol{S}$ tables as shown in Figure 3. Samples are generated for each intermediate clique cluster one at a time, following the join order: First from $C_1(D, I, G)$ and then $C_2(I, G)$. As shown in Figure 3b, the first sample we obtain from $C_1$ is $\langle 0, 0, 1 \rangle$*

$$\phi_D(D=0) \cdot \phi_G(D=0, I=0, G=1) = 0.6 \cdot 0.3 = 0.18$$

*$S_1(I, G)$ is correspondingly updated with the tuple $\langle 0, 1 \rangle$ with aggregates $\langle 1, 0.18 \rangle$. Then the second sample is drawn from $C_2(I, G)$. For this example, we will assume the random sampler selects $\langle 0, 1 \rangle$, which has probability:*

$$S_1(I=0, G=1) \cdot \phi_I(I=0)$$

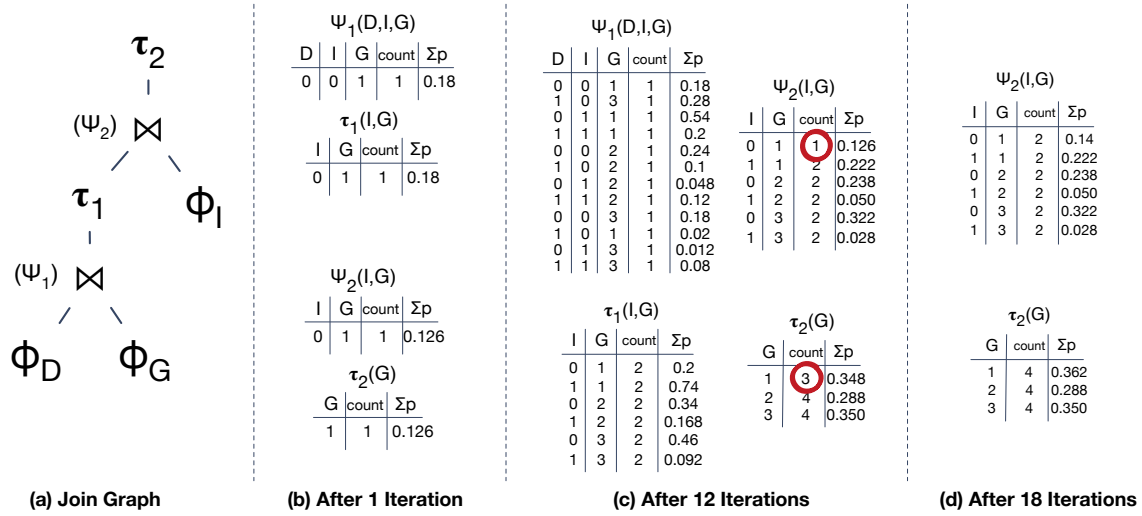



Figure 3: Leaky Joins example join graph (a) and the algorithm's state after 1, 12, and 18 iterations (b-d). In the 12-iteration column (c), incomplete sample counts are circled.

*Although we do not have a precise value for $S_1$ we can still approximate it at this time and update $S_2$ accordingly. After 12 samples, $C_1$ has completed a round of sampling and is ready to be finalized. The state at this point is shown in Figure 3c. Note that the approximation of $S_2$ is still incorrect — The approximation made in step 1 and several following steps resulted in only partial data for $\psi_2(I = 1, G = 1)$ (circled counts in Fig. 3c). However, this error will only persist until the next sample is drawn for $C_2(0, 1)$, at which point the system will have converged to a final result.*

**Cost Model.** We next evaluate the cost of reaching an exact solution using the Leaky Join algorithm. Assume we have $k$ random variables $X_1, ..., X_k$, and the corresponding $k$ factors $\phi_{X_1}, ..., \phi_{X_k}$. Furthermore, assume the variables are already arranged in the optimal elimination order, and we wish to marginalize out variables $X_1, \ldots, X_j$. Leaky joins generates exactly the same set of $j$ cliques and separators as Variable Elimination. Like variable elimination, we can measure the computation complexity by counting multiplication and addition steps. The primary difference between Variable Elimination and Leaky Joins is that some aggregation steps will base on approximations and must be repeated multiple times. Let $V$ denote the cost of constructing the largest joint factor in Variable Elimination (i.e., the time complexity of Variable Elimination is $O(V)$). After $V$ iterations, the lowest level of the join tree is guaranteed to be finalized. After a successive $V$ iterations, the second level of the tree is guaranteed to be finalized, and so forth. The maximum depth of the join tree is the number of variables $k$, so a loose bound for the complexity Leaky Joins is $O(kV)$.

