StarfishDB: a Query Execution Engine for Relational Probabilistic Programming

OUAEL BEN AMARA*, University of Michigan-Dearborn, U.S.A.
SAMI HADOUAJ*, University of Michigan-Dearborn, U.S.A.
NICCOLÒ MENEGHETTI, University of Michigan-Dearborn, U.S.A.

We introduce StarfishDB, a query execution engine optimized for relational probabilistic programming. Our engine adopts the model of Gamma Probabilistic Databases, representing probabilistic programs as a collection of relational constraints, imposed against a generative stochastic process. We extend the model with the support for recursion, factorization and the ability to leverage just-in-time compilation techniques to speed up inference. We test our engine against a state-of-the-art sampler for Latent Dirichlet Allocation.

CCS Concepts: • **Information systems** → **Uncertainty**; • **Computing methodologies** → **Statistical relational learning**; **Probabilistic reasoning**; • **Mathematics of computing** → *Gibbs sampling*.

Additional Key Words and Phrases: Probabilistic Programming, Probabilistic Databases, Bayesian Inference

ACM Reference Format:
Ouael Ben Amara*, Sami Hadouaj*, and Niccolò Meneghetti. 2024. StarfishDB: a Query Execution Engine for Relational Probabilistic Programming. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 185 (June 2024), 31 pages. https://doi.org/10.1145/3654988

1 INTRODUCTION

Probabilistic Programming (PP) is one of the most successful formalisms to design and deploy Bayesian statistical models [127]. A *probabilistic program* consists of two components: (i) a formal specification of a stochastic generative process and (ii) a collection of observed outcomes for such process. Together, they define a posterior probability distribution over the latent variables of the generative process, obtained by conditioning the process with respect to the given observations. Starting from a generic probabilistic program, a PP compiler can automatically synthesize an inference algorithm to approximate the posterior distribution, without requiring any additional implementation effort from the end user. This approach has been very popular, giving rise to a plethora of languages, like Stan [19], Edward [125], PyMC3 [112] and many others [6, 44, 72, 75, 82, 119, 124].

Recently, a new breed of PP frameworks has been developed in the context of Statistical Relational Learning (or SRL[106]). These languages adopt a *declarative* approach to probabilistic programming, by expressing a program as a *relational* stochastic process that is conditioned by first-order logic constraints. The two most recent additions to this line of work are Gamma Probabilistic Databases (or Gamma-PDBs [78]) and Probabilistic Programming Datalog (or PPDL [6]). The framework of

*These authors contributed equally to this work.

Authors' addresses: Ouael Ben Amara*, benamara@umich.edu, University of Michigan-Dearborn, Dearborn, MI, U.S.A.; Sami Hadouaj*, shadouaj@umich.edu, University of Michigan-Dearborn, Dearborn, MI, U.S.A.; Niccolò Meneghetti, niccolom@umich.edu, University of Michigan-Dearborn, Dearborn, MI, U.S.A..

ACM 2836-6573/2024/6-ART185
https://doi.org/10.1145/3654988

Gamma-PDBs is a novel database-centric probabilistic programming language, where probabilistic programs are expressed as relational algebra constraints [1], imposed against a probabilistic database [120]. PPDL introduces a probabilistic variant of Datalog [21], that can express probabilistic programs through the use of relational constraints. **The main technical contribution of this paper is to merge these two distinct and independent lines of research, introducing a novel PP language that combines elements of [78] and [6]**. In doing so, we improve on the state-of-the-art in several directions:

(1) PPDL is strictly more expressive than Gamma-PDBs, but lacks any mechanism to translate a probabilistic Datalog theory into a functional inference algorithm; consequently, it lacks any functional implementation. We address this problem by identifying a syntactic restriction to PPDL that allows us to apply the techniques from [78] to synthesize a Gibbs sampler from a declarative PPDL theory. We verify this first claim by designing a PPDL theory for the Latent Dirichlet Allocation model (or LDA [13]), from which our system can synthesize a functional collapsed Gibbs sampler.
(2) We show that this newly identified, restricted PPDL language from (1) is strictly more expressive than the language originally adopted by Gamma-PDBs [78], extending it with a restricted form of *recursion*. We verify this second claim by designing a PPDL theory for Hidden Markov models (or HMMs [104]), from which our system can synthesize a functional inference algorithm. This result extends [78] significantly, since HMMs remain intractable without recursion.
(3) We equip our PP engine with a just-in-time compiler [37], that allows our declarative PP programs to compete in performance with tailor-coded implementations of common inference algorithms. We verify this third claim by comparing our collapsed Gibbs sampler from (1) with Mallet [76], a highly-optimized implementation of the very same inference algorithm.

The remainder of this paper is organized as follows: in Section 2 we formalize a propositional language for probabilistic programming, in Section 3 we provide a lifted representation for the language based on Datalog, in Section 4 we describe the practical implementation of our framework. We conclude our discussion with experimental results.

2 PROPOSITIONAL PROBABILISTIC PROGRAMMING

In this section we introduce a simple propositional language to encode probabilistic programs. We are going to use it as a stepping-stone to later (Section 3) introduce a more powerful, yet less intuitive formalism based on Datalog. The sentences of our propositional language consist of Boolean constraints, that are enforced over a collection of discrete random variables. We start by defining a prior probability distribution over these variables.

2.1 Bayesian Dice

Let $\boldsymbol{\theta}$ be a random vector with d components $(\theta_1, .., \theta_d)$; we assume that $\boldsymbol{\theta}$ follows a Dirichlet distribution, parametrized by some known d-dimensional vector $\boldsymbol{\alpha}$:

$$p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) \propto \prod_{i=1}^{d} \theta_i^{\alpha_i - 1} \tag{1}$$

By construction, $\boldsymbol{\alpha}$ is a vector of positive real numbers (i.e. $\boldsymbol{\alpha} \in \mathbb{R}_+^d$), and $\boldsymbol{\theta}$ ranges over $\mathcal{S}_d$, the (d-1)-dimensional probabilistic simplex

$$\mathcal{S}_d \stackrel{\text{def}}{=} \{(y_1, \ldots, y_d) \in \mathbb{R}_+^d \ : \ \sum_{i=1}^{d} y_i = 1\} \tag{2}$$

Let $\boldsymbol{X} \stackrel{\text{def}}{=} (X[l_1], .., X[l_m])$ be a vector of m categorical random variables, that all range over the same discrete domain $\{v_1, \ldots, v_d\}$. We use the labels $\{l_1, .., l_m\}$ to index the elements of $\boldsymbol{X}$; we also use random vector $\boldsymbol{\theta}$ to parametrize a categorical distribution for all the random variables in $\boldsymbol{X}$:

$$\forall j \in \{1, .., m\} \quad P(X[l_j] \mid \boldsymbol{\theta}) \stackrel{\text{def}}{=} \prod_{i=1}^{d} \theta_i^{\mathbb{1}_{(X[l_j]=v_i)}} \tag{3}$$

Here $\mathbb{1}_{(X[l_j]=v_i)}$ denotes the indicator function, that evaluates to 1 when $X[l_j] = v_i$ holds true, and evaluates to 0 otherwise. Hence, Equation (3) simply states that $P(X[l_j] = v_i \mid \boldsymbol{\theta}) = \theta_i$. Equations (1) and (3), together, define a joint density for both $\boldsymbol{\theta}$ and $\boldsymbol{X}$:

$$p(\boldsymbol{X}, \boldsymbol{\theta} \mid \boldsymbol{\alpha}) \stackrel{\text{def}}{=} P(\boldsymbol{X} \mid \boldsymbol{\theta}) \cdot p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) = \prod_{j=1}^{m} \{P(X[l_j] \mid \boldsymbol{\theta})\} \cdot p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) \tag{4}$$

One can imagine the tuple $(\boldsymbol{\alpha}, \boldsymbol{\theta}, \boldsymbol{X})$ as a simple Bayesian model, that describes a loaded die with d faces: vector $\boldsymbol{\theta}$ represents the (possibly unknown) bias of the die, values $\{v_1, \ldots, v_d\}$ represent the different faces of the die, each variable in $\boldsymbol{X}$ represents the outcome of rolling the die once, and vector $\boldsymbol{\alpha}$ represents our prior knowledge about the bias $\boldsymbol{\theta}$. Following this intuition, from now on we will use the term "*Bayesian die*" to refer to any model in the form $(\boldsymbol{\alpha}, \boldsymbol{\theta}, \boldsymbol{X})$ that respects the assumptions made above. As per common practice, we will refer to $\boldsymbol{\alpha}$ as a vector of *hyperparameters*, and to $\boldsymbol{\theta}$ as a vector of *latent parameters*.

A Bayesian die always admits a closed posterior; the posterior density of $\boldsymbol{\theta}$ after observing variable $X[l_j]$ taking value v_r is just another Dirichlet distribution:

$$p(\boldsymbol{\theta} \mid \boldsymbol{\alpha},\ X[l_j] = v_r) \propto \prod_{i=1}^{d} \theta_i^{\alpha_i + \mathbb{1}_{(i=r)} - 1} \tag{5}$$

The same remains true if we have multiple observations. Let $\hat{\boldsymbol{X}}$ be a vector in $\{v_1, \ldots, v_d\}^m$ and let's denote by $c(r, \hat{\boldsymbol{X}})$ the number of times value v_r appears in $\hat{\boldsymbol{X}}$. The posterior of $\boldsymbol{\theta}$ w.r.t. the m observations $\boldsymbol{X} = \hat{\boldsymbol{X}}$ is given by

$$p(\boldsymbol{\theta} \mid \boldsymbol{\alpha},\ \boldsymbol{X} = \hat{\boldsymbol{X}}) \propto \prod_{i=1}^{d} \theta_i^{\alpha_i + c(i, \hat{\boldsymbol{X}}) - 1} \tag{6}$$

Equations (5) and (6) follow immediately from the properties of conjugate priors. They provide an intuitive strategy to incorporate new information into a Bayesian die: if we observe the m events $\boldsymbol{X} = \hat{\boldsymbol{X}}$, we can update our model to reflect this new evidence by simply increasing the value of each hyper-parameter α_i by $c(i, \hat{\boldsymbol{X}})$ units. Following common practice, we will call these units *pseudo-counts* or *votes*. The marginal likelihood of any variable $X[l_j]$ in $\boldsymbol{X}$ w.r.t. $\boldsymbol{\alpha}$ admits the following closed representation

$$P(X[l_j] \mid \boldsymbol{\alpha}) = \int_{\mathcal{S}_d} P(X[l_j] \mid \boldsymbol{\theta}) \cdot p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) d\boldsymbol{\theta} = \frac{\prod_{i=1}^{d} \alpha_i^{\mathbb{1}_{(X[l_j]=v_i)}}}{\sum_{i=1}^{d} \alpha_i} \tag{7}$$

If we happen to observe $\boldsymbol{\theta}$, all the variables in $\boldsymbol{X}$ are, by construction, pairwise independent, i.e. $P(\boldsymbol{X} \mid \boldsymbol{\theta}) = \prod_{j=1}^{m} P(X_j \mid \boldsymbol{\theta})$. The same *does not* hold true if the value of $\boldsymbol{\theta}$ is unknown:

$$p(\boldsymbol{X} \mid \boldsymbol{\alpha}) \stackrel{\text{def}}{=} \int_{\mathcal{S}_d} P(\boldsymbol{X} \mid \boldsymbol{\theta}) \cdot p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) d\boldsymbol{\theta} \neq \prod_{j=1}^{m} P(X[l_j] \mid \boldsymbol{\alpha}) \tag{8}$$

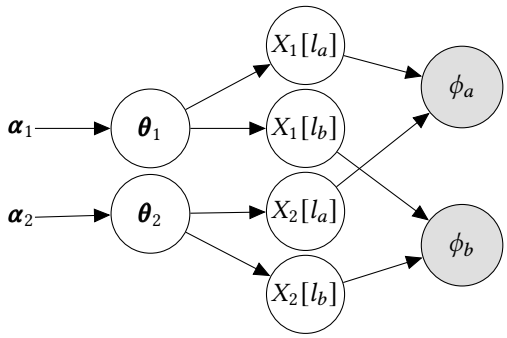


Fig. 1. A basic probabilistic program with two Bayesian dice and two exchangeable constraints (ϕ_a and ϕ_b). Clear nodes represent latent random variables, shaded nodes represent observable variables.

Thus, when $\boldsymbol{\theta}$ is unknown, our knowledge about some unobserved variable $X[l_j]$ may be directly influenced by any observation of any other variable in $\boldsymbol{X}$. For example, if we happen to observe variable $X[l_k]$ taking value v_r, then

$$P(X[l_j] \mid \boldsymbol{\alpha}, X[l_k] = v_r) = \int_{\mathcal{S}_d} P(X[l_j] \mid \boldsymbol{\theta}) \cdot p(\boldsymbol{\theta} \mid \boldsymbol{\alpha}, X[l_k] = v_r) d\boldsymbol{\theta} \tag{9}$$

We can deduce that $P(X[l_j] \mid \boldsymbol{\alpha}) \neq P(X[l_j] \mid \boldsymbol{\alpha}, X[l_k] = v_r)$. In the following we will express the above conclusions by simply stating that any two distinct variables in $\boldsymbol{X}$ are *conditionally independent given* $\boldsymbol{\theta}$; using standard notation we will write

$$\forall (j,k) \in \{1,..,m\}^2 : j \neq k \quad X[l_j] \perp X[l_k] \mid \boldsymbol{\theta} \tag{10}$$

Let $\hat{\boldsymbol{Y}}$ be a vector obtained by applying some arbitrary permutation to the elements of $\hat{\boldsymbol{X}}$; by construction $\hat{\boldsymbol{X}}$ and $\hat{\boldsymbol{Y}}$ will have the same likelihood, i.e. $P(\boldsymbol{X} = \hat{\boldsymbol{Y}} \mid \boldsymbol{\alpha}) = P(\boldsymbol{X} = \hat{\boldsymbol{X}} \mid \boldsymbol{\alpha})$. Whenever this is the case we say that the variables in $\boldsymbol{X}$ are *exchangeable*[28]. In the following sections we will show that exchangeability plays a crucial role in our probabilistic programming language.

