Approximate Lifted Inference in Probabilistic Databases

Wolfgang Gatterbauer
Carnegie Mellon University
gatt@cmu.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

This paper proposes a new approach for approximate evaluation of \#P-hard queries over probabilistic databases. In our approach, every query is evaluated entirely in the database engine by evaluating a fixed number of query plans, each providing an upper bound on the true probability, then taking their minimum. We provide an algorithm that takes into account important schema information to enumerate only the minimal necessary plans among all possible plans. Importantly, this algorithm is a strict generalization of all known results of \PTIME self-join free conjunctive queries: A query is safe if and only if our algorithm returns one single plan. We also apply three relational query optimization techniques to evaluate all minimal safe plans very fast. We give a detailed experimental evaluation of our approach and, in the process, provide a new way of thinking about the value of probabilistic methods over non-probabilistic methods for ranking query answers.

1. INTRODUCTION

Probabilistic inference over large data sets is becoming a central data management problem. Recent large knowledge bases, such as Yago [27], Nell [5], DeepDive [9], or Google’s Knowledge Vault [14], have millions to billions of uncertain tuples. Data sets with missing values are often “completed” using inference in graphical models [5][22] or sophisticated low rank matrix factorization techniques [15][51], which ultimately results in a large, probabilistic database. Data sets that use crowdsourcing are also uncertain [1]. And, very recently, probabilistic databases have been applied to bootstrapping over samples of data [53].

However, probabilistic inference is known to be \#P-hard in the size of the database, even for some very simple queries [7]. Today’s state of the art inference engines use either sampling based methods or are based on some variant of the DPLL algorithm for Weighted Model Counting. For example Tuffy [36], a popular implementation of Markov Logic Networks (MLN) over relational databases, uses Markov Chain Monte Carlo methods (MCMC). Gibbs sampling can be significantly improved by adapting some classical relational optimization techniques [56]. For another example, MayBMS [3] and its successor Sprout [39] use query plans to guide a DPLL-based algorithm for Weighted Model Counting [25]. While both approaches deploy some advanced relational optimization techniques, at their core they are based on general purpose probabilistic inference techniques, which either run in exponential time (DPLL-based algorithms have been proven recently to take exponential time even for queries computable in polynomial time [4]), or require many iterations until convergence.

In this paper, we propose a different approach to query evaluation on probabilistic databases. In our approach, every query is evaluated entirely in the database engine. Probability computation is done at query time, using simple arithmetic operations and aggregates. Thus, probabilistic inference is entirely reduced to a standard query evaluation problem with aggregates. There are no iterations and no exponential blowups. All benefits of relational engines (such as cost-based optimizations, multi-core query processing, shared-nothing parallelization) are immediately available to queries over probabilistic databases. To achieve this, we compute approximate rather than exact probabilities, with a one-sided guarantee: The probabilities are guaranteed to be upper bounds to the true probabilities, which we show is sufficient to rank the top query answers with high precision. Our approach consists of approximating the true query probability by evaluating a fixed number of “safe queries” (the number depends on the query), each providing an upper bound on the true probability, then taking their minimum.

We briefly review safe queries, which are queries whose data complexity is in \PTIME. They can be evaluated using safe query plans [7][17][23], which are related to a technique called lifted inference in the AI literature [12][23]: the entire computation is pushed inside the database engine and is thus efficient. For example, the query \( q_1(z) := R(z,x), S(x,y), K(x,y) \) has the safe query plan \( P_1 = \pi_z(R \cup y, \pi_x(S \cup y, K)) \), where every join operator multiplies the probabilities, and every projection with duplicate elimination treats probabilistic events as independent. The literature describes several classes of safe queries [6][17] and shows that they can be evaluated very efficiently. However, most queries are unsafe: They are provably \#P-hard and do not admit safe plans.

In this paper, we prove that every conjunctive query without self-joins can be approximated by a fixed number of safe queries, called “safe dissociations” of the original query. Every safe dissociation is guaranteed to return an upper bound on the true probability and can be evaluated in \PTIME data complexity. The number of safe dissociations depends only on the query and not the data. Moreover, we show how to find “minimal safe dissociations” which are sufficient to find the best approximation to the given query. For example, the unsafe query \( q_2(z) := R(z,x), S(x,y), T(y) \)
has two minimal safe dissociations, \( q_1(z) \) and \( q_2(z) \), where

\[
q_1(z) := R(z, x), S(x, y), T'(x, y)
\]

and

\[
q_2(z) := R'(z, x, y), S(x, y), T(y).
\]

Both queries are safe and, by setting the probability of every tuple \( R'(z, x, y) \) to that of \( R(z, x) \) and similarly for \( T' \), they return an upper bound for the probabilities of each answer tuple from \( q_2(z) \). One benefit of our approach is that, if the query happens to be safe, then it has a unique minimal safe dissociation, and our algorithm finds it.

**Contributions.** (1) We show that there exists a 1-to-1 correspondence between the safe dissociations of a self-join-free conjunctive query and its query plans. One simple consequence is that every query plan computes an upper bound of the true probability. For example, the two safe dissociations above correspond to the plans

\[
P_1^z = \pi_z (R \bowtie \pi_2 (\pi_1 (S \bowtie \pi_y T)))
\]

and

\[
P_2^z = \pi_z ((\pi_1 (R \bowtie S)) \bowtie \pi_y T).
\]

We give an intuitive systematic R-style algorithm [36] for enumerating all minimal safe dissociations of a query. Our algorithm takes into account important schema-level information: functional dependencies and whether a relation is deterministic or probabilistic. We prove that our algorithm has several desirable properties that the query is true in a randomly chosen world.

**Safe queries, safe plans.** It is known that the data complexity of any query \( q \) is either in PTIME or \#P-hard. The former are called safe queries and are characterized precisely by a syntactic property called hierarchical queries [7]. We briefly review these results:

**Definition 1 (Hierarchical query).** Query \( q \) is called hierarchical iff for any \( x, y \in \text{EVar}(q) \), one of the following three conditions hold: \( \text{at}(x) \subseteq \text{at}(y), \text{at}(x) \cap \text{at}(y) = \emptyset, \text{or } \text{at}(x) \supseteq \text{at}(y) \).

For example, the query \( q_1 := R(x, y), S(y, z), T(y, z, u) \) is hierarchical, while \( q_2 := R(x, y), S(y, z), T(z, u) \) is not, as neither of the three conditions holds for the variables \( y \) and \( z \).