**Confidence Bound.** In Section 3.1, we showed an $\epsilon$-$\delta$ bound for random sampling without replacement (Formula (2)). Here, we extend this result to give a loose bound for Leaky Joins. The primary challenge is that, in addition to sampling errors in the Leaky Join itself, the output can be affected by cumulative error from the join's inputs. We consider the problem recursively. The base case is a clique that reads from only input factors — the lowest level of joins in the query plan. Precise values for inputs are available immediately and Formula (2) can be used as-is.

Next, consider a clique $C_2$ computed from only a single leaky join output. Thus, we can say that $C_2 = S_1 \bowtie \phi$, where $\phi$ is the natural join of all input factors used by $C_2$. There are $|dom(S_1)|$ rows in $S_1$, so after $n$ sampling steps, each row of $S_1$ will have received $\frac{n}{|dom(S_1)|}$ samples. Denote the maximum number of samples per row of $S_1$ by $N_1 = \frac{|dom(S_1)|}{|dom(C_1)|}$. Then, by (2), all rows in $S_1$ will have error less than $\epsilon$ with probability:

$$\delta_1 \equiv \delta^{|dom(S_1)|} = exp\left[\frac{-2n\epsilon^2 \cdot (N_1 - 1)}{N_1 - \frac{n}{|dom(S_1)|}}\right] \quad (3)$$

Let us consider a trivial example where the cumulative error in each row of $S_1$ is bounded by $\epsilon$:

| $S_1$ | $X_1$ | $p$ |
|---|---|---|
| | 1 | $p_1 \pm \epsilon$ |
| | 2 | $p_2 \pm \epsilon$ |

| $\phi$ | $X_1$ | $p$ |
|---|---|---|
| | 1 | $p_3$ |
| | 2 | $p_4$ |

Here, the correct joint probabilities for rows of $C_2$ are $p_1 \cdot p_3$ and $p_2 \cdot p_4$ respectively. Thus a fully-sampled $S_2$ (projecting away $X_1$) will be approximated as $(p_1 \pm \epsilon)p_3 + (p_2 \pm \epsilon)p_4$. The cumulative error in this result is $(p_3 + p_4)\epsilon$, or using a pessimistic upper bound of 1 for each $p_i$, at worst $2\epsilon$. Generalizing, if one row of $S_2$ is computed from $k$ rows of $C_1$, the cumulative error in a given row of $S_2$ is at most $k\epsilon$. Repeating (3), sampling error on $S_2$ after $n$ rounds will be bounded by $\epsilon$ with probability $\delta^{|dom(S_2)|}$. After $n$ rounds of sampling, each row of $S_2$ will have received $\frac{n}{|dom(S_2)|}$ rows of $C_2$, so the cumulative error on one row is $\frac{n}{|dom(S_2)|}\epsilon$. Combining (2) and (3), we get that for one row of $S_2$:

$$P\left[|p - \mathbb{E}_{n,x}| < \left(1 + \frac{n}{|dom(S_2)|}\right)\epsilon\right] \leq exp\left[\frac{-2\frac{n}{|dom(S_2)|}\epsilon^2 \cdot (N_2 - 1)}{N_2 - \frac{n}{|dom(S_2)|}}\right] \cdot \delta_1 \quad (4)$$

The joint probability across all rows of $S_2$ is thus:

$$exp\left[\frac{-2n\epsilon^2(N_2 - 1)}{N_2 - \frac{n}{|dom(S_2)|}}\right] \cdot \delta_1^{|dom(S_2)|} \equiv \delta_2 \cdot \delta_1^{|dom(S_2)|}$$

Generalizing to any **left-deep** plan, an error $\epsilon'$ defined as:

$$\epsilon' = \epsilon \cdot \prod_{i=2}^{|\mathbf{X}|} \left(1 + \frac{n}{|dom(S_i)|}\right) \tag{5}$$

is an upper bound on the marginal in $S_{|\mathbf{X}|}$ with probability:

$$\prod_{i=1}^{|\mathbf{X}|} \delta_i^{\left(\prod_{j=1}^{i-1} |dom(S_j)|\right)} = \prod_{i=1}^{|\mathbf{X}|} e^{\left(\left(\prod_{j=1}^{i-1} |dom(S_j)|\right) \frac{-2n\epsilon^2(N_j-1)}{N_j - \frac{n}{|dom(S_j)|}}\right)} \tag{6}$$

Consider a slightly more complicated toy example clique $C_4 = S_3 \bowtie S_1$, where both $S_3$ and $S_1$ both have bounded error $\epsilon_1$ and $\epsilon_3$ respectively.