2.2 Probabilistic Programs with Boolean Constraints

In order to define a probabilistic program, we start by formulating a stochastic *generative process* $\mathcal{G}$. Our process $\mathcal{G}$ will consist of a collection of N pairwise-independent Bayesian dice $\{\mathcal{D}_1, .., \mathcal{D}_N\}$, where each die $\mathcal{D}_n = (\boldsymbol{\alpha}_n, \boldsymbol{\theta}_n, \boldsymbol{X}_n)$ can have an arbitrary number of faces (denoted as d_n) and an arbitrary number of throws (m_n). In the interest of conciseness, we will denote by $\mathbb{A}$, Θ and $\mathbb{X}$ the sets $\{\boldsymbol{\alpha}_n\}_{n\in\{1,..,N\}}$, $\{\boldsymbol{\theta}_n\}_{n\in\{1,..,N\}}$ and $\{\boldsymbol{X}_n\}_{n\in\{1,..,N\}}$, respectively. The joint distribution of the generative process is given by the following equation:

$$p(\mathbb{X}, \Theta \mid \mathbb{A}) \stackrel{\text{def}}{=} \prod_{n=1}^{N} \left(\prod_{j=1}^{m_n} \left(P(X_n[l_j] \mid \boldsymbol{\theta}_n) \right) \cdot p(\boldsymbol{\theta}_n \mid \boldsymbol{\alpha}_n) \right) \tag{11}$$

Our observations of process $\mathcal{G}$ will consist of Boolean constraints, that are defined over the variables in $\mathbb{X}$. For example, an *atomic constraint* ϕ_{atom} may restrict the value assigned to some variable $X_n[l_j] \in \mathbb{X}$ to a subset V of its domain

$$\phi_{\text{atom}} \stackrel{\text{def}}{=} \left(X_n[l_j] \in V \right) \tag{12}$$

When V contains only one value (say v_r) we will simply write ϕ_{atom} as $(X_n[l_j] = v_r)$. To express more generic constraints, we will compose the atomic ones using standard Boolean operators:

disjunction ($\vee$), conjunction ($\wedge$) and negation ($\neg$). For the sake of conciseness, we will often omit the symbol for logical conjunctions, writing $\phi_1\phi_2$ in place of $\phi_1 \wedge \phi_2$. Similarly, we will often write $\overline{\phi}$ in place of $\neg\phi$. We use the symbols $\top$ and $\bot$ to denote the constant values true and false, respectively.

Example 1. *Let's assume we have a simple generative process $\mathcal{G}$ consisting of two 6-dimensional Bayesian dice, $\mathcal{D}_1 = (\boldsymbol{\alpha}_1, \boldsymbol{\theta}_1, \boldsymbol{X}_1)$ and $\mathcal{D}_2 = (\boldsymbol{\alpha}_2, \boldsymbol{\theta}_2, \boldsymbol{X}_2)$, that range over the set $\{1, 2, 3, 4, 5, 6\}$. The following Boolean expressions define a probabilistic program over $\mathcal{G}$:*

$$\phi_a \stackrel{\text{def}}{=} (X_1[l_a] = 6) \vee (X_2[l_a] = 6)$$

$$\phi_b \stackrel{\text{def}}{=} \bigvee_{i=1}^{6} [(X_1[l_b] = i) \wedge (X_2[l_b] = i)]$$

Constraint ϕ_a states that we rolled both dice once, and observed at least one of the two returning the value 6. Constraint ϕ_b states that we rolled both dice one additional time (l_2), and observed both dice returning the very same number, which remains unknown. If we denote by $\Phi = \{\phi_a, \phi_b\}$ the set of constraints, then the pair $\mathcal{P} = (\mathcal{G}, \Phi)$ represents our probabilistic program. Figure 1 depicts $\mathcal{P}$ as a Bayesian network.

If ϕ is a Boolean constraint defined against process $\mathcal{G}$, we denote by $\textsc{Var}(\phi)$ the set of all the random variables from $\mathbb{X}$ that appear in ϕ. In Example 1, $\textsc{Var}(\phi_a)$ is equal to $\{X_1[l_a], X_2[l_a]\}$, while $\textsc{Var}(\phi_b)$ is equal to $\{X_1[l_b], X_2[l_b]\}$. Without lack of generality, we assume that $\bigcup_{\phi\in\Phi} \textsc{Var}(\phi) = \mathbb{X}$. Additionally, we denote by $\mathcal{D}(\phi)$ the set of all the latent parameters from Θ that are associated with the random variables in $\textsc{Var}(\phi)$. In Example 1 both $\mathcal{D}(\phi_a)$ and $\mathcal{D}(\phi_b)$ evaluate to $\{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2\}$.

We use the term *literal* to indicate any expression that consists of a single atomic constraint or its negation. Any expression κ that consists of a disjunction of literals is called a *clause*. Any expression τ that consists of a conjunction of literals is called a *term*. We say that an expression ϕ is in disjunctive normal form (DNF) if it consists of a disjunction of terms. Similarly, we say that ϕ is in conjunctive normal form (CNF) if it consists of a conjunction of clauses. We say that ϕ is in *negation normal form* (or NNF) if it is negation-free, except for its atomic constraints [27]. We say that ϕ is *read-once* (RO) if no element of $\textsc{Var}(\phi)$ is used more than once within the expression [43]. In Example 1 constraint ϕ_1 is a read-once DNF expression, while ϕ_2 is a CNF expression that does not admit any read-once representation. Both expressions are in negation normal form.

Let V be a set of random variables such that $\mathbb{X} \supseteq V \supseteq \textsc{Var}(\phi)$. We denote by $\textsc{Asst}(V)$ the set of all the possible value-assignments for the variables in V; for convenience, we will often represent such assignments as term expressions. Reusing the terminology from [25], we will refer to the elements of $\textsc{Asst}(\mathbb{X})$ as the *possible worlds* of process $\mathcal{G}$. We denote by $\textsc{Sat}(\phi, V)$ the subset of $\textsc{Asst}(V)$ where expression ϕ evaluates to true. In Example 1 the expression $\textsc{Asst}(\textsc{Var}(\phi_b))$ represents the set $\{(X_1[l_b] = i) \wedge (X_2[l_b] = j)\}_{(i,j)\in\{1,..,6\}^2}$, while $\textsc{Sat}(\phi_b, \textsc{Var}(\phi_b))$ is equal to $\{(X_1[l_b] = i) \wedge (X_2[l_b] = i)\}_{i\in\{1,..,6\}}$. We say that two expressions ϕ_1 and ϕ_2 are logically equivalent when $\textsc{Sat}(\phi_1, \textsc{Var}(\phi_1) \cup \textsc{Var}(\phi_2)) = \textsc{Sat}(\phi_2, \textsc{Var}(\phi_1) \cup \textsc{Var}(\phi_2))$. We say that expression ϕ_1 *entails* expression ϕ_2 (and write $\phi_1 \models \phi_2$) when the expression $\phi_2 \vee \neg\phi_1$ is logically equivalent to $\top$.

Any stochastic process $\mathcal{G}$ entails a well-formed probability distribution over the truth-values of the Boolean constraints that are defined over it; such distribution is described by the following equation

$$P(\phi \mid \mathbb{A}) \stackrel{\text{def}}{=} \sum_{\tau\in\textsc{Sat}(\phi,\mathbb{X})} P(\tau \mid \mathbb{A}) \tag{13}$$

Furthermore, we can condition such distribution w.r.t. any other Boolean constraint

$$P(\phi_1 \mid \phi_2, \mathbb{A}) \stackrel{\text{def}}{=} \frac{P(\phi_1 \wedge \phi_2 \mid \mathbb{A})}{P(\phi_2 \mid \mathbb{A})} \tag{14}$$

$P(\phi \mid \mathbb{A})$ represents the odds of sampling from $\mathcal{G}$ a possible world that satisfies constraint ϕ; $P(\phi_1 \mid \phi_2, \mathbb{A})$ represents the odds of sampling a possible world that satisfies constraint ϕ_1 from the set of possible worlds that satisfy ϕ_2.

We say that two constraints ϕ_1 and ϕ_2 are *independent* when their joint distribution $P(\phi_1, \phi_2 \mid \mathbb{A})$ can be expressed as the product $P(\phi_1 \mid \mathbb{A}) \cdot P(\phi_2 \mid \mathbb{A})$. We say that two constraints ϕ_1 and ϕ_2 are *exchangeable* when $P(\phi_1, \phi_2 \mid \mathcal{D}(\phi_1), \mathcal{D}(\phi_2)) = P(\phi_1 \mid \mathcal{D}(\phi_1)) \cdot P(\phi_2 \mid \mathcal{D}(\phi_2))$. It is easy to see that $\mathcal{D}(\phi_1) \cap \mathcal{D}(\phi_2) = \emptyset$ is a sufficient condition for ϕ_1 and ϕ_2 to be independent; similarly $\textsc{Var}(\phi_1) \cap \textsc{Var}(\phi_2) = \emptyset$ is a sufficient condition for exchangeability. Thus, we can enforce these properties in our probabilistic programs through simple syntactic restrictions. In Example 1 constraints ϕ_a and ϕ_b are exchangeable, but not independent.

We are now ready to define the syntax of our propositional language for probabilistic programming.

Definition 1. *Let $\mathcal{G} = \{\mathcal{D}_1, .., \mathcal{D}_N\}$ be a generative process, and let $\Phi = \{\phi_1, ..\phi_{nc}\}$ be a collection of nc Boolean constraints, defined over $\mathcal{G}$. The pair $\mathcal{P} = (\mathcal{G}, \Phi)$ represents a* well-formed probabilistic program *if the following conditions are met:*

(1) For any constraint $\phi \in \Phi$ all the elements of $\textsc{Var}(\phi)$ are pairwise independent.
(2) All the expressions in Φ are pairwise exchangeable.
(3) Set Φ is satisfiable, i.e. $\textsc{Sat}\left(\bigwedge_{\phi \in \Phi} \phi, \mathbb{X}\right) \neq \emptyset$.

The probabilistic program defined in Example 1 is well-formed. Notice that expression ϕ_b reuses each variable in $\textsc{Var}(\phi_b)$ more than once and this does not constitute a violation of condition (1).

For any well-formed probabilistic program $\mathcal{P} = (\mathcal{G}, \Phi)$, with $\mathbb{A}$, Θ and $\mathbb{X}$ defined in the usual way, our goal is to compute the posterior density of latent parameters Θ w.r.t. Φ. Such posterior is defined by the following equation

$$p(\Theta \mid \Phi, \mathbb{A}) = \sum_{\tau \in \textsc{Sat}(\Phi, \mathbb{X})} p(\Theta \mid \tau, \mathbb{A}) \cdot P(\tau \mid \Phi, \mathbb{A}) \tag{15}$$

While $p(\Theta \mid \tau, \mathbb{A})$ is easy to compute, as per Equation (6), evaluating $p(\Theta \mid \Phi, \mathbb{A})$ by Equation (15) remains impractical, since $\textsc{Sat}(\Phi, \mathbb{X})$ may contain an exponentially large number of terms. Notice that $p(\Theta \mid \Phi, \mathbb{A})$ can be seen as the expected value of $p(\Theta \mid \tau, \mathbb{A})$ when τ is sampled from distribution $P(\tau \mid \Phi, \mathbb{A})$. Therefore, if we are able to devise an efficient method to sample terms in $\textsc{Sat}(\Phi, \mathbb{X})$ from $P(\tau \mid \Phi, \mathbb{A})$, then we will have an effective way to approximate $p(\Theta \mid \Phi, \mathbb{A})$. The framework proposed in [78] successfully achieves this very goal: it shows how to design an efficient Gibbs sampler [41] for $P(\tau \mid \Phi, \mathbb{A})$, for any arbitrary, well-formed probabilistic program. Algorithm 1 shows the pseudo-code of such sampler. Its internal state consists of a mapping ($\mathcal{S}$) that associates each constraint $\phi \in \Phi$ with one term in $\textsc{Sat}(\phi, \textsc{Var}(\phi))$. This mapping is initialized between lines 1 and 3. Once $\mathcal{S}$ is initialized, the expression $\bigwedge_{\phi \in \Phi} \mathcal{S}[\phi]$ is guaranteed to belong to $\textsc{Sat}(\Phi, \mathbb{X})$. This invariant will remain true for the whole execution of the sampler. Between lines 4 and 8 the algorithm repeatedly resamples a satisfying term τ_c for each constraint $\phi_c \in \Phi$. The new term is sampled from distribution $P(\tau_c \mid \phi_c, \tau_{-c}, \mathbb{A})$, where τ_{-c} is a term expression that satisfies all the constraints in Φ, with the exception of ϕ_c. Notice that sampling from $P(\tau_c \mid \phi_c, \tau_{-c}, \mathbb{A})$ is much easier than sampling directly from $P(\tau \mid \Phi, \mathbb{A})$, since expressions ϕ_c and τ_{-c} are, by construction, exchangeable. Furthemore, the distribution $P(\tau_c \mid \phi_c, \tau_{-c}, \mathbb{A})$, used at line 7, constantly shifts during the execution of the algorithm, consistently with the logic of Markov Chain Monte Carlo algorithms: in [78] it is shown that the sequence of updates to $\mathcal{S}$ performed by Algorithm 1 forms a Markov chain that is irreducible and aperiodic, and has $P(\tau \mid \Phi, \mathbb{A})$ at its stationary distribution.

Algorithm 1: GibbsSatSampler

Input: A well-formed probabilistic program $(\mathcal{G}, \Phi)$, with $\mathcal{G} = (\mathbb{A}, \Theta, \mathbb{X})$ and $\Phi = \{\phi_1, .., \phi_{nc}\}$.
Output: A sample τ from $P(\tau \mid \Phi, \mathbb{A})$.

1 **for** $\phi_c \in \Phi$ **do**
2 Select some arbitrary term τ_c from Sat $(\phi_c, \text{Var}(\phi_c))$;
3 $\mathcal{S}[\phi_c] \leftarrow \tau_c$;
4 **repeat as needed**
5 **for** $\phi_c \in \Phi$ **do**
6 $\tau_{-c} \leftarrow \bigwedge_{\phi \in \Phi \setminus \{\phi_c\}} \mathcal{S}[\phi]$;
7 Sample $\tau_c \in$ Sat $(\phi_c, \text{Var}(\phi_c))$ from $P(\tau_c \mid \phi_c, \tau_{-c}, \mathbb{A})$;
8 $\mathcal{S}[\phi_c] \leftarrow \tau_c$;
9 **return** $\bigwedge_{\phi \in \Phi} \mathcal{S}[\phi]$;

Example 1 (continued). *The tables below show a plausible Markov chain built by Algorithm 1, for the simple probabilistic program given in Example 1. The first table shows the evolution of mapping $\mathcal{S}$, while the second reports the posterior distribution of $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ conditioned on $\mathcal{S}$. The first row of each table reports the state at the end of the initialization for-cycle at lines 1 and 3. Each successive row reports the state observed at the end of each iteration of the "repeat" cycle between lines 4 and 8.*

i	$\mathcal{S}[\phi_a]$	$\mathcal{S}[\phi_b]$
(init)	$(X_1[l_a] = 6) \wedge (X_2[l_a] = 3)$	$(X_1[l_b] = 5) \wedge (X_2[l_b] = 5)$
0	$(X_1[l_a] = 5) \wedge (X_2[l_a] = 6)$	$(X_1[l_b] = 6) \wedge (X_2[l_b] = 6)$
1	$(X_1[l_a] = 1) \wedge (X_2[l_a] = 6)$	$(X_1[l_b] = 2) \wedge (X_2[l_b] = 2)$
2	$(X_1[l_a] = 6) \wedge (X_2[l_a] = 6)$	$(X_1[l_b] = 3) \wedge (X_2[l_b] = 3)$
...	...	...

i	$p(\boldsymbol{\theta}_1 \mid \mathbb{A}, \mathcal{S}[\phi_a], \mathcal{S}[\phi_b])$	$p(\boldsymbol{\theta}_2 \mid \mathbb{A}, \mathcal{S}[\phi_a], \mathcal{S}[\phi_b])$
(init)	$\mathcal{D}ir(\boldsymbol{\alpha}_1 + (0,0,0,0,1,1))$	$\mathcal{D}ir(\boldsymbol{\alpha}_2 + (0,0,1,0,1,0))$
0	$\mathcal{D}ir(\boldsymbol{\alpha}_1 + (0,0,0,0,1,1))$	$\mathcal{D}ir(\boldsymbol{\alpha}_2 + (0,0,0,0,0,2))$
1	$\mathcal{D}ir(\boldsymbol{\alpha}_1 + (1,1,0,0,0,0))$	$\mathcal{D}ir(\boldsymbol{\alpha}_2 + (0,1,0,0,0,1))$
2	$\mathcal{D}ir(\boldsymbol{\alpha}_1 + (0,0,1,0,0,1))$	$\mathcal{D}ir(\boldsymbol{\alpha}_2 + (0,0,1,0,0,1))$
...	...	...