**Theorem 2 (Dichotomy [7]).** If \( q \) is hierarchical, then \( \mathbb{P}(q) \) can be computed in PTIME in the size of \( D \). Otherwise, computing \( \mathbb{P}(q) \) is \#P-hard in the size of \( D \).

We next give an equivalent, recursive characterization of hierarchical queries, which we need for a few definitions. We write \( \text{SVar}(q) \) for the separator variables (or root variables); i.e., the set of existential variables that appear in every atom. \( q \) is disconnected if its atoms can be partitioned into two non-empty sets that do not share any existential variables (e.g., \( q := R(x, y) \), \( S(z, u) \), \( T(u, v) \) is disconnected and has two connected components: \( "R(x, y)" \) and \( "S(z, u), T(u, v)" \)). For every set of variables \( x \), denote \( q - x \) the query obtained by removing all variables \( x \) (and decreasing the arities of the relation symbols that contain variables from \( x \)).

**Lemma 3 (Hierarchical queries).** \( q \) is hierarchical iff either: (1) \( q \) has a single atom; (2) \( q \) has \( k \geq 2 \) connected components all of which are hierarchical; or (3) \( q \) has a separator variable \( x \) and \( q - x \) is hierarchical.

**Definition 4 (Query plan).** Let \( R_1, \ldots, R_m \) be a relational vocabulary. A query plan \( P \) is given by the grammar

\[
P := R_1(x) | \pi_a P | \exists [P_1, \ldots, P_k]
\]

where \( R_1(x) \) is a relational atom containing the variables \( x \) and constants, \( \pi_a \) is the project operator with duplicate elimination, and \( \exists [\ldots] \) is the natural join in prefix notation, which we allow to be \( \kappa \)-ary, for \( k \geq 2 \). We require that joins and projections alternate in a plan. We do not distinguish between join orders, i.e., \( \exists [P_1, P_2] \) is the same as \( \exists [P_2, P_1] \).

1 We assume w.l.o.g. that \( x \) is a tuple of only variables without constants.

2 Defined formally as \( \text{ADom}_i := \bigcup_{j \geq i} \text{SVar}(R_j) \).
We write $\mathbb{HVar}(P)$ for the head variables of $P$ (defined as the variables $X$ of the top-most projection $p_0$ (or the union of the top-most projections if the last operation is a join). Every plan $P$ represents a query $qp$ defined by taking all atoms mentioned in $P$ and setting $\mathbb{HVar}(qp) = \mathbb{HVar}(P)$. For notational convenience, we also use the “project-away” notation, by writing $\pi_{-q}(P)$ instead of $\mathbb{HVar}(P)$, where $y$ are the variables being projected away; i.e. $y = \mathbb{HVar}(P) - x$.

Given a probabilistic database $D$ and a plan $P$, each output tuple $t \in P(D)$ has a score($t$), defined inductively on the structure of $P$ as follows: If $t \in R_i(x)$, then score($t$) = $p(t)$, i.e. its probability in $D$; if $t \in \pi_{k_1}(P_1), \ldots, \pi_{k_l}(P_l)$ where $t = [t_1, \ldots, t_k]$, then score($t$) = $\prod_{i=1}^{k} \text{score}(t_i)$; and if $t \in \pi_{k_1}(P_1) \land \cdots \land \pi_{k_l}(P_l)$ are all the tuples that project into $t$, then score($t$) = $1 - \prod_{i=1}^{k} (1 - \text{score}(t_i))$.

In other words, score computes a probability by assuming that all tuples joined by $\land$ are independent, and all duplicates eliminated by $\pi$ are also independent. If these conditions hold, then score is the correct query probability, but in general the score is different from the probability. Therefore, score is not equal to the probability, in general, and is also called an extensional semantics [18, 41]. For a Boolean plan $P$, we get one single score, which we denote score($P$).

It is well known that if each variable $X$ has a probabilistic database then we interpret ev- er $q(X)$ has PTIME data complexity if it has a safe plan $P$: in that case the safe plan is unique, and $q(X) = \text{score}(P)$. For a Boolean plan $P$, each relation $R_i$ has a dissociation of $F$, a dissociated database instance: It adds some variables $\Delta R_i$ to every relation $R_i$ and a probabilistic database $D$. Let $\mathbb{P}(D)$ be two Boolean formulas with sets of variables $\mathbb{X}'$.

**Example 7 (Cont.).** Consider $q: R_i(x, y, z), \text{Adom}_{\text{y}_{1}}(y_{1}) \ldots \text{Adom}_{\text{y}_{m}}(y_{m})$. When $y_i = 0$ then we abbreviate $R_i$ with $R_i$. We give a simple example:

**Example 11 (Example 7 Cont.).** Consider $q: R_i(x, y, z)$, Then $\Delta = (y, 0)$ defines the following dissociation: $q_{\Delta} = \ldots$
R′(x,y), S(x,y), and the new relation R′ contains the tuples R′(1,4), R′(1,5), R′(2,4), R′(2,5). Notice that the lineage of the dissociated query q^A is F_{q,D} = R′(1,4), S(1,4) \lor R′(1,5), S(1,5) and is the same (up to variable renaming) as the dissociation of the lineage of query q: F = X′Y \lor X′′Z.

**Theorem 12 (Upper query bounds).** For every dissociation Δ of q: P(q) ≤ P(q^A).

**Proof.** Theorem 12 follows immediately from Theorem 8 by noting that the lineage F_{q,D} is a dissociation of the lineage F_{q,D} through the substitution \( \theta : D^A \rightarrow D \) defined as follows: for every tuple \( t^i \in R_{t}^\theta \), \( \theta(t^i) = \pi_x(t^i) \). □

**Definition 13 (Safe dissociation).** A dissociation Δ of a query q is called safe if the dissociated query q^A is safe.

By Theorem 12, a dissociation is safe (i.e., its probability can be evaluated in PTIME) iff q^A is hierarchical. Hence, amongst all dissociations, we are interested in those that are easy to evaluate and use them as a technique to approximate the probabilities of queries that are hard to compute. The idea is simple: Find a safe dissociation Δ, compute P(q^A), and thereby obtain an upper bound on P(q). In fact, we will consider all safe dissociations and take the minimum of their probabilities, since this gives an even better upper bound on P(q) than that given by a single dissociation. We call this quantity the propagation score3 of the query q.

**Definition 14 (Propagation).** The propagation score \( \rho(q) \) for a query q is the minimum score of all safe dissociations: \( \rho(q) = \min P(q^A) \) with Δ ranging over all safe dissociations.