| $C_1$ | $X_1$ | $p$ |
|---|---|---|
| | 1 | $p_1 \pm \epsilon_1$ |
| | 2 | $p_2 \pm \epsilon_1$ |

| $C_3$ | $X_1$ | $p$ |
|---|---|---|
| | 1 | $p_3 \pm \epsilon_3$ |
| | 2 | $p_4 \pm \epsilon_3$ |

As before, the correct joint probability is $p_1 \cdot p_3 + p_2 \cdot p_4$. Given an $\epsilon' = \max(\epsilon_1, \epsilon_3)$, the estimated probability will be $p_1 \cdot p_3 + p_2 \cdot p_4 + \left(\sum_{i=1}^{4} p_i \epsilon'\right) + \epsilon'^2$. As before, using an upper bound of 1 for each $p_1 \ldots p_4$ bounds the error by:

$$(|dom(S_1)| \cdot |dom(S_3)|)\epsilon' + \epsilon'^2$$

More generally, the predicted value across $m$ source tables is a sum of terms of the form $(p+\epsilon')$, and the overall cumulative error per element of $C_1$ is bounded as:

$$\epsilon_{cum} = \sum_{i=1}^{m-1} (m \mathcal{C} i)\epsilon^{m-i}$$

where $m \mathcal{C} i$ is the combinatorial operator $m$ choose $i$. Thus re-using Equation (4), we can solve the recursive case of a separator with $m$ leaky join inputs that each have error bounded by $\epsilon'$ with probability $\delta_{cum} = \prod_i^m \delta_i$. Then the total error on one row of the separator can be bounded by $\epsilon + \epsilon_{cum}$ with probability:

$$exp\left[\frac{-2\frac{n}{|dom(S_2)|}\epsilon^2 \cdot (N_2 - 1)}{N_2 - \frac{n}{|dom(S_2)|}}\right] \cdot \delta \tag{7}$$

The joint error is computed exactly as before. To estimate the error on the final result, we apply this formula recursively on the full join plan.

# 4. LESSONS LEARNED FROM IVM

Materialized views are the precomputed results of a so-called view query. As the inputs to this query are updated, the materialized results are updated in kind. Incremental view maintenance (IVM) techniques identify opportunities to compute these updates more efficiently than re-evaluating the query from scratch. Incremental view maintenance has already seen some use in Monte Carlo Markov Chain inference [43], and recent advances — so called recursive IVM techniques [4, 25] have made it even more efficient.

Our initial attempts at convergent inference were based on IVM and recursive IVM in particular. It eventually became clear that there was a fundamental disconnect between these techniques and the particular needs of graphical inference. In the interest of helping others to avoid these pitfalls, we use this section to outline our basic approach and to explain why, perhaps counter-intuitively, both classical and recursive IVM techniques are a poor fit for convergent inference on graphical models.

## 4.1 The Algorithm

Our first approach at convergent inference used IVM to compute and iteratively revise an inference query over a progressively larger fraction of the input dataset. That is, we declared the inference query as a materialized view using exactly the query defined in Section 2.2. The set of factor tables was initially empty. As in Cyclic Sampling, we iteratively insert rows of the input factor tables in a shuffled order. A backend IVM system updates the inference result, eventually converging to a correct value once all factor rows have been inserted. This process is summarized in Algorithm 4.

**Algorithm 4** `SimpleIVM-CIA`($\mathcal{B}$, $Q$)

**Require:** A bayes net $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,P)
**Require:** A conditional probability query: Q=$P(\mathbf{X_q})$
**Ensure:** The set $ret_{\mathbf{X_q}} = P(\mathbf{X_q})$
1: **for each** $\phi_i \in G(\mathbf{X})$ **do**
2: $index_{0,i} =$ `rand_int()` mod $|dom(X_i)|$
3: $a_i, b_i, m_i \leftarrow$ `InitLCG`($|dom(X_i)|$)
4: $\phi_i' \leftarrow \emptyset$
5: Compile IVM Program $Q'$ to compute $P(\mathbf{X_q})$ from $\{\phi_i'\}$
6: **for each** $k \in 1 \ldots max_i(|dom(X_i)|)$ **do**
7: **for each** $\phi_i \in G(\mathbf{X})$ **do**
8: **if** $k \leq |dom(X_i)|$ **then**
9: $index_{k,i} \leftarrow (a * index_{k-1,i} + b) \mod m_i$
10: Update $Q'$ with row $\phi_i'[index_{k,i}] \leftarrow \phi_i[index_{k,i}]$
11: $ret_{\mathbf{X_q}} \leftarrow$ the output of $Q'$

While naive cyclic sampling samples directly from the output of the join, IVM-CIA constructs the same output by iteratively combining parts of the factor tables. The resulting update sequence follows a pattern similar to that of multi-dimensional Ripple Joins [17], incrementally filling the full sample space of the join query. As in naive cyclic sampling, this process may be interrupted at any time to obtain an estimate of $P(\mathcal{Y})$ by taking the already materialized partial result and scaling it by the proportion of samples used to construct it. This proportion can be computed by adding a `COUNT(*)` aggregate to each query. IVM-CIA uses the underlying IVM engine to simultaneously track both the estimate and the progress towards an exact result.