Here we denote by $\mathcal{D}ir(\boldsymbol{\alpha})$ a Dirichlet density parametrized by vector $\boldsymbol{\alpha}$, as per Equation (1)*. Algorithm 1 leverages the properties of conjugate priors to speed up its execution. Consider the first time a term satisfying ϕ_a is sampled during iteration $i = 0$ (line 7). At that time, $p(\boldsymbol{\theta}_1 \mid \mathbb{A}, \mathcal{S}[\phi_b])$ and $p(\boldsymbol{\theta}_2 \mid \mathbb{A}, \mathcal{S}[\phi_b])$ are two Dirichlet densities parametrized by vectors $(\boldsymbol{\alpha}_1 + (0,0,0,0,1,0))$ and $(\boldsymbol{\alpha}_2 + (0,0,0,0,1,0))$, respectively. Hence, $P(X_1[l_a] \mid \mathbb{A}, \mathcal{S}[\phi_b]))$ and $P(X_2[l_a] \mid \mathbb{A}, \mathcal{S}[\phi_b]))$ can be easily computed by Equation* (7)*. This, in turn, allows Algorithm 1 to sample from $P(\tau_c \mid \phi_c, \tau_{-c}, \mathbb{A})$ without explicitly marginalizing the random variables in $\mathcal{D}_{\phi_c} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2\}$. In the literature this optimization is often called "Rao-Blackwellisation" [20] or "variable collapsing" [89].*

In the following sections, we will present several optimization techniques to speed-up the execution of Algorithm 1. First, we will show how to impose certain syntactic restrictions on the formulation of ϕ, with the goal of speeding up the sampling step at line 7. With the same goal, we will also provide a technique to partition the set Sat $(\phi, \text{Var}(\phi))$ in a compact, efficient representation. Finally, we will show how to use an existing Datalog-like language [6] to encode large collections of propositional constraints into a compact theory of first-order logic sentences.

Before doing so, let's first introduce a more practical use case for our propositional language, reusing an example from [78].

Example 2 (Latent Dirichlet Allocation). *Latent Dirichlet Allocation (or LDA [13]) is a generative model for text. It represents a textual corpus as a discrete mixture of K "topics", where each topic is defined as a discrete probability distribution over words, and K is a user-provided parameter. Both the topics and the mixture terms are modeled as categorical distributions with Dirichlet priors. The model is trained in an unsupervised fashion, by conditioning the generative process with respect to the observed textual data. A formal definition of LDA is provided in [13]; here we will focus on representing it using our Bayesian dice.*

In our propositional language, LDA can be described as a game that involves the use of two kinds of Bayesian dice: the first kind, which we label as "blue", has one face for each word in the dictionary of the corpus; the second kind, that we label as "red", has K faces, one for each topic that we aim to extract. To play the game we instantiate one "red" die R_d for each document d in the corpus, and one "blue" die B_t for each topic t. We then represent the generative process $\mathcal{G}_{lda}$ as follows: for each word $w_{d,p}$ that appears inside document d at position p, we postulate that $w_{d,p}$ was generated by first throwing the red die R_d, observing some topic-identifier t as the result of this throw, then selecting the corresponding "blue" die B_t, rolling it, and observing word $w_{d,p}$ as a result. This stochastic process is described by the following set of Boolean constraints

$$\Phi_{lda} \stackrel{\text{def}}{=} \{\bigvee_{t=1}^{K} ((R_d[p] = t) \wedge (B_t[(d,p)] = w))\}_{d,p} \tag{16}$$

Notice that Φ_{lda} contains one constraint for each document d and position p, i.e. one constraint for each and every word in the corpus. Since each constraint admits K solutions, the size of $\textsc{Sat}(\Phi_{lda}, \mathbb{X})$ grows exponentially with the size of the corpus. This justifies the use of approximate inference. By running Algorithm 1 we can approximate the posterior distribution $p(\mathcal{D}_{\Phi_{lda}} \mid \Phi_{lda}, \mathbb{A})$: the posterior of the "blue" dice will determine the content of the topics, while the posterior of the "red" dice will determine the allocation of the topics over the documents.

2.3 Knowledge Compilation for Efficient Sampling

In this section we tackle the problem of sampling a satisfying assignment τ for a given Boolean constraint ϕ; we will reuse existing results from [118] to show that, under certain syntactic restrictions on ϕ, we can perform this task very efficiently. For simplicity, we will focus on the problem of sampling a term expression from $\textsc{Sat}(\phi, \textsc{Var}(\phi))$ w.r.t. the distribution $P(\tau \mid \phi, \mathbb{A})$. Nonetheless, all our conclusions will apply seamlessly to the more general problem of sampling from distribution $P(\tau_c \mid \phi, \tau_{-c}, \mathbb{A})$, as required by our Gibbs sampler (see line 7 of Algorithm 1).

Before proceeding, we need to extend our toolbox of boolean operators. We denote by $\odot$ and $\otimes$ two Boolean operators that compute the logical conjunction and disjunction, respectively, of two independent expressions. We call these two operators "independent conjunction" and "independent disjunction". We denote by $\oplus$ a Boolean operator that computes the logical disjunction of two expressions that are mutually exclusive. Two expressions are mutually exclusive when there is no assignment that satisfies both. We call the $\oplus$ operator a "deterministic disjunction" (not to be confused with the classic XOR operator). For any two expressions ϕ_1 and ϕ_2 that are independent, it is easy to prove that $P(\phi_1 \odot \phi_2 \mid \mathbb{A}) = P(\phi_1 \mid \mathbb{A}) \cdot P(\phi_2 \mid \mathbb{A})$. Similarly, $P(\phi_1 \otimes \phi_2 \mid \mathbb{A}) = (1 - (P(\neg\phi_1 \mid \mathbb{A}) \cdot P(\neg\phi_2 \mid \mathbb{A})))$. Furthermore, if ϕ_1 and ϕ_2 are two arbitrary expressions that are mutually exclusive, then $P(\phi_1 \oplus \phi_2 \mid \mathbb{A}) = P(\phi_1 \mid \mathbb{A}) + P(\phi_2 \mid \mathbb{A})$. If a constraint is read-once, then it can be expressed using only operators $\odot$, $\otimes$ and $\neg$. If a constraint is represented using only operators $\odot$, $\otimes$, $\oplus$ and $\neg$, then its likelihood can be computed in linear time w.r.t. the size of the

representation (the problem is #P-hard for unrestricted expressions). We can always represent any arbitrary constraint using exclusively the operators $\odot$, $\otimes$, $\oplus$ and $\neg$; nonetheless, such representation may end up being exponentially larger than one that uses only the operators $\wedge$, $\vee$ and $\neg$.

As shown in Example 2, many real-world probabilistic programs consist of large collections of constraints that all share the same structure. To account for this, we introduce the concept of *expression template*. An expression template can be seen as a regular Boolean expression where the literals are replaced with place-holders, that are labeled with the symbol $\langle\cdot\rangle$. For example, the expression

$$(((X_1[l_1] = 1) \odot (X_2[l_1] = 1)) \oplus ((X_1[l_1] = 2) \odot (X_2[l_1] = 2)))$$

has template $(\langle\cdot\rangle \odot \langle\cdot\rangle) \oplus (\langle\cdot\rangle \odot \langle\cdot\rangle)$. Without lack of generality, we will assume that all templates are in negated normal form (hence the $\neg$ operator is never used outside of a literal). Starting from a template, we can build new expressions by replacing the place-holders with literals of our choice, as long as we respect the restrictions imposed by the $\odot$, $\otimes$ and $\oplus$ operators. If a Boolean constraint ϕ is obtained from a template $\mathcal{T}$ we say that ϕ is a *ground expression* [21] for template $\mathcal{T}$. Clearly, the same template can be grounded into a large number of constraints; this fact will play an important role in the implementation of our language.

To enable the use of recursion in our probabilistic programs, we now introduce the concept of *recursive template*. A recursive template is a template where certain place-holders represent a sub-expression, rather than a simple literal. These are called *recursive place-holders* and are denoted with the symbol $\langle\star\rangle$. A recursive place-holder can be grounded into either the value $\top$ (true), $\bot$ (false) or into some arbitrary ground expression of the very same template. A recursive template may contain multiple recursive place-holders, and any two distinct recursive place-holders may be grounded into the same expression, if needed, as long as this does not violate any of the restrictions imposed by the $\odot$, $\otimes$ and $\oplus$ operators.

Following the approach of [27], we will represent the Boolean constraints generated by our templates as directed acyclic graphs (DAGs). Let $\mathcal{V}$ and $\mathcal{E}$ denote the set of vertices and edges, respectively, in a DAG. Each vertex in $\mathcal{V}$ will be labeled as either a literal (i.e. an atomic constraint or its negation), a constant ($\top$ or $\bot$), or a Boolean operator ($\wedge$,$\vee$,$\odot$,$\otimes$ or $\oplus$). Whenever the elements of $\mathcal{E} \subseteq \mathcal{V}^2$ can be composed into a directed path from vertex v_a to vertex v_b, we will say that v_a is an *ancestor* to v_b, and that v_b is a *descendant* to v_a. By definition $\mathcal{E}$ cannot contain any directed cycle. We define a *sink* as a vertex with no descendants, and a *root* as a vertex with no ancestors. In our DAGs, we only allow the presence of exactly one root; such node must be an ancestor of all the other vertices. Furthermore, every sink vertex must be either a literal or a constant and every literal/constant must be a sink. Figure 2 depicts a DAG generated by the probabilistic program for Hidden Markov Models that we introduce later in Section 3.6. The DAG includes six instances of the recursive template $\langle\cdot\rangle \odot ((\langle\cdot\rangle \odot \langle\star\rangle) \oplus (\langle\cdot\rangle \odot \langle\star\rangle))$.

We say that a DAG is *read-once* if it is negation normal form and only uses the operators $\odot$ and $\otimes$. By construction, all read-once DAGs have the topology of a tree. Following the terminology of [27], we say that a DAG is *decomposable* if it is in negation normal form and all the conjunctions in it are between expressions that are independent (hence a decomposable DAG always uses the $\odot$ operator in place of $\wedge$). We say that a DAG is *deterministic* if it is in negation normal form and all the disjunctions in it are between expressions that are mutually exclusive (hence a deterministic DAG always uses the $\oplus$ operator in place of $\vee$). Finally, we say that a DAG is in *deterministic decomposable negation normal form* (d-DNNF) if it satisfies both decomposability and determinism.

With the goal of speeding up our probabilistic programs, here we will identify certain classes of DAGs, for which the operation of sampling a satisfying assignment takes linear time in the size of the DAG. The size of a DAG is defined as the total number of vertexes and edges in it.

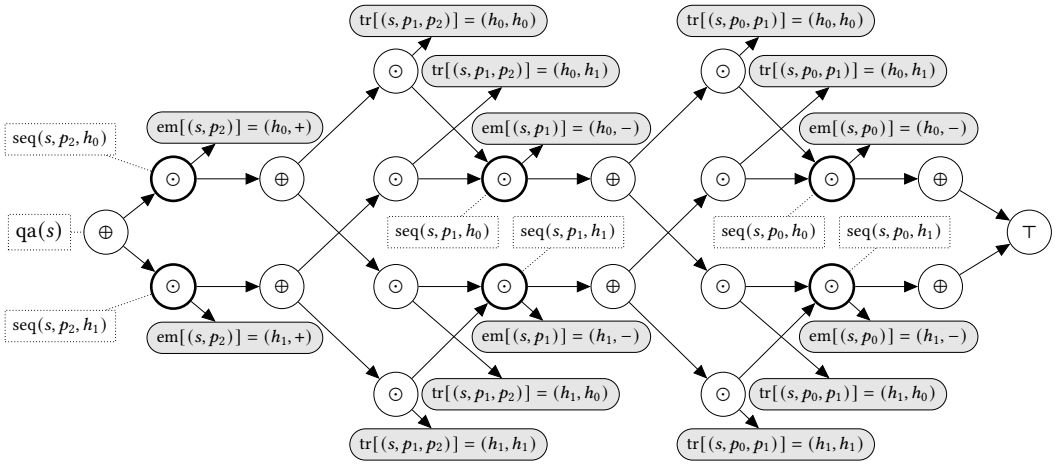


Fig. 2. A DAG generated by the probabilistic program for Hidden Markov Models (HMMs) presented in Section 3.6. The underlying HMM has two internal states (labeled as h_0 and h_1) and two observable symbols (+ and −). The DAG encodes the generative process associated with a single observable sequence (s) having three symbols, at positions p_0, p_1 and p_2. The observed sequence is "− − +". The DAG contains six instances of the recursive template $\langle\cdot\rangle \odot ((\langle\cdot\rangle \odot \langle\star\rangle) \oplus (\langle\cdot\rangle \odot \langle\star\rangle))$. For readability, each instance of the template is highlighted with a $\odot$-vertex with a bold edge, labeled with the Datalog ground predicate ($\text{seq}(\cdot,\cdot,\cdot)$) that generates it.

Proposition 1. *If ϕ is a read-once DAG, then sampling a satisfying assignment τ from $\textsc{Sat}(\phi, \textsc{Var}(\phi))$ takes linear time in the size of ϕ.*

Proposition 1 has been proven in [78], with the introduction of algorithm `SampleReadOnceSat`; in the interest of conciseness we will not report the same results here; we will just make use of algorithm SampleReadOnceSat when needed.

Proposition 2 (from [118]). *If ϕ is a* d-DNNF *DAG, then sampling a satisfying assignment τ from $\textsc{Sat}(\phi, \textsc{Var}(\phi))$ takes linear time in the size of ϕ.*

Proof. Proposition 2 has been proved in [118] for uniform distributions. Here we slightly adapt the result to our needs, which involve the use of non-uniform distributions, and contextualize it to our notation. Algorithm 2 shows how to sample a satisfying assignment for any arbitrary d-DNNF DAG ϕ^*. The algorithm requires the DAG to have each vertex $\psi \in \mathcal{V}$ annotated with the likelihood $P(\psi \mid \mathbb{A})$. These annotations can be obtained in linear time, by visiting the DAG in reverse topological order. The algorithm then proceeds to visit the DAG from the root to the leaves, iteratively composing a satisfying assignment from $\textsc{Sat}(\phi^*, \mathbb{X})$. Doing so for predicate- and $\odot$-nodes is trivial; in the case of $\oplus$-nodes the algorithm selects at random exactly one of the immediate descendants of the node, and proceeds sampling from it, ignoring the remaining immediate descendants. Notice that distribution $P(\psi_i \mid \phi^*, \mathbb{A})$ at line 12 can be computed in linear time, since all the elements in $\{\psi_i\}_i$ are pairwise mutually exclusive. □

Definition 2. *We say that a DAG ϕ is* almost-read-once *(ARO) if the following conditions are met:*

(1) ϕ is in negation normal form.
(2) ϕ only uses the operators $\odot$, $\otimes$ and $\oplus$.
(3) There is no directed path in ϕ that starts from a $\otimes$ vertex and ends in a $\oplus$ vertex.