The difficulty in computing \( \rho(q) \) is that the total number of dissociations is large, even for relatively small queries. If \( q \) has \( k \) existential variables and \( m \) atoms, then \( q \) has \( 2^k \cdot m \) possible dissociations with \( K = \sum_{i=1}^{k} (|\text{Var}(g_i)|) \) forming a partial order in the shape of a power set lattice (see Fig.1a for Example 17). Therefore, our next step is to prune the space of dissociations and examine only the minimum number necessary. We start by defining a partial order on dissociations:

**Definition 15 (Partial dissociation order).** We define the partial order on the dissociations of a query as:

\[ \Delta \preceq \Delta' \iff \forall y_i \subseteq y_i' \]

Whenever \( \Delta \preceq \Delta' \), then \( q^A, D^A \) is a dissociation of \( q^A, D^A \) (given by \( D^A = \Delta' - \Delta \)). Therefore, we obtain immediately:

**Corollary 16 (Partial dissociation order).** If \( \Delta \preceq \Delta' \) then \( P(q^A) \leq P(q^A) \).

**Example 17 (Partial dissociation order).** Consider the query \( q': -R(x,S(x),T(x,y),U(y)) \). It is unsafe and allows \( 2^3 = 8 \) dissociations which are shown in Fig. 1a with the help of an "augmented incidence matrix": each row represents one relation and each column one variable: An empty circle (⊙) indicates that a relation contains a variable; a full circle (●) indicates that a relation is dissociated on a variable (the reason for using two separate symbols becomes clear when we later include domain knowledge). Among those 8 dissociations, 5 are safe, shaded in green, and have the hierarchy among variables highlighted. Furthermore, 2 of the 5

The name comes from similarities with efficient belief propagation algorithms in graphical models. See [21] for a discussion on how query dissociation generalizes propagation algorithms from graphs to hypergraphs.

![Figure 1](image-url)
uses dissociated relations, hence each relation \( R^k_i(x, y) \) has external variables \( y \). Drop all variables \( y \) from the relations and all operators using them: This transforms \( \Delta^P \) into a regular, generally unsafe plan \( P \) for \( q \). For a trivial example, the plan corresponding to the top dissociation \( \Delta^i \) = \( \{ x \} \) of a query \( q \) is \( \pi \cdot \pi_{\{ x \}}(\Pi_x \{ R_1, \ldots, R_k \}) \); it performs all joins first, followed by all projections.

Conversely, consider any plan \( P \) for \( q \). We define its corresponding safe dissociation \( \Delta^P \) as follows. For each join operation \( \Join^P \{ R_1, \ldots, R_k \} \), let its join variables \( JVar \) be the union of the head variables of all subplans: \( JVar = \bigcup_{i} \pi_{\{ x \}}(R_i) \). For every relation \( R_i \) occurring in \( P_i \), add the missing variables \( HVar \) to \( JVar \). For example, consider \( \Join^P \{ R(x), T(x, y), U(y) \} \) (this is the lower join in query plan 5 of Fig. 1B). Here, \( JVar = \{ x, y \} \), and the corresponding safe dissociation of this subplan is \( q^i(x, y) := R(x, y) \cdot T(x, y), U^i(x, y) \). Note that while there is a one-to-one mapping between safe dissociations and query plans, unsafe dissociations do not correspond to plans.

**Theorem 18 (Safe dissociation).** Let \( q \) be a conjunctive query without self-joins. (1) The mappings \( \Delta \rightarrow \Delta^P \) and \( P \rightarrow \Delta^P \) are inverses of each other. (2) For every safe dissociation \( \Delta \), \( \mathbb{P}(q^\Delta) = \text{score}(P^\Delta) \).

**Corollary 19 (Upper bounds).** Let \( P \) be any plan for a Boolean query \( q \). Then \( \mathbb{P}(q) \leq \text{score}(P) \).

The proof follows immediately from \( \mathbb{P}(q) \leq \mathbb{P}(q^\Delta) \) (Theorem 12) and \( \mathbb{P}(q^\Delta) = \text{score}(P^\Delta) \) (Theorem 18). In other words, any plan for \( q \) computes a probability score that is guaranteed to be an upper bound on the correct probability \( \mathbb{P}(q) \).

**Theorem 18** suggests the following improved algorithm for computing the propagation score \( \rho(q) \) of a query: Iterate over all plans \( P, \) compute their scores, and retain the minimum score \( \text{min}_{P \in \text{score}(P)} \). Each plan \( P \) is evaluated directly on the original probabilistic database, and there need to materialize the dissociated database instance. However, this approach is still inefficient because it computes several plans that correspond to non-minimal dissociations. For example, in Example 17 the minimal plans are 3 and 4. The propagation score is thus the minimum of the scores of the two minimal plans: \( \rho(q) = \text{min}_{3 \leq \Delta \leq 4} \mathbb{P}(P) \). Our improved algorithm will iterate only over minimal plans, by relying on a connection between plans and sets of variables that disconnect a query: A cut-set is a set of external variables \( x \in \mathbb{E}(q) \) s.t. \( q \rightarrow x \) is disconnected.

**Algorithm 1**: generates all minimal query plans for a given query \( q \).

**Algorithm 1** computes the set of all minimal query plans.

**Conservativity.** Some probabilistic database systems first check if a query \( q \) is safe, and in that case compute the exact probability using the safe plan, otherwise use some approximation technique. We show that Algorithm 1 is conservative, in the sense that, if \( q \) is safe, then \( \rho(q) = \mathbb{P}(q) \). Indeed, in that case \( \text{MP}(q) \) returns a single plan, namely the safe \( P \) for \( q \), because the empty dissociation, \( \Delta = \{ \emptyset \} \), is safe, and it is the bottom of the dissociation lattice, making it the unique minimal safe dissociation.

**Score Quality.** We show here that the approximation of \( \mathbb{P}(q) \) by \( \rho(q) \) becomes tighter as the input probabilities in \( D \) decrease. Thus, the smaller the probabilities in the database, the closer does the ranking based on the propagation score approximate the ranking by the actual probabilities.

**Proposition 21 (Small Probabilities).** Given a query \( q \) and database \( D \). Consider the operation of scaling down the probabilities of all tuples in \( D \) with a factor \( f < 1 \). Then the relative error of approximation of \( \mathbb{P}(q) \) by the propagation score \( \rho(q) \) decreases as \( f \) goes to 0: \( \lim_{f \to 0} \frac{\mathbb{P}(q) - \rho(q)}{\mathbb{P}(q)} \to 0 \).