## 4.2 Post-Mortem

For our first attempt at an IVM-based convergent inference algorithm, we used DBToaster [25], a recursive IVM compiler. DBToaster is aimed at relational data and uses a sparse table encoding that, as we have already mentioned, is ill suited for graphical models. Recognizing this as a bottleneck, we decided to create a modified version of DBToaster that used dense array-based table encodings. Although this optimization did provide a significant speed-up, the resulting engine's performance was still inferior: It converged much slower than variable elimination and had a shallower result quality ramp than the approximation techniques (even cyclic sampling in some cases).

Ultimately, we identified two key features of graphical models that made them ill-suited for database-centric IVM and in particular recursive IVM. First, in Section 3, we noted that nodes in a Bayesian Network tend to have many neighbors and that the network tends to have high hypertree-width. The size of intermediate tables is exponential in the

hypertree-width of the query — rather large in the case of graphical models. Recursive IVM systems like DBToaster are in-effect a form of dynamic programming, improving performance by consuming more space. DBToaster in particular maintains materialized copies of intermediate tables for all possible join plans. As one might imagine, the space required for even a BN of moderate size can quickly outpace the available memory.

The second, even more limiting factor of IVM for graphical models is the fan-out of joins. Because of the density of a graphical model's input factors and intermediate tables, joins are frequently not just many-many, but all-all. Thus a single insertion (or batch of insertions) is virtually guaranteed to affect every result row. In recursive IVM, the problem is worse, as each insertion can trigger full-table updates for nearly every intermediate table (which as already noted, can be large). Batching did improve performance, but not significantly enough to warrant replacing cyclic sampling.

Our approach of Leaky Joins was inspired, in large part, by an alternative form of recursive IVM initially described by Ross et. al., [32], which only materializes intermediates for a single join plan. Like this approach, Leaky Joins materializes only a single join plan, propagating changes through the entire query tree. However, unlike the Ross recursive join algorithm, each Leaky Join operator acts as a sort of batching blocker. New row updates are held at each Leaky Join, and only propagated in a random order dictated by the LCG.

# 5. EVALUATION

Recall in Section 3, we claimed CIAs should satisfy four properties. In this section we present experimental results to show that they do. Specifically, we want to show: (1,2) **Flexibility:** CIAs are able to provide both approximate results and exact results in the inference process. (3) **Approximation Accuracy:** Given the same amount of time, CIAs can provide approximate results with an accuracy that is competitive with state-of-the-art approximation algorithms. (4) **Exact Inference Efficiency:** The time a CIA takes to generate an exact result is competitive with state-of-the-art exact inference algorithms.

## 5.1 Experimental Setup and Data

Experiments were run on a 12 core, 2.5 GHz Intel Xeon with 198 GB of RAM running Ubuntu 16.04.1 LTS. Our experimental code was written in Java and compiled and run single-threaded under the Java HotSpot version 1.8 JDK/JVM. For experiments, we used five probabilistic graphical models from publicly available sources, including the bnlearn Machine Learning Repository [35] to compare the available algorithms. Visualizations of all five graphs are shown in Figure 4.

**Student.** The first data set is the extended version of the Student graphical model from [26]. This graphical model contains 8 random variables. All the random variables are discrete. In order to observe how CIAs are influenced by exponential blowup of scale, we use this graph as a micro-benchmark by generating synthetic factor tables for the Student graph. In the synthetic data, we vary the domain size of each random variable from 2 to 25. Marginals were computed over the `Happiness` attribute.

**Child.** The second graphical model captures the symptoms and diagnosis of Asphyxia in children [12]. The number of random variables in the graph is 20. All the random variables are discrete with different domain sizes. There are 230 parameters in factors. The average degree of nodes in the graph is 3.5 and the maximum in-degree in the graph is 4. Marginals were computed over the `sick` variable.

**Insurance.** The third graphical model we used models potential clients for car insurance policies [8]. It contains 27 nodes. All the random variables are discrete with different domain sizes. There are 984 parameters in factors. The average number of degree is 3.85 and the maximum in-degree in the graph is 3. Marginals were computed over the `PropCost` variable.