Corollary 1. *If ϕ is an almost-read-once DAG, then sampling a satisfying assignment τ from $\textsc{Sat}(\phi, \textsc{Var}(\phi))$ takes linear time in the size of the DAG.*

Proof. The thesis follows immediately from the fact that any arbitrary ARO expression can always be represented as a d-DNNF expression whose literals consist of read-once expressions. To obtain a functional sampling algorithm, it is sufficient to extend Algorithm 2 and have it process any $\otimes$-node by invoking the method SampleReadOnceSat developed in [78]. □

2.4 Dynamic Boolean Expressions

Every Boolean expression ϕ can be interpreted as a constructive representation of the set $\textsc{Sat}(\phi, \textsc{Var}(\phi))$. In this section we will reuse some ideas from [78] to compactify such set, introducing the concept of *dynamic Boolean expressions*.

Before doing so, we will first introduce some formal notation to define expressions' rewritings. Let $X[l]$ be some arbitrary variable in $\textsc{Var}(\phi)$, and v some arbitrary value in its domain. We denote by $\phi||X[l] = v$ the expression obtained by replacing every atomic constraint in ϕ that mentions $X[l]$ with either the value $\bot$ (when $X[l] = v$ and the constraint are mutually exclusive) or the value $\top$ (otherwise). If τ is a term expression and $\textsc{Var}(\tau) \subseteq \textsc{Var}(\phi)$, we denote by $\phi||\tau$ the expression obtained by iteratively rewriting ϕ w.r.t. every literal in τ. We say that variable $X[l]$ is *inessential* in ϕ whenever for any two distinct values v_1 and v_2 in the domain of $X[l]$ the expressions $\phi||X[l] = v_1$ and $\phi||X[l] = v_2$ are logically equivalent. Otherwise, we say that $X[l]$ is *essential* in ϕ. If $X[l]$ is inessential in ϕ, then ϕ can be safely rewritten without mentioning $X[l]$, leading to a more compact representation of the set of its satisfying assignments.

A dynamic Boolean expression is a regular Boolean constraint that is defined over the union of two disjoint sets of variables, the *regular* variables $\mathbb{X} = \{X_1[l], \ldots, X_r[l]\}$ and the *volatile* variables $\mathbb{Y} = \{Y_1[l], \ldots, Y_s[l]\}$. Each volatile variable $Y_j[l]$ is annotated with an *activation condition* $\text{AC}(Y_j[l])$, a Boolean expression that may refer to any other variable in ϕ, either volatile or not. We say that a volatile variable $Y_j[l]$ is *active* whenever its activation condition is satisfied. Regular variables in $\mathbb{X}$ are considered to be always active. A dynamic expression ϕ is well-formed when it

Algorithm 2: SampleDdnnfSat (adapted from [118])

Input: A d-DNNF DAG ϕ^*, where each internal node $\psi \in \mathcal{V}$ is annotated with its likelihood $P[\psi \mid \mathbb{A}]$; a set of random variables $\mathbb{X}^*$ such that $\mathbb{X} \supseteq \mathbb{X}^* \supseteq \textsc{Var}(\phi^*)$.

Output: A term $\tau \in \textsc{Sat}(\phi^*, \mathbb{X}^*)$, sampled from $P[\tau|\phi^*, \mathbb{A}]$.

SampleDdnnfSat$(\phi^*, \mathbb{X}^*)$**:**
  if ϕ^* *is a literal* **then**
    Sample $\tau \in \textsc{Sat}(\phi^*, \mathbb{X}^*)$ from $P(\tau \mid \phi^*, \mathbb{A})$;
    return τ;
  else if $\phi^* = \oplus_{i=1}^{k}(\psi_k)$ **then**
    Compute $P[\psi_i \mid \phi^*, \mathbb{A}]$ for all $i \in \{1, .., k\}$;
    Sample $i \in \{1, .., k\}$ from $P[\psi_i \mid \phi^*, \mathbb{A}]$;
    return SampleDdnnfSat$(\psi_i, \mathbb{X}^*)$;
  else if $\phi^* = \odot_{i=1}^{k}(\psi_k)$ **then**
    for $i \in \{1, .., k\}$ **do**
      $\tau_i \leftarrow$ SampleDdnnfSat$(\psi_i, \textsc{Var}(\psi_i))$;
    Sample $\tau_{k+1} \in \textsc{Sat}\left(\top, \mathbb{X}^* \setminus \cup_{i=1}^{k}\textsc{Var}(\psi_i)\right)$ from $P[\tau \mid \mathbb{A}]$;
    return $\wedge_{i=1}^{k+1}\tau_i$;

satisfies the following two properties: (i) for every volatile variable $Y_j[l]$ and every assignment $\tau \in \textsc{Sat}\left(\neg\text{AC}(Y_j[l]), \textsc{Var}(\text{AC}(Y_j[l]))\right)$ that leaves it inactive, $Y_j[l]$ must be inessential w.r.t. expression $\phi||\tau$, (ii) if any volatile variable $Y_i[l]$ is essential in the activation expression of some other volatile variable $Y_j[l]$, then $\text{AC}(Y_j[l]) \models \text{AC}(Y_i[l])$ must hold true. As shown in [78], a well-formed dynamic Boolean expression induces a partition over the set of assignments that satisfy it. To model such partition, we define $\textsc{Dsat}(\phi)$ as the set of term-expressions $\{\tau_1, \ldots, \tau_m\}$ that satisfy the following properties:

(1) $\forall \tau \in \textsc{Dsat}(\phi) \quad \mathbb{X} \subseteq \textsc{Var}(\tau) \subseteq \mathbb{X} \cup \mathbb{Y}$
(2) $\forall \tau \in \textsc{Dsat}(\phi) \quad \tau \models \phi$
(3) $\forall \tau' \in \textsc{Sat}(\phi, \textsc{Var}(\phi)) \quad \exists \tau \in \textsc{Dsat}(\phi) \quad \tau' \models \tau$
(4) $\forall \tau \in \textsc{Dsat}(\phi) \quad$ if $Y_j[l] \in \textsc{Var}(\tau) \cap \mathbb{Y}$ then $\tau \models \text{AC}(Y_j[l])$
(5) $\forall \tau \in \textsc{Dsat}(\phi) \quad$ if $Y_j[l] \in \mathbb{Y} - \textsc{Var}(\tau)$ then $\tau \models \neg\text{AC}(Y_j[l])$

In simple terms, $\textsc{Dsat}(\phi)$ represents a set of term expressions that satisfy ϕ and use exclusively active variables; by construction the disjunction of all the terms in $\textsc{Dsat}(\phi)$ is logically equivalent to the disjunction of all the terms in $\textsc{Sat}(\phi, \textsc{Var}(\phi))$. In terms of probabilistic programming, one can think about the activation condition $\text{AC}(Y_j[l])$ as a simple probabilistic process that decides whether to roll Bayesian die Y_j one additional time (l) or not.

In order to use dynamic Boolean expressions in our probabilistic programs, we introduce a new variant of the $\oplus$ operator. Let ϕ be a dynamic Boolean expression and let $Y_j[l]$ be one of its volatile variables with activation condition $\text{AC}(Y_j[l])$. We will express ϕ as $\psi_1 \oplus^{\text{AC}(Y_j[l])} \psi_2$ whenever ψ_1 and ψ_2 are two expressions that are logically equivalent to $\phi \wedge \neg\text{AC}(Y_j[l])$ and $\phi \wedge \text{AC}(Y_j[l])$,

Algorithm 3: SampleDSat

Input: A dynamic almost-read-once DAG ϕ^*, with regular variables $\mathbb{X}^*$ and volatile variables $\mathbb{Y}^*$, where each internal node $\psi \in \mathcal{V}$ is annotated with its likelihood $P[\psi \mid \mathbb{A}]$.
Output: A term $\tau \in \textsc{Dsat}(\phi^*)$, sampled from $P[\tau|\phi^*, \mathbb{A}]$.

SampleDSat$(\phi^*, \mathbb{X}^*, \mathbb{Y}^*)$**:**
  if ϕ^* *is read-once* **then**
    return SampleReadOnceSat$(\phi^*, \mathbb{X}^*)$;
  else if $\phi^* = \psi_1 \oplus^{AC(Y_j[l])} \psi_2$ **then**
    $r \leftarrow$ RndUniform(0,1);
    if $r < (P[\psi_1|\mathbb{A}]/(P[\psi_1|\mathbb{A}] + P[\psi_2|\mathbb{A}]))$ **then**
      return SampleDSat$(\psi_1, \mathbb{X}^*, \mathbb{Y}^* - \{Y_j[l]\})$;
    else
      return SampleDSat$(\psi_2, \mathbb{X} \cup \{Y_j[l]\}, \mathbb{Y} - \{Y_j[l]\})$;
  else if $\phi^* = \oplus_{i=1}^{k}(\psi_k)$ **then**
    Compute $P[\psi_i \mid \phi^*, \mathbb{A}]$ for all $i \in \{1, .., k\}$;
    Sample $i \in \{1, .., k\}$ from $P[\psi_i \mid \phi^*, \mathbb{A}]$;
    return SampleDSat$(\psi_i, \mathbb{X}^*, \mathbb{Y}^*)$;
  else if $\phi^* = \odot_{i=1}^{k}(\psi_k)$ **then**
    for $i \in \{1, .., k\}$ **do**
      $\tau_i \leftarrow$ SampleDSat$(\psi_i, \mathbb{X}^* \cap \textsc{Var}(\psi_i), \mathbb{Y}^* \cap \textsc{Var}(\psi_i))$;
    Sample $\tau_{k+1} \in \textsc{Sat}\left(\top, \mathbb{X}^* \setminus \cup_{i=1}^{k}\textsc{Var}(\psi_i)\right)$ from $P[\tau \mid \mathbb{A}]$;
    return $\wedge_{i=1}^{k+1}\tau_i$;

respectively. Notice that, by construction, ψ_1 and ψ_2 are mutually exclusive and variable $Y_j[l]$ is essential in ψ_2 but inessential in ψ_1. Using the terminology from [27] we call $\oplus^{\text{AC}(Y_j[l])}$ a *decision node*, that determines whether volatile variable $Y_j[l]$ is active or not. As shown in [78], a well-formed dynamic Boolean expression always induces a natural evaluation order for all its activation conditions; our DAGs will respect such evaluation order and have one decision node for each volatile variable. Algorithm 3 shows how to sample a term from $\textsc{Dsat}\ (\phi^*)$ for any dynamic almost-read-once DAG ϕ^*, with regular variables $\mathbb{X}^*$ and volatile variables $\mathbb{Y}^*$, in linear time w.r.t. the size of ϕ^*. Notice that when $\mathbb{Y}^* = \emptyset$, the algorithm will simply sample from $\textsc{Sat}\ (\phi^*, \mathbb{X}^*)$.

Algorithm 3 concludes the formalization of our toy propositional language: as per Definition 1, a program consists of a (potentially large) collection of Boolean constraints, annotated with activation conditions as needed. To perform inference, we run Algorithm 1 (GibbsSatSampler); at each invocation of the sampling procedure at line 7, we execute Algorithm 3 (SampleDSat). While functional, this toy language remains impractical: a shown in Example 2, even a simple model may require us to specify a very large collection of Boolean constraints, all sharing some common structure, as in Equation (16). This redundancy is captured by the concept of template (both in the simple and in the recursive variants). In the next section we will introduce a new, alternative language to specify our probabilistic programs, based on Datalog; the new formalism will allow us to encode a large collection of Boolean constraints into concise first-order logic theories, leveraging the inherent structural redundancy captured by the templates.

3 QUERY-DRIVEN PROBABILISTIC PROGRAMMING

In this section we will repurpose a standard database query language to the goal of encoding a probabilistic program $(\mathcal{G}, \Phi)$, defined as per Definition 1. This will allow us to encode a large collection of propositional Boolean constraints (Φ) into a concise, lifted representation. This approach relies on two main ideas: (i) we model the generative process $\mathcal{G}$ as a probabilistic database [121] and (ii) we model the constraints in Φ as a collection of queries, that must return certain predetermined answers when run against $\mathcal{G}$. This approach is called "learning from query-answers" in [78–80] and "declarative probabilistic programming" in [5, 6]; the first line of research developed a probabilistic variant of Relational Algebra (γRA), used to encode the probabilistic programs in terms of relational constraints; the second introduced a probabilistic variant of Datalog (PPDL), with the same purpose. Before proceeding, we will compare strengths and weaknesses of these two approaches.

When it comes to representing a stochastic generative process (like $\mathcal{G}$), PPDL has more expressive power than γRA, as it supports both recursion and the use of arbitrary prior distributions (γRA is limited to distributions in the exponential family). Furthermore, PPDL introduces a well-defined chase procedure [2, 69] to model the (possibly infinite) instantiations of process $\mathcal{G}$, and derives from it a robust equivalence criterion for generative Datalog programs.

When it comes to conditioning the generative process with relational constraints (like those in Φ), γRA is equipped with (1) a well-founded mechanism to derive the posterior distribution of $\mathcal{G}$, (2) a functional inference algorithm to sample from it, (3) a procedure to update the model $\mathcal{G}$, if so desired; these components are entirely missing in PPDL.

In this paper we combine the two approaches: we identify a fragment of PPDL that is equivalent to the γRA language used in [78]; we extend it with recursion and adopt it as our query language of choice; we then design conditioning and inference methods for it. Our framework will inherit the elegant foundations of PPDL, while maintaining the extended capabilities and practicality of γRA. Following a common practice in the design of probabilistic databases, we will reinterpret the propositional Boolean constraints from Section 2 as a provenance mechanism [17, 48] for our newly adopted query language.

3.1 Data Model

We elect Gamma Probabilistic Databases (or γPDBs, [78]) as our data model of choice to represent $\mathcal{G}$. A γPDB is defined as a collection of probabilistic relation instances, called δ-tables, possibly paired with a set of deterministic tables. Here we will reformulate such model reusing the concept of Bayesian die.

Definition 3 (δ-tuple). *Let R be a deterministic relation instance of size d (i.e. R is a* set *of d distinct tuples) with schema $\mathcal{S}$. We define a δ-tuple as a d-dimensional Bayesian die where each of the d faces is annotated with, and uniquely identified by, one of the tuples from R.*

Definition 4 (δ-table). *Let R be a deterministic relation instance of size d with schema $\mathcal{S}$. We define a δ-table as a set of pairwise-independent δ-tuples that all range over the same domain R, under the assumption that schema $\mathcal{S}$ contains a key that uniquely identifies each die in the set.*

Definition 5 (γ-PDB). *We define a Gamma Probabilistic Database as a collection of δ-tables and regular, deterministic relations.*

For simplicity, and without lack of generality, we will assume the existence in each relation schema $\mathcal{S}$ of an attribute labeled `valID` that we can use to uniquely identify the faces of each δ-tuple. Similarly, we will assume the existence of an attribute labeled `varID` that we can use to uniquely identify each δ-tuple across the whole database instance.