**Number of Dissociations.** While the number of minimal safe dissociations is exponential in the size of the query, recall that it is
3.3 Minimal plans with schema knowledge

Next, we show how knowledge of deterministic relations (i.e. all tuples have probability = 1), and functional dependencies can reduce the number of plans needed to calculate the propagation score.

3.3.1 Deterministic relations (DRs)

Notice that we can treat deterministic relations (DRs) just like probabilistic relations, and Corollary 19 with \( P(q) ≤ \text{score}(P) \) still holds for any plan \( P \). Just as before, our goal is to find a minimum number of plans that compute the minimal score of \( \Delta \) for each minimum equivalence class since \( \Delta \) is deterministic, then \( \Delta \rightarrow \Delta' \), and thus, \( \Delta \) is safe. We now explain two simple modifications to Algorithm 1 that achieve exactly our desired optimizations described above:

(1) Denote with \( \text{MinPCuts}(q) \) the set of minimal cut-sets that disconnect the query into at least two connected components with probabilistic tables. Replace \( \text{MinCuts}(q) \) in line 10 with \( \text{MinPCuts}(q) \).

(2) Denote with \( m_p \) the number of probabilistic relations in a query. Replace the stopping condition in line 1 with: if \( m_p ≤ 1 \) then \( \mathscr{P} ← \{ R_k \oplus \sum_{i} R_i \} \), where \( \oplus \) denotes a probabilistic relation. For example, for \( q \rightarrow R(x), S(x,y), T^d(y) \), \( \text{MinPCuts}(q) = \{(x), (y)\} \), while \( \text{MinCuts}(q) = \{\{x\}\} \). Therefore, the modified algorithm returns \( p^\Delta \) as single plan. For \( q \rightarrow R^d(x), S(x,y), T^d(y) \), the stopping condition is reached (also, \( \text{MinPCuts}(q) = \{\{x\}\} \) and the algorithm returns \( p^\Delta \) as single plan (see Fig. 3c).

Theorem 24 (Algorithm 1 with DRs). Algorithm 1 with above 2 modifications returns a minimum number of plans to calculate \( \rho(q) \) given schema knowledge about DRs.

For example, for \( q \rightarrow R(x), S(x,y), T^d(y) \), \( \text{MinPCuts}(q) = \{(x), (y)\} \), while \( \text{MinCuts}(q) = \{\{x\}\} \). Therefore, the modified algorithm returns \( p^\Delta \) as single plan. For \( q \rightarrow R^d(x), S(x,y), T^d(y) \), the stopping condition is reached (also, \( \text{MinPCuts}(q) = \{\{x\}\} \) and the algorithm returns \( p^\Delta \) as single plan (see Fig. 3c).

3.3.2 Functional dependencies (FDs)

Knowledge of functional dependencies (FDs), such as keys, can also restrict the number of necessary minimal plans. A well-known example is the query \( q \rightarrow R(x), S(x,y), T(y) \) from Example 23 becomes safe if we know that \( S \) satisfies the FD \( \Gamma \rightarrow x \rightarrow y \) and has a unique safe plan that corresponds to dissociation \( \Delta_2 \). In other words, we would like our modified algorithm to take \( \Gamma \) into account and to not return the plan corresponding to dissociation \( \Delta_1 \).
Let Ψ be the set of FDs on Var(q) consisting of the union of FDs on every atom Rᵢ in q. As usual, denote Ψ⁺ the closure of a set of attributes xᵢ, and denote Δ'= (y₁, ..., yₘ) the dissociation defined as follows: for every atom Rᵢ(yᵢ) in q, yᵢ = x⁺ᵢ \ \setminus \ \setminus \ !xᵢ. Then we show:

**Lemma 25 (Dissociation and FDs).** Dissociating a table Rᵢ on any variable \( yᵢ \in xᵢ \) does not change the probability.

This lemma is similar to **Lemma 22**. We can thus further refine our probabilistic dissociation preorder \( \preceq_p \) by:

\[
\Delta \preceq_p \Delta' \iff \forall i, Rᵢ \text{ probabilistic} : yᵢ \setminus \setminus \setminus \setminus \setminus xᵢ \subseteq yᵢ \setminus \setminus \setminus \setminus \setminus xᵢ^+
\]

As a consequence, using \( \preceq_p \) instead of \( \preceq \), allows us to further reduce the number of minimal safe equivalence classes. We next state a result by [39] in our notation:

**Proposition 26 (Safety and FDs [39, Prop. IV.5]).** A query q is safe iff \( q^{=\Psi} \) is hierarchical.

This justifies our third modification to **Algorithm 1** for computing \( p(q) \) of a query q over a database that satisfies \( \Gamma \): First compute \( \Delta \), then run \( q^{\Delta} \) on our previously modified **Algorithm 1**.

**Theorem 27 (Algorithm 1 with FDs).** **Algorithm 1** with above 3 modifications returns a minimum number of plans to calculate \( p(q) \) given schema knowledge about DRs and FDs.

It is easy to see that our modified algorithm returns one single plan iff the query is safe, taking into account its structure, DRs and FDs. It is thus a strict generalization of all known safe self-join-free conjunctive queries [7, 39]. In particular, we can reformulate the known safe query dichotomy [7] in our notation very succinctly:

**Corollary 28 (Dichotomy).** \( p(q) \) can be calculated in **PTIME** iff there exists a dissociation \( \Delta \) of q that is (i) hierarchical, and (ii) in an equivalence class with q under \( \preceq_p \).

To see what the corollary says, assume first that there are no FDs: Then q is in **PTIME** iff there exists a dissociation \( \Delta \) of the DRs only, such that \( p(q) = p(q^{\Delta}) \). If there are FDs, then we first compute the full dissociation \( \Delta \) (called “full chase” in [39]), then apply the same criterion to \( q^{\Delta} \).

4. **MULTI-QUERY OPTIMIZATIONS**

So far, **Algorithm 1** enumerates all minimal query plans. We then take the minimum score of those plans in order to calculate the propagation score \( p(q) \). In this section, we develop three optimizations that can considerably reduce the necessary calculations for evaluating all minimal query plans. Note that these three optimizations and the two optimizations from the previous section are orthogonal and can be arbitrarily combined in the obvious way. We use the following example to illustrate the first two optimizations.