**Barley.** The fourth graphical model is developed from a decision support system for mechanical weed control in malting barley [27]. The graph contains 48 nodes. All the random variables are discrete with different domain sizes. There are 114005 parameters in factors. The average degree of nodes in the graph is 3.5 and the maximum in-degree in the graph is 4. Marginals were computed over the `ntilg` variable.

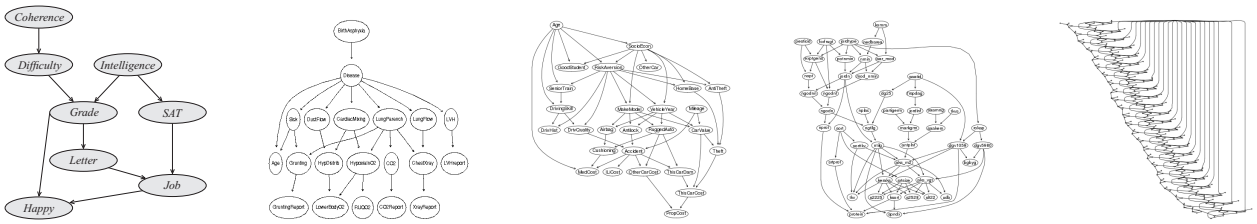
**Diabetes.** The fifth graphical model is a very large graph that captures a model for adjusting insulin [5]. The number of random variables in the graph is 413. All the random variables are discrete with different domain sizes. There are 429409 parameters in factors. In average, there are 2.92 degrees and the maximum in-degree in the graph is 2. Marginals were computed over the `bg_5` variable.

## 5.2 Inference Methods

We compare our two convergent inference algorithms with variable elimination (for exact inference results) and Gibbs sampling (for approximate inference). We are interested in measuring runtime for exact inference and accuracy for approximate inference. We assign an index for each node in the Bayes net following the topological order. We assume that for each factor $\phi_i$, the variables are ordered by the following conventions: $pa(\mathbf{X}) \prec \mathbf{X}$, where $\mathbf{X}=\{X_i, \ldots, X_j, \ldots\}$ is ordered increasingly by index. The domain values in each random variable is ordered increasingly.

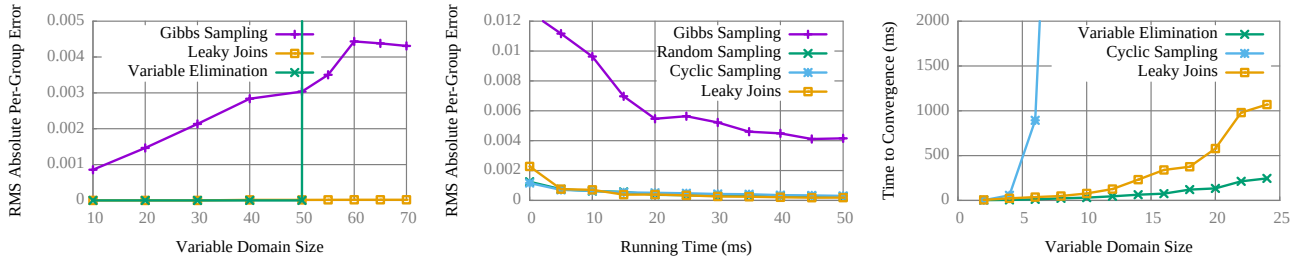
**Variable Elimination.** Variable elimination is the classic exact inference algorithm used for graphical models, as detailed in Section 2. As is standard in variable elimination, intermediate join results are streamed and never actually materialized. To decide on an elimination (join) ordering, we adopt the heuristic methods in [26]. At each point, the algorithm evaluates each of the remaining variables in the Bayes net based on a heuristic cost function. The cost criteria used for evaluating each variable is Min-weight, where the cost of a node is the product of weights, where weight is the domain cardinality of the node's neighbors. For example, using Min-weight, the selected order for the extended student graph in Figure 4a is: $C$, $D$, $S$, $I$, $L$, $J$, $G$. $H$ is the target random variable.

**Gibbs Sampling.** As discussed in Section 2, Gibbs sampling first generates the initial sample with each variable $X_i$ in $\mathcal{B} = (\mathcal{G}(\mathbf{X}), \Phi)$ following the conditional probability distribution $p(X_i|pa(X_i))$. Then, we randomly fix the value for some random variable $X_j$ and use it as evidence to generate next sample. With more and more samples collect, the distribution will get increasingly closer to the posterior. We skip the first hundred samples for small graphs 4a, 4b and 4c, and thousand samples for larger graphs 4d and 4e at the beginning of the sample process. The target probability




(a) Extended Student (b) Child (c) Insurance (d) Barley (e) Diabetes

Figure 4: Visualizations of five graphical models from [35] used in our experiments.