3.2 Generative Datalog

To run queries against our γ-PDB $\mathcal{G}$ we use a fragment of GDatalog, a probabilistic variant of Datalog proposed in [6]. Let's start by defining formally the syntax and semantics of Datalog. A deterministic Datalog program is a first-order logic theory made of *facts* and *rules*. Facts represent knowledge about the real world and are expressed using ground predicates. For example, the ground predicate "parent(ada, bob)" may encode the knowledge that some individual named Ada is one of the parents of the individual named Bob. In the terminology of Datalog, "parent" is a *predicate* symbol, while "ada" and "bob" are *constant* symbols. A predicate is said to be ground when it only uses constant symbols, without any variables. A Datalog rule is a universally-quantified Horn clause, i.e. a disjunction of atoms having at most one positive, unnegated term (the clause's head). For example, the following Datalog rule states that "if individual A is parent to individual B *and* individual B is parent to individual C, *then* individual A must be a grand parent to individual B".

$$\text{grand-parent}(A, C) \leftarrow \text{parent}(A, B), \text{parent}(B, C).$$

In Datalog terminology, A, B and C are *variable* symbols, the atom "grandParent(A, C)" is the *head* of the clause, while the conjunction "parent(A, B), parent(B, C)" is its *body*. All variable symbols that are used in the head of a clause must appear in its body. Constant symbols may appear anywhere in a clause. Every Datalog theory admits a unique *least Herbrand model* [17], the set of all the ground predicates that can be derived from the initial set of facts, through the application of the clauses. The set of ground facts stored into a Datalog theory is often called its *extensional* theory; its rules form the *intensional* theory. Datalog is a database-friendly language: ground predicates can be easily stored into a relational database, by simply instantiating a relation instance (i.e. a table) for every predicate symbol. Furthermore, for recursion-free Datalog theories the least Herbrand model can be computed using standard relational algebra [21]. GDatalog [6] extends classic Datalog by introducing an element of uncertainty: instead of producing a set of certain ground facts, clauses are allowed to randomize the grounding of their head atom. For example, the following probabilistic clause can be interpreted as follows: if C is a city, and P is the likelihood of precipitation in that

city and T is a timestamp, then we should sample a value w from {sun, rain} w.r.t. a Categorical distribution parametrized by P and emit the resulting fact for predicate "weather".

$$\text{weather}(\underline{C, T}, w \in \{\text{sun}, \text{rain}\} \sim \mathcal{C}at[\![P]\!]) \leftarrow \text{city}(C, P), \text{ts}(T).$$

If we pair the above clause with the ground facts "city(fargo, [.1, .9])" and "ts(noon)", then the program will generate the ground fact "weather(fargo, noon, sun)" with probability .1, and the ground fact "weather(fargo, noon, rain)" with probability .9. The key constraint $\underline{C, T}$ on predicate "weather" ensures that only one value from {sun, rain} is sampled for any specific location and moment in time.

To represent a Gamma Probabilistic Database instance we will use only two probabilistic clauses, that are predetermined and will remain fixed in all the probabilistic programs. These two clauses are:

$$\begin{aligned} \text{obs}(\underline{\textit{VarId}, \textit{ObsId}}, v \in D \sim \mathcal{C}at[\![P]\!]) &\leftarrow \text{lp}(\textit{VarId}, D, P), \\ &\qquad \text{sample}(\textit{VarId}, \textit{ObsId}). \\ \text{lp}(\underline{\textit{VarId}}, D, p \in \mathcal{S}_{|D|} \sim \mathcal{D}ir[\![H]\!]) &\leftarrow \text{dt}(\textit{VarId}, D, H). \end{aligned}$$

Here $\mathcal{D}ir[\![H]\!]$ represents a Dirichlet density parametrized by vector H. No other probabilistic clause is allowed in our programs; the rest of the program must consist of regular Datalog clauses, where the use of predicates "dt", "sample", "obs" and "lp" is restricted as follows: (1) Predicate "dt" can only be used in ground facts (or deterministic derivations); it is used to instantiate δ-tuples and declare their domains (D) and hyper-parameters (H). Attribute *VarId* must act as a key for it. (2) Predicate "sample" is unrestricted, and can be used to draw samples from any δ-tuple. Notice that if the same grounding of predicate "sample" is derived multiple times, only one sample is generated. (3) Predicate "obs" can only be used within the body of a clause, to gather the result of a sampling operation. (4) Predicate "lp" cannot be used outside of the two fixed probabilistic clauses.

In conclusion, we will represent any Gamma Probabilistic Database instance $\mathcal{G}$ as a collection of ground facts for the predicate "dt" (having one such fact for each δ-tuple), together with the two probabilistic clauses listed above. The Boolean constraints (Φ) will be expressed using classic Datalog clauses and a provenance mechanism, as explained in the following section.

3.3 Provenance

We are now ready to equip our restricted fragment of GDatalog with a provenance/lineage [48, 49] mechanism. In classic Datalog, lineage expressions are used to link every ground fact in the least Herbrand model with all the facts in the extensional theory it can be derived from. For any fact t in the least Herbrand model, the lineage of t is a Boolean expression that uses the facts from the extensional theory as literals, and identifies all the subsets of the extensional theory that allows us to derive fact t, through the intensional theory. For example, let's consider the following Datalog theory:

$$\begin{aligned} &\text{works-for}(\text{ada}, \text{bob}). \\ &\text{works-for}(\text{ada}, \text{chloe}). \\ &\text{works-for}(\text{bob}, \text{zoe}). \\ &\text{works-for}(\text{chloe}, \text{zoe}). \\ &\text{works-for}(A, C) \leftarrow \text{works-for}(A, B), \text{works-for}(B, C). \end{aligned}$$

The lineage expression of the ground fact "works-for(ada,zoe)" is defined as follows:

$$(\text{works-for}(\text{ada}, \text{bob}) \wedge \text{works-for}(\text{bob}, \text{zoe}))$$
$$\vee$$
$$(\text{works-for}(\text{ada}, \text{chloe}) \wedge \text{works-for}(\text{chloe}, \text{zoe}))$$

In the context of GDatalog, each ground term of the predicate "obs" represents an atomic constraint over a Bayesian die, and multiple such constraints can be composed together into a lineage expression. These lineage expressions will represent the constraints (Φ) of our probabilistic program.

Let t be any ground fact that can be derived with non-zero probability, from now on we denote by $\Psi(t)$ its lineage expression. If t is a ground fact of any predicate other than "obs", that can be derived by joining m ground facts $\{t_1, .., t_m\}$, then we define the lineage of t as $(\wedge_{i=1}^{m} \Psi(t_i))$. Similarly, if t can be derived from *any* of the m ground facts $\{t_1, .., t_m\}$, then we define its lineage as $(\vee_{i=1}^{n} \Psi(t_i))$. We adopt special rules for ground instances of the predicate "obs". By construction, predicate "obs" can only be derived by one or multiple instances of predicate "sample". Let's denote by $\{t_1, \ldots, t_m\}$ these ground instance of "sample". We define the lineage of ground predicate "obs(var,obs,val)" as $(X_{\text{var}}[\text{obs}] = val) \wedge (\vee_{i=1}^{m}(\Psi(t_i)))$. Furthermore, we declare the expression $(\vee_{i=1}^{m}(\Psi(t_i)))$ as the activation condition for the constraint $(X_{\text{var}}[\text{obs}] = val)$.

Since a γ-PDB may contain deterministic relations, we assume that every deterministic extensional fact has lineage $\top$, that we interpret as a Boolean variable that is true with probability 1.0.

It is crucial to notice that whenever two distinct ground facts are inferred by the same first-order derivation, their lineage expressions will share the same structure (i.e. the same template). Thus, we can represent a large collection of constraints using concise first-order logic theories, and leverage such compact representation at compilation time.

3.4 Conditioning

Let process $\mathcal{G}$ be defined as a γ-PDB paired with a GDatalog theory (under the syntactic restrictions discussed above); in order to constrain $\mathcal{G}$, we select a subset C of the predicate symbols used by $\mathcal{G}$; we call C the set of "constrained predicates". We then define an additional deterministic Datalog program $\mathcal{K}$, that generates a set of ground instances for these predicates. Let's denote by RG_C the set of ground predicates generated by $\mathcal{K}$, and by PG_C the set of all ground instances of the constrained predicates that can be derived with non-zero probability from $\mathcal{G}$. The Boolean constraints of our probabilistic program are defined as follows

$$\Phi \stackrel{\text{def}}{=} \{\Psi(t)\}_{t \in (PG_C \cap RG_C)} \cup \{\neg\Psi(t)\}_{t \in (PG_C \setminus RG_C)} \tag{17}$$

Whenever $PG_C \subseteq RG_C$ holds true, we say that our program is *positively* constrained, and we simply mark the constrained predicates with an asterisk ($*$). We can conclude that a γ-PDB, paired with a (restricted) GDatalog theory and a conditioning Datalog program $\mathcal{K}$, fully defines a probabilistic program $(\mathcal{G}, \Phi)$, as per Definition 1.

3.5 Latent Dirichlet Allocation

We are now ready to give our probabilistic programming language a test drive. We start by showing how to encode the Latent Dirichlet Allocation (LDA) model from Example 2. We will assume that the training data consists of ground predicates in the form "d(DOCID, POS, WORDID)", where DOCID identifies a document in the corpus, POS identifies a specific position within the document and WORDID identifies the specific term used at position POS inside the document identified by DOCID. Furthermore, we use the predicate "t(TOPICID)" to declare unique identifiers for the topics

that we want to extract. A plausible extensional theory with two documents and two topics is provided below:

```
t(t1).      d(d0, p0, 'the').      d(d1, p0, 'the').
t(t2).      d(d0, p1, 'cat').      d(d1, p1, 'dog').
            d(d0, p2, 'naps').     d(d1, p2, 'barks').
```

The intensional theory instantiates two kinds of δ-tuples, one that ranges over the set of words ($ws \stackrel{\text{def}}{=} \{\text{barks, cat, dog, naps, the}\}$) and another that ranges over the set of topics ($ts \stackrel{\text{def}}{=} \{\text{t1}, \text{t2}\}$). They are used to model the "blue" and "red" dice from Example 2, respectively:

$$\begin{aligned}
\text{dt}([\text{red}, D], \text{ts}, [1, 1, .., 1]) &\leftarrow \text{d}(D, P, W).\\
\text{dt}([\text{blue}, T], \text{ws}, [1, 1, .., 1]) &\leftarrow \text{t}(T).\\
\text{sample}([\text{red}, D], P) &\leftarrow \text{d}(D, P, W).\\
\text{sample}([\text{blue}, T], [D, P]) &\leftarrow \text{d}(D, P, W), \text{obs}([\text{red}, D], P, T).\\
\text{qa}^*(D, P, W) &\leftarrow \text{d}(D, P, W), \text{obs}([\text{blue}, T], [D, P], W).
\end{aligned}$$

The first clause instantiates one "red" die for each document in the corpus; the second clause instantiates one "blue" die for each topic. The remaining three clauses encode the generative process $\mathcal{G}_{\text{lda}}$ described in Example 2: for each word in the corpus we throw the "red" die associated with the document where the word is located (clause 3); based on the outcome of this first throw, we choose one of the "blue" dice and roll it (clause 4); the word that we observe as the result of this second throw must match the initial word that we observed in the corpus (clause 5). The following mapping shows the lineage expressions generated by this program.

$$\begin{aligned}
&\forall d, p, w\\
&\Psi(\text{qa}(d, p, w)) \mapsto \oplus^{\text{AC}}_{t \in \text{ts}} ((\text{red}, d)[p] = t) \odot ((\text{blue}, t)[(d, p)] = w))
\end{aligned}$$

Here d denotes a document, p as position and w a word. In the above lineage expressions the literal $((\text{red}, d)[p] = t)$ acts as an activation condition for the literal $((\text{blue}, t)[(d, p)] = w)$. Notice that all the constraints are almost-read-once, as per Definition 2, and all share the very same template. Running Algorithm 1 against this probabilistic program is functionally equivalent to running the collapsed Gibbs sampler for LDA originally proposed by [50].

3.6 Hidden Markov Models

Hidden Markov Models (or HMMs [104]) are a popular probabilistic model for sequential data. They are based on the assumption that some observed sequences of symbols are generated by a latent, probabilistic finite state machine [129, 130]. The goal of the model is to learn the latent parameters of these machines from the data [8]. The model is formally defined by a set of internal states (H) and a set of observable symbols (V), paired with two probability distributions, one defined over $H \times H$ (the state-transition distribution) and one over $H \times V$ (the symbol-emission distribution). To express the model in our language, we assume that the training data is encoded as a collection of ground predicates in the form "d(SEQID, POS, SYMB)", where SEQID identifies a training sequence, POS is a positional index and SYMB represents the character observed at position POS of the sequence identified by SEQID. Furthermore, the predicate "eos(SEQID, POS)" marks the last position in a sequence. We implement the HMM model by declaring two δ-tuples, one to represent the state-transition probabilities (tr) and another for the symbol-emission probabilities

(em). The former will range over set trs $\stackrel{\text{def}}{=} H \times H$, the latter will range over set ems $\stackrel{\text{def}}{=} H \times V$.

$$
\begin{aligned}
\mathrm{dt}(\mathrm{tr}, \mathrm{trs}, [1, 1, .., 1]).\\
\mathrm{dt}(\mathrm{em}, \mathrm{ems}, [1, 1, .., 1]).\\
\mathrm{dd}(S, P_1, P_2, V_1, V_2) &\leftarrow \mathrm{d}(S, P_1, V_1), \mathrm{d}(S, P_2, V_2), P_2 = P_1 + 1.\\
\mathrm{sample}(\mathrm{tr}, O) &\leftarrow \mathrm{dd}(S, P_1, P_2, V_1, V_2), O = [S, P_1, P_2].\\
\mathrm{sample}(\mathrm{em}, O) &\leftarrow \mathrm{d}(S, P, V), O = [S, P].\\
\mathrm{seq}(S, 1, H_1) &\leftarrow \mathrm{dd}(S, 0, 1, V_0, V_1), \mathrm{obs}(\mathrm{tr}, [S, 0, 1], [H_0, H_1]),\\
&\qquad \mathrm{obs}(\mathrm{em}, [S, 0], [H_0, V_0]),\\
&\qquad \mathrm{obs}(\mathrm{em}, [S, 1], [H_1, V_1]).\\
\mathrm{seq}(S, P_2, H_2) &\leftarrow \mathrm{seq}(S, P_1, H_1), d(S, P_2, V_2), P_2 = P_1 + 1,\\
&\qquad \mathrm{obs}(\mathrm{tr}, [S, P_1, P_2], [H_1, H_2]),\\
&\qquad \mathrm{obs}(\mathrm{em}, [S, P_2], [H_2, V_2]).\\
\mathrm{qa}^*(S) &\leftarrow \mathrm{eos}(S, P), \mathrm{seq}(S, P, H).
\end{aligned}
$$

For the reader's convenience, the lineage expressions generated by this program are provided below:

$$
\begin{aligned}
&\forall s, p, h\\
&\Psi(\mathrm{seq}(s, 0, h)) \mapsto \left[\ \mathrm{em}[(s, 0)] = (h, v_{s,0})\right]\\
&\Psi(\mathrm{seq}(s, p, h)) \mapsto \left[\ \mathrm{em}[(s, p)] = (h, v_{s,p})\right] \wedge [\chi(s, p, h)]\\
&\chi(s, p, h) \mapsto [\oplus_{h' \in H} \Psi(\mathrm{seq}(s, p_-, h')) \wedge \mathrm{tr}[(s, p_-, p)] = (h', h)]\\
&\Psi(\mathrm{qa}(s)) \mapsto \left[\oplus_{h' \in H} \Psi(\mathrm{seq}(s, p_f, h'))\right]
\end{aligned}
$$

Here s identifies a training sequence, p identifies a position within sequence s, p_- denotes the position that precedes it, p_f the last position in sequence s, $v_{s,p}$ denotes the symbol at position p of sequence s, symbols h and h' denote internal states. Figure 2 shows one DAG generated by this probabilistic program, for a HMM with two internal states (h_0 and h_1) and two observable symbols (+ and −). The sequence that generates the DAG is − − +. Notice that the lineage expression is not almost-read-once (due to the use of the $\wedge$ operator). The Gibbs update can nonetheless be approximated with an almost-read-once DAG, consistently with the logic of blocked Gibbs samplers.