**Example 29 (Optimizations).** Consider \( q = R(x, z), S(y, u), T(z), U(a), M(x, y, z, u) \). Our default is to evaluate all 6 minimal plans returned by **Algorithm 1** then take the minimum score (shown in Fig. 4a). Figure 4b and Fig. 4c illustrate the optimized evaluations after applying Opt. 1, or Opt. 1 and Opt. 2, respectively. ■

4.1 **Opt. 1: One single query plan**

Our first optimization creates one single query plan by pushing the min-operator down into the leaves. It thus avoids calculations when it is clear that other calculations must have lower bounds. The idea is simple: Instead of creating one query subplan for each top set \( y \in \text{MinCut}(q) \) in line 12 of **Algorithm 1**, the adapted **Algorithm 2** takes the minimum score over those top sets, for each tuple of the head variables in line 11. It thus creates one single query plan.

4.2 **Opt. 2: Re-using common subplans**

Our second optimization calculates only once, then re-uses common subplans shared between the minimal plans. Thus, whereas our first optimization reduces computation by combining plans at their roots, the second optimization stores and re-uses common results in the branches. The adapted **Algorithm 3** works as follows: It first traverses the whole single query plan (**FindingCommonSubPlan**) and remembers each subplan by the atoms used and its head variables in a HashSet [38, line 14]. If it sees a subplan twice (line 13), it creates a new view for this subplan, mapping the subplan to a new view definition. The actual plan (**ViewReusingPlan**) then uses these views whenever possible (line 18). The order in which the views are created (line 5) assures that the algorithm also discovers and exploits nested common subexpressions. Figure 4c illustrates for Example 29 that both the main plan and the view \( V_2 \) re-use views \( V_1 \) and \( V_2 \).

4.3 **Opt. 3: Deterministic semi-join reduction**

The most expensive operations in probabilistic query plans are the group-bys for the probabilistic project operations. These are often applied early in the plans to tuples which are later pruned and do not contribute to the final query result. Our third optimization is to first apply a full semi-join reduction on the input relations before starting the probabilistic evaluation from these reduced input relations. We like to draw here an important connection to [39], which introduces the idea of “lazy plans” and shows orders of magnitude performance improvements for safe plans by computing confidences not after each join and projection, but rather at the very end of the plan. We note that our semi-join reduction serves the same purpose with similar performance improvements and also apply for safe queries. The advantage of semi-join reductions, however, is that we do not require any modifications to the query engine.

5. **EXPERIMENTS**
Recursive algorithm: SP (SinglePlan)
Input: Query q(x): R₁(x₁), ..., Rₖ(xₖ)
Output: Single query plan P
1 if m = 1 then P = π₉R₈(x₈)
2 else
3 if q is disconnected then
4 Let q = q₁, ..., qₙ be the components connected by EVar(q)
5 Let HVar(q) = HVar(q) ∪ Var(q)
6 P = 0
7 if HVar(q) = ∅ then return
8 else
9 Let MinCuts(q) = (y₁, ..., yₙ)
10 Let q'_i = q_i with HVar(q'_i) = HVar(q) ∪ y_i
11 if j = 1 then P = π₉PₙSP(q'_j)
12 else P = min{π₉PₙSP(q'_j), ..., π₉PₙSP(q'_₁)}

Algorithm 2: Optimization 1 recursively pushes the min operator into the leaves and generates one single query plan.

Algorithm: UsingCommonSubplans
Input: Query q(x): R₁(x₁), ..., Rₖ(xₖ)
Output: Ordered set of view definitions Y, final query plan P
1 HS ← ∅ // HashSet of all subplans
2 HH ← (0, 0) // HashMap from subplans to unique view names
3 Y ← ∅ // Set of view definitions
4 FS(q)
5 foreach q_i ∈ HH keys in increasing size of HVar(q_i) and Var(q_i) do
6 if q_i is disconnected then
7 Let q = q₁, ..., qₙ be the components connected by EVar(q)
8 foreach q do FS(q(x_i))
9 else
10 if (m = 1 ∧ x₁ = x₈) ∨ HVar(q) ≠ ∅ then return
11 if q ∈ HH ∧ HH(q) ≠ ∅ then HS(q) ← HH(q) // new view name
12 HS = HS ∪ {q}
13 foreach y ∈ MinCuts(q) do
14 Let q' = q with HVar(q') = HVar(q) ∪ y
15 FS(q')
16 Recursive function: RP (ViewReusingPlan)
Input: Query q(x): R₁(x₁), ..., Rₖ(xₖ)
Output: Query plan P that reuses views from HashMap HS
17 if HS(q) ≠ ∅ then P = HS(q)
18 else
19 Insert here lines 1-11 from Algorithm 2 replacing SP with RP

Algorithm 3: Optimizations 1 & 2 together create a query plan which re-uses several previously defined temporary views.

We are interested in both the quality and the efficiency of dissociation as compared to exact probabilistic inference. Monte Carlo simulation (MC), and standard deterministic query evaluation (“deterministic SQL”). Our experiments, thus, investigate the following questions: How much can our three optimizations improve dissociation? How fast is dissociation compared to exact probabilistic inference, MC, and deterministic query evaluation? How good is the ranking from dissociation as compared to MC and ranking by lineage size? What are the most important parameters determining the ranking quality for each of the three methods?

Ranking quality. We use mean average precision (MAP) to evaluate the quality of ranking of a method by comparing it against the ranking from exact probabilistic inference as ground truth (GT). MAP rewards rankings that place items correctly; the best possible value is 1, and the worst possible 0 [34]. Average Precision at 10 (AP@10) is defined as AP@10 = \[\frac{1}{n} \sum_{k=1}^{n} P@k \times k\] where P@k is the precision at kth answer returned. Averaging over several rankings yields MAP. We use a variant of the analytic method proposed in [35] to calculate AP in the presence of ties. As baseline for no ranking, we assume all tuples to have the same score and thus be tied for the same position and call this random average precision.

Exact probabilistic inference. Whenever possible, we calculate GT rankings with a tool called SampleSearch [23, 27], which also serves to evaluate the cost of exact probabilistic inference. We describe the method of transforming the lineage DNF into a format that can be read by SampleSearch in [22].

Monte Carlo (MC). We evaluate the MC simulations for different numbers of samples and write MC(x) for x samples. For example, AP for MC(10k) is the result of sampling the individual tuple scores 10,000 times from their lineages and then evaluat- ing AP once over the sampled scores. The MAP scores together with the standard deviations are then the average over several repetitions.

Ranking by lineage size. To evaluate the potential of non-probabilistic methods for ranking answers, we also rank the answer tuples by decreasing size of their lineages; i.e., number of terms. Intuitively, a larger lineage size should indicate that an answer tuple has more “support” and should thus be more important.