(a) Accuracy within 10ms (b) Accuracy over time (c) Running time to convergence

Figure 5: Microbenchmarks on the synthetic, extended Student graph (Figure 4a)

distribution is calculated by normalizing the sum of sample frequencies for each value in the target variables $X_q$.

**Cyclic Sampling.** Cyclic Sampling is our first CIA, described in Section 3.1 and in Algorithm 2. Note that cyclic sampling does not materialize the joint probability distribution, but rather constructs rows of the joint distribution dynamically using a LCG (Equation 1). This process takes constant time for a fixed graph.

**Leaky Joins.** Leaky Joins, our second CIA, were described in Section 3.2. We use the same elimination (join) order as in variable elimination algorithm to construct the clique tree and materialize intermediate tables, Clique's clusters **C** and Clique's seperator **S**. Then we conduct the "passing partial messages" process according to Algorithm 3.

## 5.3 Flexibility

We first explore the flexibility of CIAs by comparing the accuracy of different algorithms (both exact and approximate) within a finite cutoff time. We imitate the situation that time is the major concern for user and the goal is to provide an accurate inference result at a given time. We compute the marginal probability, cutting each algorithm off after the predefined period, and average the fractional error across each marginal "group".

Figure 5a shows the average fractional error for each inference algorithm on the **student** graph with a cutoff of 10 seconds. We vary the factor size from 10 to 70 to simulate small and large graphs. Variable elimination provides an exact inference result for variables with domain size smaller than 50. On larger graphs, it times out, resulting in a 100% error. Gibbs sampling can always provide an approximate result, but produces results that are inaccurate, even when variable elimination can produce an exact result. Leaky joins provide exact inference results when the domain size is smaller than 40, but when the domain size passes 40, it still provides approximate results with a lower error than Gibbs sampling. This graph shows that, with leaky joins, the same algorithm can support both the exact inference and approximate inference cases; neither users or inference engines need to anticipate which class of algorithms to use.

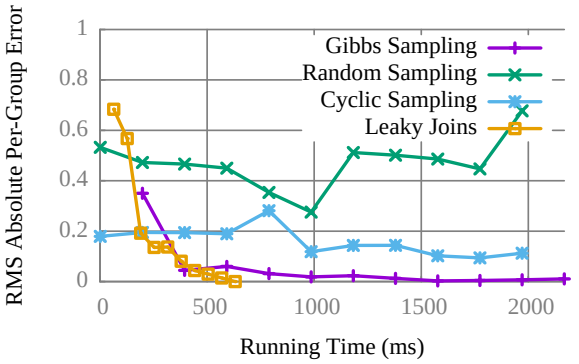
## 5.4 Approximate Inference Accuracy

Figure 5b compares each algorithm's approximate accuracy relative to time spent on the **student** graph with a node size of 35. At each time $t$, the average fractional error of leaky joins is smaller than that of Gibbs sampling. In addition, leaky join converges to exact result, which Gibbs sampling will never do. The steep initial error in leaky joins at the start stems from the first few sampling rounds for each intermediate table $C_i$ being based on weak preliminary approximations; The algorithm needs some burn-in time to have samples to cover their domains to provide approximate results. Gibbs sampling algorithm also has burn-in process. The sharp curve for Gibbs sampling is because it takes more samples for Gibbs sampling to cover corner cases, and generate samples with less probabilities. Figure 6a, Figure 6b and Figure 6d show the approximate accuracy result for child and insurance graph. Gibbs sampling performs well in this two graph for that the domain of the target variables $X_q$ are small (the domain of sick node for child graph is two and propcost for insurance is 4). There will be less corner cases and by *Chernoff's bound* [11], the number of samples to required decreases as the probability of $\mathrm{P}(X_q)$ increases.

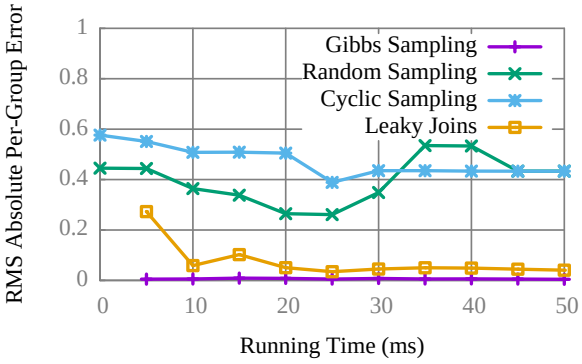
$$P_{est}(P_{est}(X_q) \notin P(X_q)(1 \pm \epsilon)) \leq 2e^{-NP(X_q)\epsilon^2/3} \leq \delta.$$

The result shows that even in this situation, the approximate result of leaky joins is still comparable to Gibbs sampling. Of these four graphs, the "Diabetes" graph was the most complex, and only Leaky Joins produced meaningful results.