4 IMPLEMENTATION

We implemented StarfishDB as an extension of Apache Acero, an experimental query engine that is integral part of the Apache Arrow project. Acero is a relational, push-oriented [117] execution engine, that offers efficient implementations for the physical operators typically found in database systems (scan, projection, selection, join, group by, etc.). At the time of writing, Acero is mature enough to support TPC-H queries. Internally, Acero adopts Apache Arrow's representation format, which is column-oriented [15] and optimized for vectorized execution[16].

We extended Acero with an additional physical operator, called Sampler. This operator is responsible for the execution of the GibbsSatSampler method described in Algorithm 1. In a nutshell, we use Acero as a grounding engine to compute a large, propositional representation of a

probabilistic program; we then feed the data generated by Acero into the Sampler operator, that uses just-in-time compilation techniques [91] to run Algorithm 1 efficiently.

4.1 Just-in-time compilation for Gibbs Sampling

A lineage expression template, as defined in Section 2.3, can be seen as an abstract Boolean constraint, having a known structure (defined by the template itself) but unknown literals. Hence, we can use expression templates to provide a compact, lifted representation of a large collection of Boolean constraints, that all share the same common structure.

In our implementation expression templates are used to parametrize the Sampler operator. At initialization time, the Sampler operator compiles the given template into a static function, that implements the method SampleDSat defined in Algorithm 3. This function is tailored and optimized for the structure defined by the template. Each tuple generated by the grounding engine (Acero), and fed into the Sampler, represents a way to ground the template into a fully-defined Boolean constraint, something that our just-in-time compiled function can sample from. Thus, the sampling function simply reads the ground tuples one by one, and performs all the necessary steps to execute a Monte Carlo simulation. The sampling function is compiled only once, but it is executed many times, consistently with the logic of Algorithm 1. Thanks to recursive templates, we can represent complex Boolean DAGs using multiple ground records. If that is the case, the grounding engine will make sure to batch together all the records that refer to the same ground DAG. Recursive templates allow us to process very complex DAGs, without the risk of compiling equally complex sampling functions, that would likely saturate the instruction cache. Our implementation relies on the ClangJIT compiler [37] to perform just-in-time compilation. ClangJIT is an experimental fork of the LLVM project, that extends the clang compiler with the ability to compile C++ templates at runtime. Consistently, our expression templates are implemented as a collection of variadic C++ templates, one for each of the following Boolean operators: $\odot$, $\otimes$, $\oplus$, $\neg$ and $\langle\star\rangle$.

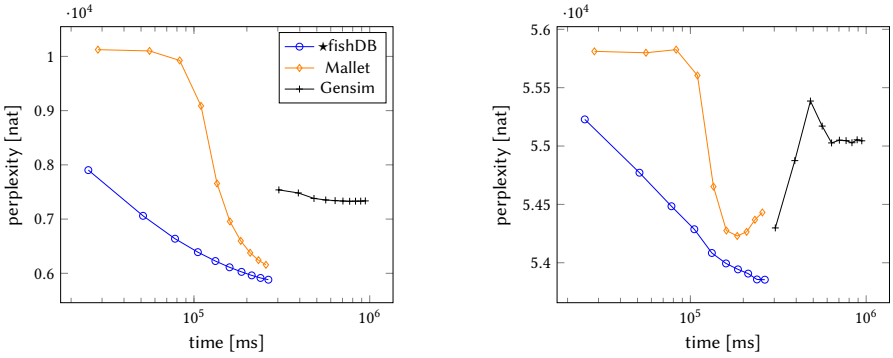
Example 3. *The table below represents the relation computed by Acero to ground the predicates "seq(s,p,h)" from the DAG shown in Figure 2. The DAG is obtained by compiling the recursive template* $\langle\cdot\rangle \odot ((\langle\cdot\rangle \odot \langle\star\rangle) \oplus (\langle\cdot\rangle \odot \langle\star\rangle))$ *once and then grounding it with each and every row in the relation below.*

rowID	seqID	pos	out	hs	ps0	ref0	ps1	ref1
r_0	s_0	0	−	h_0	NULL	NULL	NULL	NULL
r_1	s_0	0	−	h_1	NULL	NULL	NULL	NULL
r_2	s_0	1	−	h_0	h_0	r_0	h_1	r_1
r_3	s_0	1	−	h_1	h_0	r_0	h_1	r_1
r_4	s_0	2	+	h_0	h_0	r_2	h_1	r_3
r_5	s_0	2	+	h_1	h_0	r_2	h_1	r_3

The use of a push-oriented query-engine is justified by our observation that most grounding plans can be expressed with conjunctive queries. The idea of splitting a lineage expression over multiple ground tuples was inspired by [3].

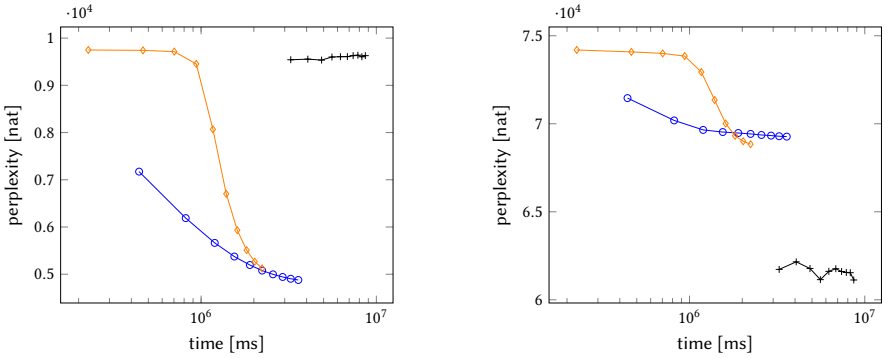
5 EXPERIMENTS

We verify claims (1) and (3) from Section 1 by compiling the LDA Datalog theory introduced in Section 3.5 with StarfishDB. The resulting inference algorithm generated by our compiler is functionally equivalent to the collapsed Gibbs sampler proposed in [50]. We choose to run LDA because the model makes strong structural assumptions about exchangeability, as thoroughly explained by Blei in [13]; therefore it is a natural match to our PP language, which follows the very same design philosophy about exchangeability. Furthermore, LDA is popular enough within the



(a) Train-set NYTIMES **(b)** Test-set NYTIMES

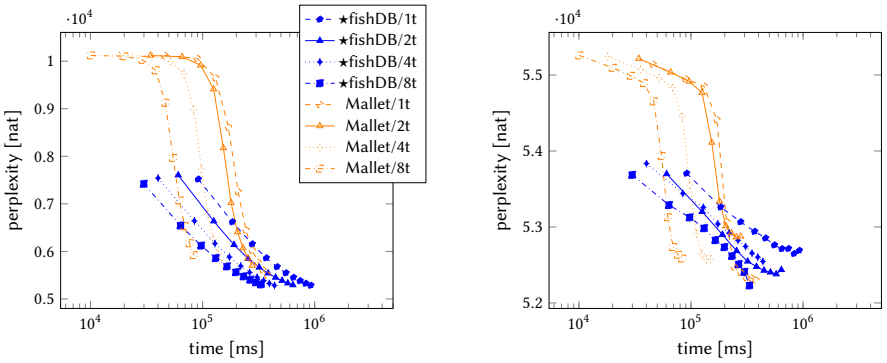
Fig. 3. LDA single threaded benchmarking on NYTIMES using 20 topics



(a) Train-Set PUBMED **(b)** Test-Set PUBMED

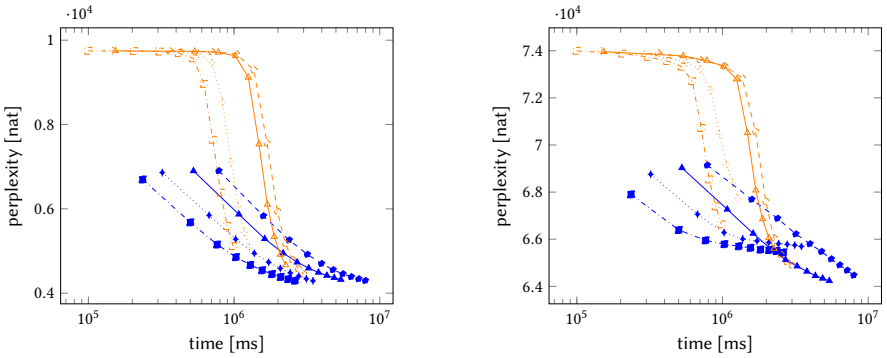
Fig. 4. LDA single threaded benchmarking on PUBMED using 20 topics

NLP community to have excellent implementations. One of these comes with the Mallet library [77], that offers a highly optimized implementation of the sampler proposed by [50]. Since both Mallet and our system run the very same sampler, the comparison is fair and informative. Furthermore, since we plan to extend StarfishDB with the support for variational inference, we also compare it against the Online Variational Bayes (OVB, [52]) method implemented in Gensim. The goal of this test is not to prove the superiority of any of the competing systems. It is simply to prove that (1) it is possible to compile a PPDL theory into a functional LDA sampler, and (2) the resulting code remains as practical as using an external library. We also notice that our LDA sampler can be implemented by simply writing five PPDL clauses; the development of tools like Mallet and Gensim requires much more effort and know-how. To compare the three implementations, we use *perplexity* as a proxy for the likelihood of the data [132], as it is common practice in NLP. In general terms, lower perplexity indicates a better fit of the model to the data. Nonetheless, the goal of our test is not to assess the quality of the generated topics, but just to compare the convergence speed of the three samplers. Our benchmark adopts two textual datasets, publicly available from the UCI [32] Machine Learning Repository: NYTIMES, and PUBMED. The NYTIMES dataset consists of 299,752 news articles, collectively containing about 100 million tokens, with a unique word count of



(a) Train-Set NYTIMES **(b)** Test-Set NYTIMES

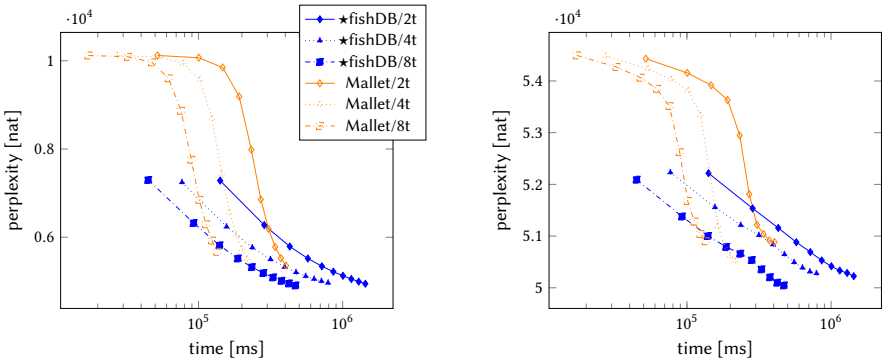
Fig. 5. LDA Multithreaded 50 topics benchmarking on NYTIMES using 1, 2, 4 and 8 threads



(a) Train-Set PUBMED **(b)** Test-Set PUBMED

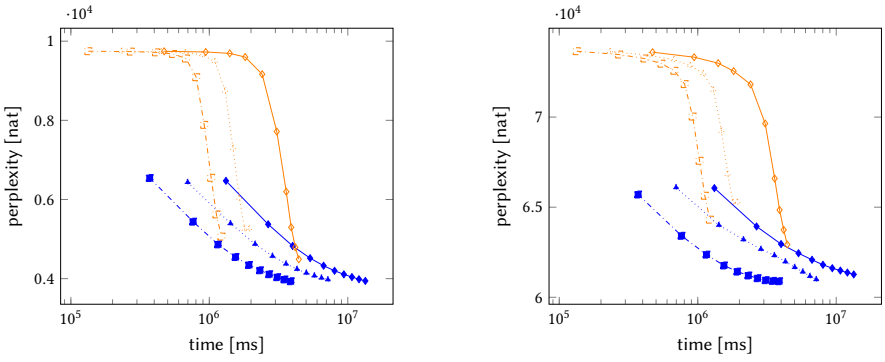
Fig. 6. LDA Multithreaded 50 topics benchmarking on PUBMED using 1, 2, 4 and 8 threads

102,660. PUBMED, the largest dataset, includes about 8.2 million research paper abstracts, totaling roughly 730 million tokens, drawn from a vocabulary of 141,043 distinct words. With each dataset, we earmarked 5 percent of the documents for testing, chosen at random, and used the remaining documents to train the model. The experiments are run on a 14-cores Xeon processor, operating at a base frequency of 2.20GHz. The system is equipped with 256 GBs of RAM and runs RedHat Linux, version 7.9. Each run of StarfishDB and Mallet consists of 20 iterations of the collapsed Gibbs sampler; each run of Gensim consists of 40 iterations of OVB. For all the three competing inference algorithms, we parametrize the priors of LDA in the same way: the symmetric Dirichlet prior over the document composition is set to 0.2, the symmetric Dirichlet prior over the topics' compositions is set to 0.1. Figures 3 and 4 reports the results of our test executed in single-threaded mode. The figures plot perplexity against execution time. The execution time is averaged over three runs, and excludes any time spent on I/O operations or grounding. Overall we observe that the performance of StarfishDB remains comparable with one of the other competitors. In the next batch of experiments (Figures 5 to 8), we assess the ability StarfishDB to automatically parallelize the operator Sampler. We compare it with the parallel implementation of LDA offered by Mallet. As described in [92, 135], this implementation leverages model-specific optimizations, that assume



(a) Train-Set NYTIMES **(b)** Test-Set NYTIMES

Fig. 7. LDA Multithreaded 100 topics benchmarking on NYTIMES using 2, 4 and 8 threads



(a) Train-Set PUBMED **(b)** Test-Set PUBMED

Fig. 8. LDA Multithreaded 100 topics benchmarking on PUBMED using 2, 4 and 8 threads

sparsity of topic-to-words assignments. StarfishDB, being designed as a general-purpose tool for probabilistic programming, does not implement any model-specific optimization. Nonetheless, it still manages to reach a comparable convergence time.