Setup 1. We use the TPC-H DBGEN data generator [54] to generate a 1GB database to which we add a column P for each tuple and store it in PostgreSQL 9.2 [43]. We assign to each input tuple a random probability p_i uniformly chosen from the interval [0, p_{max}], resulting in an expected average input probability avg[p_i] = p_{max}/2. By using databases with avg[p_i] < 0.5, we can avoid output probabilities close to 1 for queries with very large lineages. We use the following parameterized query:

\[Q(a): = S(x, a), P(S(x, a), P(u, n), s \leq 1, n \text{ like } 2)\]

select distinct s.nationkey from Supplier, Partsupp, Part where s.supkey = p.supkey and p.ppartkey = p.partkey and s.supkey <= 1 and p.name like 2$2$

Parameters $1$ and $2$ allow us to change the lineage size. Tables Supplier, Partsupp and Part have 10k, 800k and 200k tuples, respectively. There are 25 different numeric attributes for nationkey and our goal is to efficiently rank these 25 nations. As baseline for not ranking, we use random average precision for 25 answers, which leads to MAP@10 = 0.220. This query has two minimal query plans and we will compare the speed-up from either evaluating both individually or performing a deterministic semi-join re-duction (Optimization 3) on the input tables.

Setup 2. We compare the run times for our three optimizations against evaluation of all plans for k-chain queries and k-star queries over varying database sizes (data complexities) and varying query sizes (query complexities). The k-chain queries have arity = 2 and several results, whereas the star queries have arity = 1 and cardinality = 1, representing a Boolean query:

k-chain: q(x₀, x₁) : = R₁(x₀, x₁), R₂(x₁, x₂), ..., Rₖ(xₖ−₁, xₖ)

k-star: q′(a′) : = R₁(a′, x₁), R₂(x₂), ..., Rₖ(xₖ), R₀(x₈, x₉, ..., xₙ)

We denote the length of the query with k, the number of tuples per table with n, and the domain size with N. We use integer values which are uniformly randomly drawn from the range \(\{0, 1, \ldots, N-1\}\). This parameter determines the selectivity and is varied as to keep the answer cardinality constant around 20-50 for chain queries, and the answer probability between 0.90 and 0.95 for star queries. For the data complexity experiments, we vary the number of tuples n per table between 100 and 10⁶. For the query complexity experiments, we vary k between 2 and 8 for chain queries. For these experiments, the optimized (and often extremely
long) SQL statements are “calculated” in JAVA and then sent to Microsoft SQL server 2012.

5.1 Run time experiments

**QUESTION 1.** When and how much do our three query optimizations speed up query evaluation?

![Result 1. Combining plans (Opt. 1) and using intermediate views (Opt. 2) almost always speeds up query times. The semi-join reduction (Opt. 3) slows down queries with high selectivities, but considerably speeds up queries with small selectivities.](image1)

Figures 5a to 5d show the results for setup 2 for increasing database sizes or query sizes. For example, Fig. 5a shows the performance of computing a 7-chain query which has 132 safe dissociations. Evaluating each of these queries separately takes a long time, while our optimization techniques bring evaluation time close to deterministic query evaluation. Especially on larger databases, where the running time is I/O bound, the penalty of the probabilistic inference is only a factor of 2-3 in this example. Notice here the trade-off between optimization 1,2 and optimization 1,2,3: Optimization 3 applies a full semi-join reduction on the input relations before starting the probabilistic plan evaluation from these reduced input relations.

This operation imposes a rather large constant overhead, both at the query optimizer and at query execution. For larger databases (but constant selectivity), this overhead is amortized. In practice, this suggests that dissociation allows us a large space of optimizations depending on the query and particular database instance that can conservatively extend the space of optimizations performed today in deterministic query optimizers.

**Figures 5e to 5g** compare the running times on setup 1 between dissociation with two minimal query plans ("Diss"), dissociation with semi-join reduction ("Diss + Opt3"), exact probabilistic inference ("SampleSearch"), Monte Carlo with 1000 samples ("MC(1k)"), retrieving the lineage only ("Lineage query"), and deterministic query evaluation without ranking ("Standard SQL").

We fixed $s_2 \in \{\%\text{red}\%\text{green}\%\,\%\text{red}\%, \%\text{green}\%\}\}$ and varied $s_1 \in \{500, 1000, \ldots, 10k\}$. Figure 5h combines all three previous plots and shows the times as function of the maximum lineage size (i.e. the size of the lineage for the tuple with the maximum lineage) of a query. We see here again that the semi-join reduction speeds up evaluation considerably for small lineage sizes (Fig. 5c shows speedups of up to 36). For large lineages, however, the semi-join reduction is an unnecessary overhead, as most tuples are participating in the join anyway (Fig. 5d shows overhead of up to 2).

**QUESTION 2.** How does dissociation compare against other probabilistic methods and standard query evaluation?

![Figures 5d to 5h show that SampleSearch does not scale to larger lineages as the performance of exact probabilistic inference depends on the tree-width of the Boolean lineage formula, which generally increases with the size of the data. In contrast, dissociation is independent of the treewidth. For example, SampleSearch needed 780 sec for calculating the ground truth for a query with max[ln] = 5.9k for which dissociation took 3.0 sec, and MC(1k) took 42 sec for a query with max[ln] = 4.2k for which dissociation took 2.4 sec. Dissociation takes only 10.5 sec for our largest query $s_2 = \%a\%$ and $s_1 = 10k$ with max[ln] = 35k. Retrieving the lineage for that query alone takes 5.8 sec, which implies that any probabilistic method that evaluates the probabilities outside of the database engine needs to issue this query to retrieve the DNF for each answer and would thus have to evaluate lineages of sizes around 35k in only 4.7 (≈ 10.5 - 5.8) sec to be faster than dissociation.]()

5.2 Ranking experiments

For the following experiments, we are limited to those query parameters $s_1$ and $s_2$ for which we can get the ground truth (and results from MC) in acceptable time. We systematically vary $p_{\text{max}}$ between 0.1 and 1 (and thus avg$p_{\text{lin}}$ between 0.05 and 0.5) and evaluate the rankings several times over randomly assigned input tuple probabilities. We only keep data points (i.e. results of individual ranking experiments) for which the output probabilities are not too close to 1 to be meaningful ($\max[p_{\text{lin}}] < 0.999999$).

**QUESTION 3.** How does ranking quality compare for our three ranking methods and which are the most important factors that determine the quality for each method?