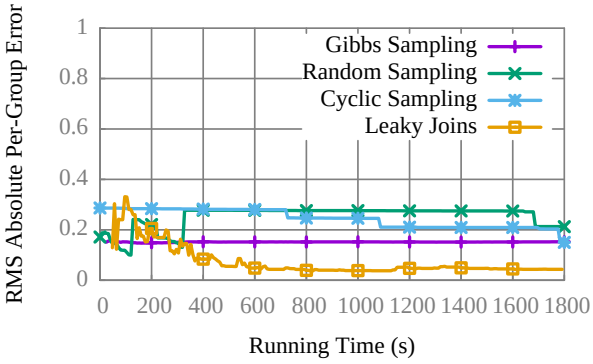
For comparison with the accuracy results in Figure 6, Variable Elimination produces exact results for "Child" in 19 ms, for "Insurance" in 49 ms, for Barley in approximately 1.4 hours, and for Diabetes in approximately 1.5 hours.



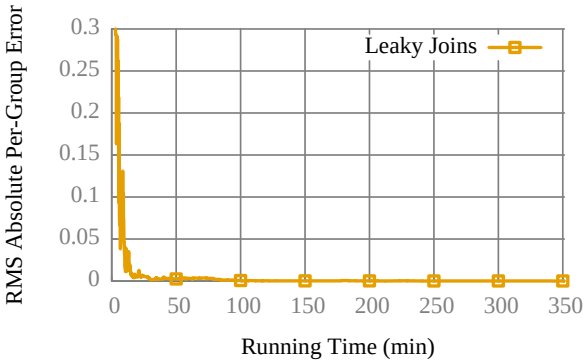

(a) The "Child" Graph (Figure 4b)




(b) The "Insurance" Graph (Figure 4c)




(c) The "Barley" Graph (Figure 4d)




(d) The "Diabetes" Graph (Figure 4e)

Figure 6: Approximation accuracy for real-world graphs

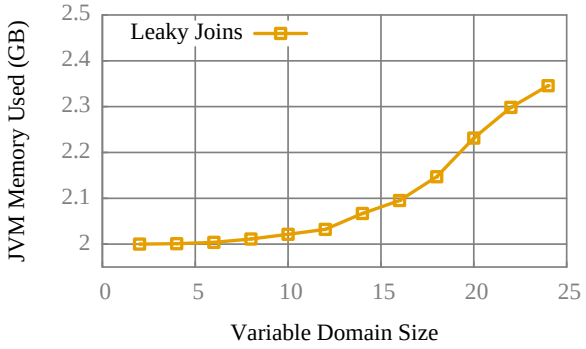



Figure 7: JVM Memory use for Cyclic Sampling. Note that at startup, Java has already allocated roughly 2 GB.

## 5.5 Convergence Time

Figure 5c shows the exact running time for variable elimination and leaky joins. Recall that the running complexity of variable elimination and leaky join is dominated by the size of the clique's cluster, $O(k|C_{max}|)$, where $|C_{max}|$ is the size of largest clique's cluster. The difference is that leaky join has a constant k. As the factor size increases, $C_{max}$ increases and both algorithms get slower at equivalent rates.

## 5.6 Memory

Unlike variable elimination and many of the approximate algorithms, leaky joins does continuously maintain materialized intermediate results. To measure memory use, we used the JVM's `Runtime.totalMemory()` method, sampling immediately after results were produced, but before garbage collection could be run. Figure 7 shows Java's memory usage with leaky joins for the "Student" micro benchmark. The actual memory needs of leaky joins are comparatively small: The two largest intermediate results have roughly 20-thousand rows, with 5 columns each. Overall, Leaky Joins only forms a small portion of Java's overall footprint.

# 6. CONCLUSIONS AND FUTURE WORK

We introduced a class of convergent inference algorithms (CIAs) based on sampling without replacement using linear congruential generators. We proposed CIAs built over incremental view maintenance and a novel aggregate join algorithm that we call Leaky Joins. We evaluated both IVM-CIA and Leaky Joins, and found that Leaky Joins were able to approximate the performance of Variable Elimination on simple graphs, and the accuracy of state-of-the-art approximation techniques on complex graphs. As graph complexity increased, the bounded-time accuracy of Leaky Joins degraded gracefully.

Our algorithms has one limitation which represents opportunities for future work. Our algorithm didn't consider scalability. In the era of Big Data, distributed inference in graphical model is necessary for performance. Similar to works for parallelizing existing inference algorithms [7, 14], our algorithm is possible to run in parallel.