We verified claim (2) from Section 1 by compiling the HMM Datalog theory presented in Section 3.6. We tested the resulting inference algorithm against a simple synthetic classification problem with three labels. We used three pre-trained HMM models, with three internal symbols and three observable symbols each, to generate random sequences. Then we used StarfishDB to train three additional HMM models, sharing the same structural assumptions (3 internal states and 3 observable symbols), with the goal of associating each test sequence with the correct generating HMM. The classifier reached a near-perfect accuracy of 99.67%.

6 RELATED WORK

Our work combines contributions from many different fields; below we discuss some relevant works from the literature, comparing and contrasting with our approach.

Probabilistic Programming. Probabilistic programming [47, 124] has emerged as one of the leading approaches to Bayesian AI [42], with applications in many domains, including computer vision [65], computer graphics [109], natural language processing [108], data cleaning [68], cognitive [45] and environmental science [63], large-scale simulations [9, 10] and many others. Several research communities have independently developed competing PP systems, with major contributions from the fields of statistics [19, 85, 86, 99, 119, 123, 125], programming languages [4, 7, 23, 40, 53, 54, 88, 94, 97, 98, 111], AI [38, 61, 100, 107, 113], machine learning [44, 71, 82, 84, 112, 124] and, more recently, database systems [6, 78] . No single language has emerged as inherently superior to the others: each language excels in specific modeling scenarios, and offers a very unique trade-off of expressive power, inference speed, practicality and ease of use. We can broadly categorize PP languages by the programming paradigm they adopt, either imperative [11, 46, 75, 82, 84, 94, 98, 112], functional [33, 44, 71, 97, 119, 124] or logic-based [6, 61, 107, 113, 128], and the inference methods they support, either approximate or exact. General purpose languages tend to rely on approximate inference methods; these include likelihood weighting [83], slice sampling [101], Metropolis-Hastings [51, 81], Gibbs Sampling [41], Sequential Monte Carlo [133], Hamiltonian Monte Carlo [90], variational methods [12, 64]. Exact inference methods are much more robust in terms of accuracy and soundness, but are limited to models that admit either a tractable factorization or a closed symbolic representation. These methods include variable elimination [137], weighted model counting [22], Boolean circuits [53], lifted inference [29, 58], sum-product expressions [111], symbolic inference [40]. In the vast world of PP, StarfishDB can be characterized as a logic-based, relational language, whose syntax is restricted to discrete distributions with continuous conjugate priors; our language is targeted at Bayesian models with strong exchangeability assumptions, and uses collapsed Gibbs sampling for fast, approximate inference. StarfishDB relies heavily on compilation techniques to speed up inference; such approach has been successfully explored and validated by other languages [19, 131, 134].

Statistical Relational Learning. Statistical Relational Learning [106] is a branch of AI that studies the use of logic-based methods for probabilistic reasoning over relational structures, with applications in statistical machine learning. The field has introduced a number of innovative PP languages [6, 61, 105, 107, 113], that use restricted fragments of first-order logic (such as Horn clauses, datalog, prolog or answer-set programming) to express probabilistic models. Unlike the traditional systems, these languages are declarative in nature, and can represent large propositional programs in a very compact form, through the use of concise first-order theories [126]. StarfishDB belongs to this family, and inherits the same properties. PPDL [6] is the logic-based language that is closest to ours; similarities and differences between the two are discussed in detail in Section 3.

Knowledge Compilation. A recent trend in AI is to study probabilistic models that admit a tractable representation, with the goal of supporting exact inference. Efforts in this direction include frameworks like probabilistic decision graphs [55], and/or search spaces [30, 73], multi-valued decision diagrams [74], Boolean/arithmetic circuits [26], sum-product networks [102], probabilistic sentential decision diagrams [62]. Some of these methods have enough expressive power to encode deep neural networks [96], and have been used for a variety of AI tasks, including sampling [67, 118], equivalence testing [103], fairness assessment [116]; an excellent survey is provided in [24]. Many of these techniques have also found application into proper PP languages [53, 111], especially in the context of Statistical Relational Learning [36]. Good examples are ProbLog2 [31] and the Dice language [53], that use, respectively, compilation to d-DNNF expressions and binary decision diagrams (BDDs) to speed up inference. StarfishDB adopts similar strategies, but with a different goal: rather than using knowledge-compilation techniques to perform direct likelihood estimation, we use them to sample from conditional distributions and perform Gibbs sampling. This allows StarfishDB to support a broader class of *Bayesian* probabilistic models, that generalizes the tractable

ones. LDA provides a practical example of a Bayesian model for which no tractable exact inference is known, neither for parameter learning or simple likelihood estimation [132]. Thus, the use of Dirichlet priors makes StarfishDB more flexible when compared against languages like Dice, or any other system that depends on tractable, exact inference.

Database Systems. Integrating ML workloads into existing database management systems has been a long-standing goal for the data management research community [14, 34, 56, 115, 138]. This effort has been split between the development of systems that offer scalable relational engines for inference tasks [18, 39, 66, 87, 122, 139], that often use factorized representations [57, 59, 60, 93, 95], and systems that use AI methods to analyze the data they store [70, 110, 114, 136]. Research in database systems has also contributed to the field of Statistical Relational Learning, through the development of probabilistic databases [121].

7 CONCLUSIONS

We introduced a novel query execution engine to perform inference over Gamma Probabilistic programs. Unlike the original implementation from [78], the new engine supports recursion, advanced factorization techniques, and leverages just-in-time compilation. In the future we plan to investigate the ability of our probabilistic programming language to model and enforce relational fairness constraints [35].

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. 1979. The Theory of Joins in Relational Databases. *ACM Trans. Database Syst.* 4, 3 (1979), 297–314. https://doi.org/10.1145/320083.320091

[3] Lyublena Antova, Christoph Koch, and Dan Olteanu. 2009. $10^{(10^6)}$ worlds and beyond: efficient representation and processing of incomplete information. *VLDB J.* 18, 5 (2009), 1021–1040. https://doi.org/10.1007/s00778-009-0149-y

[4] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-symbolic inference for efficient streaming probabilistic programming. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1668–1696. https://doi.org/10.1145/3563347

[5] Vince Bárány, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2016. Declarative Probabilistic Programming with Datalog. In *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016 (LIPIcs, Vol. 48)*, Wim Martens and Thomas Zeume (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:19. https://doi.org/10.4230/LIPIcs.ICDT.2016.7

[6] Vince Bárány, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative Probabilistic Programming with Datalog. *ACM Trans. Database Syst.* 42, 4 (2017), 22:1–22:35. https://doi.org/10.1145/3132700

[7] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive probabilistic programming. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 898–912. https://doi.org/10.1145/3385412.3386009

[8] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. 1970. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The annals of mathematical statistics* 41, 1 (1970), 164–171.

[9] Atilim Günes Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip H. S. Torr, Victor W. Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Etalumis: bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, 29:1–29:24. https://doi.org/10.1145/3295500.3356180

[10] Atilim Gunes Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip H. S. Torr, Victor W. Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence

d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 5460–5473. https://proceedings.neurips.cc/paper/2019/hash/6d19c113404cee55b4036fce1a37c058-Abstract.html

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

[12] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. 2017. Variational inference: A review for statisticians. *Journal of the American statistical Association* 112, 518 (2017), 859–877.

[13] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (2003), 993–1022. http://jmlr.org/papers/v3/blei03a.html

[14] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00895ED1V01Y201901DTM057

[15] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85. https://doi.org/10.1145/1409360.1409380

[16] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[17] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 1973)*, Jan Van den Bussche and Victor Vianu (Eds.). Springer, 316–330. https://doi.org/10.1007/3-540-44503-X_20

[18] Zhuhua Cai, Zografoula Vagena, Luis Leopoldo Perez, Subramanian Arumugam, Peter J. Haas, and Christopher M. Jermaine. 2013. Simulation of database-valued markov chains using SimSQL. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 637–648. https://doi.org/10.1145/2463676.2465283

[19] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017).

[20] George Casella and Christian P Robert. 1996. Rao-Blackwellisation of sampling schemes. *Biometrika* 83, 1 (1996), 81–94.

[21] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

[22] Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.* 172, 6-7 (2008), 772–799. https://doi.org/10.1016/J.ARTINT.2007.11.002

[23] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact Recursive Probabilistic Programming. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 665–695. https://doi.org/10.1145/3586050

[24] Y Choi, Antonio Vergari, and Guy Van den Broeck. 2020. Probabilistic circuits: A unifying framework for tractable probabilistic models. http://starai.cs.ucla.edu/papers/ProbCirc20.pdf

[25] Nilesh N. Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *VLDB J.* 16, 4 (2007), 523–544. https://doi.org/10.1007/s00778-006-0004-3

[26] Adnan Darwiche. 2021. Tractable Boolean and Arithmetic Circuits. In *Neuro-Symbolic Artificial Intelligence: The State of the Art*, Pascal Hitzler and Md. Kamruzzaman Sarker (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 342. IOS Press, 146–172. https://doi.org/10.3233/FAIA210353

[27] Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *J. Artif. Intell. Res.* 17 (2002), 229–264. https://doi.org/10.1613/jair.989

[28] Bruno De Finetti. 1974. Theory of probability: a critical introductory treatment. (1974).

[29] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. 2005. Lifted First-Order Probabilistic Inference. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, Leslie Pack Kaelbling and Alessandro Saffiotti (Eds.). Professional Book Center, 1319–1325. http://ijcai.org/Proceedings/05/Papers/1548.pdf

[30] Rina Dechter and Robert Mateescu. 2007. AND/OR search spaces for graphical models. *Artif. Intell.* 171, 2-3 (2007), 73–106. https://doi.org/10.1016/J.ARTINT.2006.11.003

[31] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. 2015. ProbLog2: Probabilistic Logic Programming. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 9286)*, Albert Bifet, Michael May, Bianca Zadrozny, Ricard Gavaldà, Dino Pedreschi, Francesco Bonchi, Jaime S. Cardoso, and Myra Spiliopoulou (Eds.). Springer, 312–315. https://doi.org/10.1007/978-3-319-23461-8_37

[32] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml
[33] Martin Erwig and Steve Kollmansberger. 2006. Functional Pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.* 16, 1 (2006), 21–34. https://doi.org/10.1017/S0956796805005721
[34] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 755–768. https://doi.org/10.1145/3318464.3386137
[35] Golnoosh Farnadi, Behrouz Babaki, and Lise Getoor. 2019. A Declarative Approach to Fairness in Relational Domains. *IEEE Data Eng. Bull.* 42, 3 (2019), 36–48. http://sites.computer.org/debull/A19sept/p36.pdf
[36] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory Pract. Log. Program.* 15, 3 (2015), 358–401. https://doi.org/10.1017/S1471068414000076
[37] Hal Finkel, David Poliakoff, Jean-Sylvain Camier, and David F. Richards. 2019. ClangJIT: Enhancing C++ with Just-in-Time Compilation. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2019, Denver, CO, USA, November 22, 2019*. IEEE, 82–95. https://doi.org/10.1109/P3HPC49587.2019.00013
[38] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. 1999. Learning Probabilistic Relational Models. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, Thomas Dean (Ed.). Morgan Kaufmann, 1300–1309. http://ijcai.org/Proceedings/99-2/Papers/090.pdf
[39] Zekai J. Gao, Shangyu Luo, Luis Leopoldo Perez, and Chris Jermaine. 2017. The BUDS Language for Distributed Bayesian Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 961–976. https://doi.org/10.1145/3035918.3035937
[40] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4
[41] Stuart German and Donald German. 1988. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. In *Neurocomputing: foundations of research*. 611–634.
[42] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521, 7553 (2015), 452–459.
[43] Martin Charles Golumbic, Vladimir Gurvich, Yves Crama, and Peter L Hammer. 2011. Read-once functions. *Boolean Functions: Theory, Algorithms and Applications* (2011), 519–560.
[44] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, David A. McAllester and Petri Myllymäki (Eds.). AUAI Press, 220–229. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24
[45] Noah D Goodman, Joshua B. Tenenbaum, and The ProbMods Contributors. 2016. Probabilistic Models of Cognition. http://probmods.org/v2. Accessed: 2024-1-13.
[46] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio V. Russo, Johannes Borgström, and John Guiver. 2014. Tabular: a schema-driven probabilistic programming language. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 321–334. https://doi.org/10.1145/2535838.2535850
[47] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 167–181. https://doi.org/10.1145/2593882.2593900
[48] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.
[49] Todd J. Green and Val Tannen. 2006. Models for Incomplete and Probabilistic Information. *IEEE Data Eng. Bull.* 29, 1 (2006), 17–24. http://sites.computer.org/debull/A06mar/green.ps
[50] Thomas L Griffiths and Mark Steyvers. 2004. Finding scientific topics. *Proceedings of the National academy of Sciences* 101, suppl 1 (2004), 5228–5235.
[51] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. (1970).
[52] Matthew Hoffman, Francis Bach, and David Blei. 2010. Online learning for latent dirichlet allocation. *advances in neural information processing systems* 23 (2010).
[53] Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 140:1–140:31. https://doi.org/10.1145/3428208

[54] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 111–125. https://doi.org/10.1145/3062341.3062375

[55] Manfred Jaeger. 2004. Probabilistic Decision Graphs - Combining Verification And Ai Techniques For Probabilistic Inference. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* 12, Supplement-1 (2004), 19–42. https://doi.org/10.1142/S0218488504002564

[56] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. 2020. Declarative Recursive Computation on an RDBMS: or, Why You Should Use a Database For Distributed Machine Learning. *SIGMOD Rec.* 49, 1 (2020), 43–50. https://doi.org/10.1145/3422648.3422659

[57] David Justo, Shaoqing Yi, Lukas Stadler, Nadia Polikarpova, and Arun Kumar. 2021. Towards A Polyglot Framework for Factorized ML. *Proc. VLDB Endow.* 14, 12 (2021), 2918–2931. https://doi.org/10.14778/3476311.3476372

[58] Kristian Kersting. 2012. Lifted Probabilistic Inference.. In *ECAI*. 33–38.