![Figure 5i shows averaged results of our probabilistic methods for $s_2 = \%\text{red}\%\text{green}\%$. Shaded areas indicate standard deviations and the x-axis shows varying numbers of MC samples. We only used those data points for which avg$p_{\text{lin}}$ of the top 10 ranked tuples is between 0.1 and 0.9 according to ground truth (≈ 6k data points for dissociation and lineage, ≈ 60k data points for MC, as we repeated each MC simulation 10 times), as this is the best regime for MC, according to Result 4. We also evaluated quality for dissociation and ranking by lineage for more queries by choosing parameter values for $s_2$ from a set of 28 strings, such as \%s\%\%s\%\%s\% and \%h\%\%e\%\%a\%\%n\%\%d\%. The average MAP over all 28 choices for parameters $s_2$ is 0.997 for ranking by dissociation and 0.520 for ranking by lineage size (≈ 100k data points). Most of those queries have too large of a lineage to evaluate MC. Note that ranking by lineage always returns the same ranking for given parameters $s_1$ and $s_2$, but the GT ranking would change with different input probabilities.

![Result 3. Dissociation performs better than MC which performs better than ranking by lineage size.](image2)

**Result 3.** Dissociation performs better than MC which performs better than ranking by lineage size.

![Result 4. Ranking quality of MC increases with the number of samples and decreases when the average probability of the answer tuples avg$p_{\text{lin}}$ is close to 0 or 1.](image3)

**Result 4.** Ranking quality of MC increases with the number of samples and decreases when the average probability of the answer tuples avg$p_{\text{lin}}$ is close to 0 or 1.

**Figure 5j** shows the AP as a function of avg$p_{\text{lin}}$ of the top 10 ranked tuples according to ground truth by logarithmic scaling of the x-axis (each point in the plot averages AP over ≈ 450 experiments for dissociation and lineage and over ≈ 4.5k experiments for MC). We see that MC performs increasingly poor for ranking answer tuples with probabilities close to 0 or 1 and even approach the quality of random ranking (MAP@10 = 0.22). This is so because, for these parameters, the probabilities of the top 10 answers are very close, and MC needs many iterations to distinguish them. Therefore, MC performs increasingly poorly for increasing size of lineage but fixed average input probability avg$p_{\text{lin}}$ ≈ 0.5, as the average answer probabilities avg$p_{\text{lin}}$ will be close to 1. In order not to “bias against our competitor,” we compared against MC in its best regime with 0.1 < avg$p_{\text{lin}}$ < 0.9 in Fig. 5i.

The time needed for the lineage query thus serves as minimum benchmark for any probabilistic approximation. The reported times for SampleSearch and MC are the sum of time for retrieving the lineage plus the actual calculations, without the time for reading and writing the input and output files for SampleSearch.

*Results for MC and other parameters of $s_2$ are similar. However, the evaluation time for the experiments becomes quickly infeasible.*
Figure 5: Timing results: (a)-(c) For increasing database sizes and constant cardinalities, our optimizations approach deterministic SQL performance. (d) Our optimizations can even evaluate very large number of minimal plans efficiently (here shown up to 429 for an 8-chain query). (e)-(h) For the TPC-H query, the best evaluation for dissociation is within a factor of 6 of that for deterministic query evaluation. (i)-(p) Ranking experiments on TPC-H: Assumptions for each subfigure and conclusions that can be drawn are described in the main text in the respective result paragraph. [From Wolfgang: increase font size for annotations in figure!!!]
Result 5. Ranking by lineage size has good quality only when all input tuples have the same probability.

Figure 5K shows that ranking by lineage is good only when all tuples in the database have the same probability (labeled by \( p_i = \text{const} \) as compared to \( \text{avg}[p_i] = \text{const} \)). This is a consequence of the output probabilities depending mostly on the size of the lineages if all probabilities are equal. Dependence on other parameters, such as overall lineage size and magnitude of input probabilities (here shown for \( p_i = 0.1 \) and \( p_i = 0.5 \)), seem to matter only slightly.

Result 6. The quality of dissociation decreases with the average number of dissociations per tuple \( \text{avg}[d] \) and with the average input probabilities \( \text{avg}[p_i] \). Dissociation performs very well and notably better than MC(10k) if either \( \text{avg}[d] \) or \( \text{avg}[p_i] \) are small.

Each answer tuple \( a \) gets its score \( p_a \) from one of two query plans \( P_3 \) and \( P_4 \) that dissociate tuples in tables \( S \) and \( P \), respectively. For example, if the lineage size for tuple \( a \) is 100 and the lineage contains 20 unique suppliers from table \( S \) and 50 unique parts from table \( P \), then \( P_3 \) dissociates each tuple from \( S \) into 5 tuples and \( P_4 \) each tuple from \( P \) into 2 tuples, on average. Most often, \( P_2 \) will then give the better bounds as it has fewer average dissociations. Let \( \text{avg}[d] \) be the mean number of dissociations for each tuple in the dissociated table of its respective optimal query plan, averaged across all top 10 ranked answer tuples. For all our queries (even those with \( S_1 = 10k \) and \( S_2 = \% \)), \( \text{avg}[d] \) stays below 1.1 as, for each tuple, there is usually one plan that dissociates few variables. In order to understand the impact of higher numbers of dissociations (increasing \( \text{avg}[d] \)), we also measured AP for the ranking for each query plan individually. Hence, for each choice of random parameters, we record two new data points — one for ranking all answer tuples by using only \( P_3 \) and one for using only \( P_4 \) — together with the values of \( \text{avg}[d] \) in the respective table that gets dissociated. This allows us to draw conclusions for a larger set of parameters. Figure 5I plots MAP values as a function of \( \text{avg}[d] \) for the top 10 ranked tuples on the horizontal axis, and various values of \( \text{avg}[p_i] \) (\( \text{avg}[p_i] = 0.05, 0.10, \ldots , 0.5 \)). Each plotted point averages over at least 10 data points (some have 10, other several 1000s). Dashed lines show a fitted parameterized curve to the data points on \( \text{avg}[p_i] \) and \( \text{avg}[d] \). The figure also shows the standard deviations as shaded areas for \( \text{avg}[p_i] = 0.5 \). We see that the quality is very dependent on \( \text{avg}[p_i] \), as predicted by [Prop. 27].