**Acknowledgements.** This work was supported by a gift from Oracle Academic Relations, by NPS Grant N00244-16-1-0022 and by NSF Grants #1409551 and #1640864. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of Oracle, the Naval Postgraduate School, or the National Science Foundation.

# 7. REFERENCES


[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. *SIGMOD Rec.*, 28(2):574–576, 1999.

[2] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: Interactive queries on very large data. *pVLDB*, 5(12):1902–1905, 2012.

[3] S. Agarwal, A. Panda, B. Mozafari, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. Technical report, ArXiV, 03 2012.

[4] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *pVLDB*, 5(10):968–979, 2012.

[5] S. Andreassen, R. Hovorka, J. Benn, K. G. Olesen, and E. R. Carson. A model-based approach to insulin adjustment. In *AIME 91*, pages 239–248. Springer, 1991.

[6] S. Arumugam, F. Xu, R. Jampani, C. Jermaine, L. L. Perez, and P. J. Haas. MCDB-R: Risk analysis in the database. *pVLDB*, 3(1-2):782–793, 2010.

[7] R. Bekkerman, M. Bilenko, and J. Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[8] J. Binder, D. Koller, S. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29(2-3):213–244, 1997.

[9] H. C. Bravo and et al. Optimizing mpf queries: Decision support and probabilistic inference, 2007.

[10] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[11] H. Chernoff. A career in statistics. *Past, Present, and Future of Statistical Science*, page 29, 2014.

[12] A. P. Dawid. Prequential analysis, stochastic complexity and bayesian inference. *Bayesian statistics*, 4:109–125, 1992.

[13] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.

[14] F. Diez and J. Mira. Distributed inference in bayesian networks. *Cybernetics and Systems: An International Journal*, 25(1):39–61, 1994.

[15] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in DBO. *pVLDB*, 2(1):419–430, Aug. 2009.

[16] P. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, pages 51–62, 1997.

[17] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, 1997.

[19] W. Hoeffding. Probability inequalities for sums of bounded random variables. *JASS*, 58(301):13–30, 1963.

[20] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. MCDB: A monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[21] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. *TODS*, 33(4):23:1–23:54, 2008.

[22] O. Kennedy and S. Nath. Jigsaw: Efficient optimization over uncertain enterprise data. In *SIGMOD*, 2011.

[23] A. Klein, R. Gemulla, P. Rösch, and W. Lehner. Derby/S: A DBMS for sample-based query answering. In *SIGMOD*, 2006.

[24] D. Knuth. The art of computer programming. semi-numerical algorithms. 1968.

[25] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDBJ*, 23(2):253–278, 2014.

[26] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[27] K. Kristensen and I. Rasmussen. A decision support system for mechanical weed control in malting barley. In *ECITA*, 1997.

[28] P. L'Ecuyer. Random numbers for simulation. *CACM*, 33(10):85–97, 1990.

[29] C. Mayfield, J. Neville, and S. Prabhakar. Eracer: A database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[30] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, 1986.

[31] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *CACM*, 31(10):1192–1201, 1988.

[32] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. *SIGMOD Rec.*, 25(2):447–458, 1996.

[33] F. Rusu and A. Dobra. Glade: A scalable framework for efficient analytics. *OSR*, 46(1):12–18, Feb. 2012.

[34] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley & Sons, 2007.

[35] M. Scutari. The bnlearn bayesian network repository. http://www.bnlearn.com/bnrepository/.

[36] P. Sen, A. Deshpande, and L. Getoor. Prdb: Managing and exploiting rich correlations in probabilistic databases. *VLDB Journal, special issue on uncertain and probabilistic databases*, 2009.

[37] R. J. Serfling. Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics*, pages 39–48, 1974.

[38] F. L. Severence. *System modeling and simulation: an introduction*. John Wiley & Sons, 2009.

[39] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. *The Architecture of SciDB*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[40] D. Z. Wang, Y. Chen, C. E. Grant, and K. Li. Efficient in-database analytics with graphical models. *IEEE Data Eng. Bull.*, 37(3):41–51, 2014.

[41] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick. Hybrid in-database inference for declarative information extraction. In *SIGMOD*, 2011.

[42] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. *pVLDB*, 1(1):340–351, 2008.

[43] M. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *pVLDB*, 3(1-2):794–804, 2010.

[44] M. L. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and MCMC. *CoRR*, abs/1005.1934, 2010.

[45] S. Wu, C. Zhang, F. Wang, and C. Ré. Incremental knowledge base construction using deepdive. *CoRR*, abs/1502.00731, 2015.