[59] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Sebastian Schelter, Stephan Seufert, and Arun Kumar (Eds.). ACM, 8:1–8:10. https://doi.org/10.1145/3209889.3209896

[60] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Trans. Database Syst.* 45, 2 (2020), 7:1–7:66. https://doi.org/10.1145/3375661

[61] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory Pract. Log. Program.* 11, 2-3 (2011), 235–262. https://doi.org/10.1017/S1471068410000566

[62] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. 2014. Probabilistic Sentential Decision Diagrams. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter (Eds.). AAAI Press. http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/8005

[63] Christopher Krapu and Mark E. Borsuk. 2019. Probabilistic programming: A review for environmental modellers. *Environ. Model. Softw.* 114 (2019), 40–48. https://doi.org/10.1016/J.ENVSOFT.2019.01.014

[64] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2017. Automatic Differentiation Variational Inference. *J. Mach. Learn. Res.* 18 (2017), 14:1–14:45. http://jmlr.org/papers/v18/16-107.html

[65] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. 2015. Picture: A probabilistic programming language for scene perception. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 4390–4399. https://doi.org/10.1109/CVPR.2015.7299068

[66] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. 2021. Cerebro: A Layered Data Platform for Scalable Deep Learning. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper25.pdf

[67] Steven Lang, Martin Mundt, Fabrizio Ventola, Robert Peharz, and Kristian Kersting. 2021. Elevating Perceptual Sample Quality in PCs through Differentiable Sampling. In *NeurIPS 2021 Workshop on Pre-Registration in Machine Learning, 13 December 2021, Virtual (Proceedings of Machine Learning Research, Vol. 181)*, Samuel Albanie, João F. Henriques, Luca Bertinetto, Alex Hernández-García, Hazel Doughty, and Gül Varol (Eds.). PMLR, 1–25. https://proceedings.mlr.press/v181/lang22a.html

[68] Alexander K. Lew, Monica Agrawal, David A. Sontag, and Vikash Mansinghka. 2021. PClean: Bayesian Data Cleaning at Scale with Domain-Specific Probabilistic Programming. In *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 130)*, Arindam Banerjee and Kenji Fukumizu (Eds.). PMLR, 1927–1935. http://proceedings.mlr.press/v130/lew21a.html

[69] David Maier, Yehoshua Sagiv, and Mihalis Yannakakis. 1981. On the Complexity of Testing Implications of Functional and Join Dependencies. *J. ACM* 28, 4 (1981), 680–695. https://doi.org/10.1145/322276.322280

[70] Vikash Mansinghka, Ulrich Schaechthle, Zane Shelby, Harish Tella, Cameron Freer, Feras Saad, Joshua Thayer, Amanda Brower, Rachael Paiste, Jonathan Rees, et al. 2022. BayesDB For Data-Centric Scientific Discovery. (2022).

[71] Vikash Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099 (2014). arXiv:1404.0099 http://arxiv.org/abs/1404.0099

[72] Vikash Mansinghka, Richard Tibbetts, Jay Baxter, Patrick Shafto, and Baxter Eaves. 2015. BayesDB: A probabilistic programming system for querying the probable implications of data. *CoRR* abs/1512.05006 (2015). arXiv:1512.05006 http://arxiv.org/abs/1512.05006

[73] Radu Marinescu and Rina Dechter. 2009. AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artif. Intell.* 173, 16-17 (2009), 1457–1491. https://doi.org/10.1016/J.ARTINT.2009.07.003

[74] Robert Mateescu and Rina Dechter. 2007. AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Weighted Graphical Models. In *UAI 2007, Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence, Vancouver, BC, Canada, July 19-22, 2007*, Ronald Parr and Linda C. van der Gaag (Eds.). AUAI Press, 276–284. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1700&proceeding_id=23

[75] Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta (Eds.). Curran Associates, Inc., 1249–1257. http://papers.nips.cc/paper/3654-factorie-probabilistic-programming-via-imperatively-defined-factor-graphs

[76] Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit. *http://mallet. cs. umass. edu* (2002).

[77] Andrew Kachites McCallum. 2002. MALLET: A Machine Learning for Language Toolkit. (2002). http://www.cs.umass.edu/ mccallum/mallet.

[78] Niccolò Meneghetti and Ouael Ben Amara. 2022. Gamma Probabilistic Databases: Learning from Exchangeable Query-Answers. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang (Eds.). OpenProceedings.org, 2:260–2:273. https://doi.org/10.48786/edbt.2022.14

[79] Niccolò Meneghetti, Oliver Kennedy, and Wolfgang Gatterbauer. 2017. Beta Probabilistic Databases: A Scalable Approach to Belief Updating and Parameter Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 573–586. https://doi.org/10.1145/3035918.3064026

[80] Niccolò Meneghetti, Oliver Kennedy, and Wolfgang Gatterbauer. 2018. Learning From Query-Answers: A Scalable Approach to Belief Updating and Parameter Learning. *ACM Trans. Database Syst.* 43, 4 (2018), 17:1–17:41. https://doi.org/10.1145/3277503

[81] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21, 6 (1953), 1087–1092.

[82] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David A. Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, Leslie Pack Kaelbling and Alessandro Saffiotti (Eds.). Professional Book Center, 1352–1359. http://ijcai.org/Proceedings/05/Papers/1546.pdf

[83] Brian Milch, Bhaskara Marthi, David A. Sontag, Stuart Russell, Daniel L. Ong, and Andrey Kolobov. 2005. Approximate Inference for Infinite Contingent Bayesian Networks. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS 2005, Bridgetown, Barbados, January 6-8, 2005*, Robert G. Cowell and Zoubin Ghahramani (Eds.). Society for Artificial Intelligence and Statistics. http://www.gatsby.ucl.ac.uk/aistats/fullpapers/263.pdf

[84] Tom Minka. 2012. Infer. NET 2.5. *http://research. microsoft. com/infernet* (2012).

[85] Lawrence M. Murray. 2015. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. *Journal of Statistical Software* 67, 10 (2015), 1–36. https://doi.org/10.18637/jss.v067.i10

[86] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain (Proceedings of Machine Learning Research, Vol. 84)*, Amos J. Storkey and Fernando Pérez-Cruz (Eds.). PMLR, 1037–1046. http://proceedings.mlr.press/v84/murray18a.html

[87] Supun Nakandala and Arun Kumar. 2022. Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 506–520. https://doi.org/10.1145/3514221.3517846

[88] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9613)*, Oleg Kiselyov and Andy King (Eds.). Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

[89] Radford M Neal. 2000. Markov chain sampling methods for Dirichlet process mixture models. *Journal of computational and graphical statistics* 9, 2 (2000), 249–265.

[90] Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo* 2, 11 (2011), 2.

[91] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[92] David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. 2009. Distributed Algorithms for Topic Models. *J. Mach. Learn. Res.* 10 (dec 2009), 1801–1828.

[93] Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2017. In-Database Factorized Learning. In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017 (CEUR Workshop Proceedings, Vol. 1912)*, Juan L. Reutter and Divesh Srivastava (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-1912/paper21.pdf

[94] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, Carla E. Brodley and Peter Stone (Eds.). AAAI Press, 2476–2482. https://doi.org/10.1609/AAAI.V28I1.9060

[95] Dan Olteanu and Maximilian Schleich. 2016. F: Regression Models over Factorized Views. *Proc. VLDB Endow.* 9, 13 (2016), 1573–1576. https://doi.org/10.14778/3007263.3007312

[96] Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. 2020. Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 7563–7574. http://proceedings.mlr.press/v119/peharz20a.html

[97] Avi Pfeffer. 2001. IBAL: A Probabilistic Rational Programming Language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, Bernhard Nebel (Ed.). Morgan Kaufmann, 733–740.

[98] Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report* 137, 96 (2009), 4.

[99] Martyn Plummer et al. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, Vol. 124. Vienna, Austria, 1–10.

[100] David Poole. 2008. The Independent Choice Logic and Beyond. In *Probabilistic Inductive Logic Programming - Theory and Applications*, Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen H. Muggleton (Eds.). Lecture Notes in Computer Science, Vol. 4911. Springer, 222–243. https://doi.org/10.1007/978-3-540-78652-8_8

[101] Hoifung Poon and Pedro M. Domingos. 2006. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 458–463. http://www.aaai.org/Library/AAAI/2006/aaai06-073.php

[102] Hoifung Poon and Pedro M. Domingos. 2011. Sum-Product Networks: A New Deep Architecture. In *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, Fábio Gagliardi Cozman and Avi Pfeffer (Eds.). AUAI Press, 337–346. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2194&proceeding_id=27

[103] Yash Pote and Kuldeep S. Meel. 2021. Testing Probabilistic Circuits. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 22336–22347. https://proceedings.neurips.cc/paper/2021/hash/bc573864331a9e42e4511de6f678aa83-Abstract.html

[104] Lawrence R Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* 77, 2 (1989), 257–286.

[105] Luc De Raedt and Kristian Kersting. 2008. Probabilistic Inductive Logic Programming. In *Probabilistic Inductive Logic Programming - Theory and Applications*, Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen H. Muggleton (Eds.). Lecture Notes in Computer Science, Vol. 4911. Springer, 1–27. https://doi.org/10.1007/978-3-540-78652-8_1

[106] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00692ED1V01Y201601AIM032

[107] Matthew Richardson and Pedro M. Domingos. 2006. Markov logic networks. *Mach. Learn.* 62, 1-2 (2006), 107–136. https://doi.org/10.1007/s10994-006-5833-1

[108] Fabrizio Riguzzi, Evelina Lamma, Marco Alberti, Elena Bellodi, Riccardo Zese, and Giuseppe Cota. 2016. Probabilistic Logic Programming for Natural Language Processing. In *Proceedings of the AI*IA Workshop on Deep Understanding and Reasoning: A Challenge for Next-generation Intelligent Agents 2016 co-located with 15th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2016), Genova, Italy, November 28th, 2016 (CEUR Workshop*

Proceedings, Vol. 1802), Federico Chesani, Paola Mello, and Michela Milano (Eds.). CEUR-WS.org, 30–37. https://ceur-ws.org/Vol-1802/paper4.pdf

[109] Daniel Ritchie, Ben Mildenhall, Noah D. Goodman, and Pat Hanrahan. 2015. Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM Trans. Graph.* 34, 4 (2015), 105:1–105:11. https://doi.org/10.1145/2766895

[110] Feras Saad and Vikash K. Mansinghka. 2016. A Probabilistic Programming Approach To Probabilistic Data Analysis. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2011–2019. https://proceedings.neurips.cc/paper/2016/hash/46072631582fc240dd2674a7d063b040-Abstract.html

[111] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: probabilistic programming with fast exact symbolic inference. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 804–819. https://doi.org/10.1145/3453483.3454078

[112] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. (2016).

[113] Taisuke Sato and Yoshitaka Kameya. 1997. PRISM: A Language for Symbolic-Statistical Modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, 1330–1339. http://ijcai.org/Proceedings/97-2/Papers/078.pdf

[114] Ulrich Schaechtle, Cameron Freer, Zane Shelby, Feras Saad, and Vikash Mansinghka. 2022. Bayesian AutoML for databases via the InferenceQL probabilistic programming system. In *First Conference on Automated Machine Learning (Late-Breaking Workshop)*.

[115] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. Learning Models over Relational Data: A Brief Tutorial. In *Scalable Uncertainty Management - 13th International Conference, SUM 2019, Compiègne, France, December 16-18, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11940)*, Nahla Ben Amor, Benjamin Quost, and Martin Theobald (Eds.). Springer, 423–432. https://doi.org/10.1007/978-3-030-35514-2_32

[116] Nikil Roashan Selvam, Guy Van den Broeck, and YooJung Choi. 2023. Certifying Fairness of Probabilistic Circuits. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.). AAAI Press, 12278–12286. https://doi.org/10.1609/AAAI.V37I10.26447

[117] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *J. Funct. Program.* 28 (2018), e10. https://doi.org/10.1017/S0956796818000102

[118] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge Compilation meets Uniform Sampling. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018 (EPiC Series in Computing, Vol. 57)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.). EasyChair, 620–636. https://doi.org/10.29007/H4P9

[119] David J Spiegelhalter, Andrew Thomas, Nicky G Best, Wally Gilks, and D Lunn. 1996. BUGS: Bayesian inference using Gibbs sampling. *Version 0.5,(version ii) http://www. mrc-bsu. cam. ac. uk/bugs* 19 (1996).

[120] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. Probabilistic databases. *Synthesis lectures on data management* 3, 2 (2011), 1–180.

[121] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00362ED1V01Y201105DTM016

[122] Yuxin Tang, Zhimin Ding, Dimitrije Jankov, Binhang Yuan, Daniel Bourgeois, and Chris Jermaine. 2023. Auto-Differentiation of Relational Computations for Very Large Scale Machine Learning. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 33581–33598. https://proceedings.mlr.press/v202/tang23a.html

[123] Adrien Todeschini, Francois Caron, Marc Fuentes, Pierrick Legrand, and Pierre del Moral. 2014. Biips software: inference in Bayesian graphical models with sequential Monte Carlo methods. In *21st International Conference on Computational Statistics (COMPSTAT 2014)*. European Regional Section of the IASC, Genève, Switzerland. https://inria.hal.science/hal-01108399

[124] David Tolpin, Jan-Willem van de Meent, and Frank D. Wood. 2015. Probabilistic Programming in Anglican. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 9286)*, Albert Bifet, Michael May, Bianca Zadrozny, Ricard Gavaldà, Dino Pedreschi, Francesco Bonchi, Jaime S. Cardoso, and Myra Spiliopoulou (Eds.). Springer, 308–311. https://doi.org/10.1007/978-3-319-23461-8_36

[125] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja R. Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *CoRR* abs/1610.09787 (2016). arXiv:1610.09787 http://arxiv.org/abs/1610.09787

[126] Efthymia Tsamoura, Víctor Gutiérrez-Basulto, and Angelika Kimmig. 2020. Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 10284–10291. https://ojs.aaai.org/index.php/AAAI/article/view/6591

[127] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *stat* 1050 (2018), 27.

[128] Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.* 9, 3 (2009), 245–308. https://doi.org/10.1017/S1471068409003767

[129] Enrique Vidal, Franck Thollard, Colin De La Higuera, Francisco Casacuberta, and Rafael C Carrasco. 2005. Probabilistic finite-state machines-part I. *IEEE transactions on pattern analysis and machine intelligence* 27, 7 (2005), 1013–1025.

[130] Enrique Vidal, Frank Thollard, Colin De La Higuera, Francisco Casacuberta, and Rafael C Carrasco. 2005. Probabilistic finite-state machines-part II. *IEEE transactions on pattern analysis and machine intelligence* 27, 7 (2005), 1026–1039.

[131] Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From high-level inference algorithms to efficient code. *Proc. ACM Program. Lang.* 3, ICFP (2019), 98:1–98:30. https://doi.org/10.1145/3341702

[132] Hanna M Wallach, Iain Murray, Ruslan Salakhutdinov, and David Mimno. 2009. Evaluation methods for topic models. In *Proceedings of the 26th annual international conference on machine learning*. 1105–1112.

[133] Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014 (JMLR Workshop and Conference Proceedings, Vol. 33)*. JMLR.org, 1024–1032. http://proceedings.mlr.press/v33/wood14.html

[134] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodík. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, Subbarao Kambhampati (Ed.). IJCAI/AAAI Press, 3637–3645. http://www.ijcai.org/Abstract/16/512

[135] Limin Yao, David Mimno, and Andrew McCallum. 2009. Efficient Methods for Topic Model Inference on Streaming Document Collections. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 937–946. https://doi.org/10.1145/1557019.1557121

[136] Ce Zhang, Jaeho Shin, Christopher Ré, Michael J. Cafarella, and Feng Niu. 2016. Extracting Databases from Dark Data with DeepDive. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 847–859. https://doi.org/10.1145/2882903.2904442

[137] Nevin L Zhang and David Poole. 1994. A simple approach to Bayesian network computations. In *Proc. of the Tenth Canadian Conference on Artificial Intelligence*.

[138] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. *Proc. VLDB Endow.* 14, 10 (2021), 1769–1782. https://doi.org/10.14778/3467861.3467867

[139] Jia Zou, R. Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1189–1204. https://doi.org/10.1145/3183713.3196933

Received October 2023; revised January 2024; accepted March 2024