Figure 5N maps the trade-off between dissociation and MC for the two important parameters for the quality of dissociation (\( \text{avg}[d] \) and \( \text{avg}[p_i] \)) and the number of samples for MC. For example, MC(1k) gives a better expected ranking than dissociation only for the small area above the thick red curve marked MC(1k). For MC, we used the test results from [Fig. 5] i.e. assuming \( 0.1 < \text{avg}[p_a] < 0.9 \) for MC. Also recall that for large lineages, having an input probability with \( \text{avg}[p_i] = 0.5 \) will often lead to answer probabilities close to 1 for which ranking is not possible anymore (recall Fig. 5K). Thus, for large lineages, we need small input probabilities to have meaningful interpretations. And for small input probabilities, dissociation considerably outperforms any other method.

Question 4. How much would the ranking change according to exact probabilistic inference if we scale down all input tuples?

Result 7. If the probabilities of all input tuples are already small, then scaling them further down does not affect the ranking much.

Here, we repeatedly evaluated the exact ranking for 7 different parameterized queries over randomly generated databases with one query plan that has \( \text{avg}[d] = 3 \), for two conditions: first on a probabilistic database with \( \text{avg}[p_i] \) input probabilities (we defined the resulting ranking as GT); then again on a scaled version, where all input probabilities in the database are multiplied by the same scaling factor \( f \in (0, 1) \). We then compared the new ranking against GT. [Figure 5n] shows that if all input probabilities are already small (and dissociation already works well), then scaling has little effect on the ranking. However, for \( \text{avg}[p_i] = 0.5 \) (and thus many tuples with \( p_i \) close to 1), we have a few tuples with \( p_i \) close to 1. These tuples are very influential for the final ranking, but their relative influence decreases if scaled down even slightly. Also note that even for \( \text{avg}[p_i] = 0.5 \), scaling a database by a factor \( f = 0.01 \) instead of \( f = 0.2 \) does not make a big difference. However, the quality remains well above ranking by lineage size (1). This suggests that the difference between ranking by lineage size (MAP = 0.529) and the ranking on a scaled database for \( f \to 0 \) (MAP = 0.879) can be attributed to the relative weights of the input tuples (we thus refer to this as “ranking by relative input weights”). The remaining difference in quality then comes from the actual probabilities assigned to each tuple. Using MAP = 0.220 as baseline for random ranking, 38% of the ranking quality can be found by the lineage size alone vs. 85% by the lineage size plus the relative weights of input tuples. The remaining 15% come from the actual probabilities (Fig. 5O).

Question 5. Does the expected ranking quality of dissociation decrease to random ranking for increasing fractions of dissociation (just like MC does for decreasing number of samples)?

Result 8. The expected performance of dissociation for increasing \( \text{avg}[d] \) for a particular query is lower bounded by the quality of ranking by relative input weights.

Here, we use a similar setup as before and now compare various rankings against each other: SampleSearch on the original database (“GT”); SampleSearch on the scaled database (“Scaled GT”); dissociation on the scaled database (“Scaled Diss”); and ranking by lineage size (which is unaffected by scaling). From Fig. 5P we see that the quality of Scaled Diss w.r.t. Scaled GT → 1 for \( f \to 0 \) since dissociation works increasingly well for small \( \text{avg}[p_i] \) (recall Prop. 27). We also see that Scaled Diss w.r.t. GT decreases towards Scaled GT w.r.t. GT for \( f \to 0 \). Since dissociation can always reproduce the ranking quality of ranking by relative input weights by first downsampling the database (though losing information about the actual probabilities) the expected quality of dissociation for smaller scales does not decrease to random ranking, but rather to ranking by relative weights. Note this result only holds for the expected MAP; any particular ranking can still be very much off.

6. RELATED WORK

Probabilistic databases. Current approaches to query evaluation on probabilistic databases can be classified into three categories: (i) incomplete approaches identify tractable cases either at the query-level [7, 17] or the data-level [38, 46, 50]; (ii) exact approaches [2, 30, 38, 39, 49] work well on queries with simple lineage expressions, but perform poorly on database instances with complex lineage expressions. (iii) approximate approaches either apply general purpose sampling methods to the entire database [29, 32, 33, 44] or approximate the number of models of the Boolean lineage expression [16, 40, 45]. Our work can be seen as a generalization of several of these techniques: Our algorithm returns the exact score if the query is safe [7, 39] or data-safe [30].

Lifted and approximate inference. Lifted inference was introduced in the AI literature as an approach to probabilistic inference that uses the first-formula order to exploit symmetries at the
grounded level [22]. This research evolved independently of that on probabilistic databases, and the two have many analogies: A formula is called domain liftable iff its data complexity is in polynomial time [28], which is the same as a safe query in probabilistic databases, and the FO-D-NNF circuits described in [12] correspond to the safe plans discussed in this paper. See [11] for a recent discussion on the similarities and differences.

Representing Correlations. The most popular approach to represent correlations between tuples in a probabilistic database is by a Markov Logic network (MLN) which is a set of soft constraints [13]. Quite remarkably, all complex correlations introduced by an MLN can be rewritten into a query over a tuple-independent probabilistic database [24] [26] [31]. In combination with such rewritings, our techniques can be also applied to MLNs if their rewritings result in conjunctive queries without self-joins.

Dissociation. Dissociation was first introduced in the workshop paper [20], presented as a way to generalize graph propagation algorithms to hypergraphs. Theoretical upper and lower bounds for dissociation of Boolean formulas, including Theorem 8 were proven in [25]. Dissociation is related to a technique called relaxation for probabilistic inference in graphical models [10].

7. CONCLUSIONS AND OUTLOOK

This paper proposes to approximate probabilistic query evaluation by evaluating a fixed number of safe queries, each providing an upper bound on the true probability, then taking their minimum. We provide an algorithm that takes into account important schema information to enumerate only the minimal necessary plans among all possible plans, and prove it to be a strict generalization of all known results of PTIME self-join free conjunctive queries. We describe relational query optimization techniques that allow us to evaluate all minimal queries in a single query and very fast. Our evaluations show that the optimizations of our approach bring probabilistic query evaluation close to standard query evaluation while providing high ranking quality. In future work, we plan to generalize the approach to full first-order queries.

Acknowledgements. This work was supported in part by NSF grants IIS-0513877, IIS-0713576, IIS-0915054, and IIS-1115188. We thank the reviewers for their careful reading of this manuscript and their detailed feedback. WG would also like to thank Manfred Hauswirth for a question in 2007 that was crucial for this paper.

8. REFERENCES


method for fast error estimation in approximate query processing. In SIGMOD,

[56] C. Zhang and C. Ré. Towards high-throughput Gibbs sampling at scale: a study