FastPDB: Towards Bag-Probabilistic Queries at Interactive Speeds

AARON HUBER, University at Buffalo, USA
OLIVER KENNEDY, University at Buffalo, USA
ATRI RUDRA, University at Buffalo, USA
ZHUOYUE ZHAO, University at Buffalo, USA
SU FENG, Nanjing Tech University, China
BORIS GLAVIC, University of Illinois, Chicago, USA

Probabilistic databases (PDBs) provide users with a principled way to query data that is incomplete or imprecise. In this work, we study computing expected multiplicities of query results over probabilistic databases under bag semantics which has PTIME data complexity. However, does this imply that bag probabilistic databases are practical? We strive to answer this question from both a theoretical as well as a systems perspective. We employ concepts from fine-grained complexity to demonstrate that exact bag probabilistic query processing is fundamentally less efficient than deterministic bag query evaluation, but that fast approximations are possible by sampling monomials from a circuit representation of a result tuple's lineage. A remaining issue, however, is that constructing such circuits, while in PTIME, can nonetheless have significant overhead. To avoid this cost, we utilize approximate query processing techniques to directly sample monomials without materializing lineage upfront. Our implementation in FastPDB provides accurate anytime approximation of probabilistic query answers and scales to datasets orders of magnitude larger than competing methods.

CCS Concepts: • **Information systems** → **Data management systems**; Database query processing; **Incomplete data**; **Uncertainty**; • **Theory of computation** → *Approximation algorithms analysis.*

Additional Key Words and Phrases: probabilistic data model, parameterized complexity, fine-grained complexity, lineage polynomials, approximate query processing

ACM Reference Format:
Aaron Huber, Oliver Kennedy, Atri Rudra, Zhuoyue Zhao, Su Feng, and Boris Glavic. 2025. FastPDB: Towards Bag-Probabilistic Queries at Interactive Speeds. *Proc. ACM Manag. Data* 3, N1 (SIGMOD), Article 041 (January 2025), 24 pages. https://doi.org/10.1145/XXXXXXX

1 INTRODUCTION

Probabilistic databases (PDBs) [55] provide users with a principled method for querying data that is incomplete or imprecise. For example, in heuristic data cleaning [8, 47, 50, 57], systems may emit a PDB when insufficient data exists to select the "correct" data repair. In this work we investigate strategies for efficiently evaluating bag-relational queries over probabilistic databases. For ease of presentation, we make the simplifying assumption that input relations are sets[1]. Specifically, we study computing *expected multiplicities* of query result tuples for positive relational algebra queries ($\mathcal{RA}^+$) which corresponds to union-select-project-join (USPJ) queries in SQL. We focus on bag-TIDBs, a generalization of *tuple-independent databases* (TIDBs) for bag semantics where each

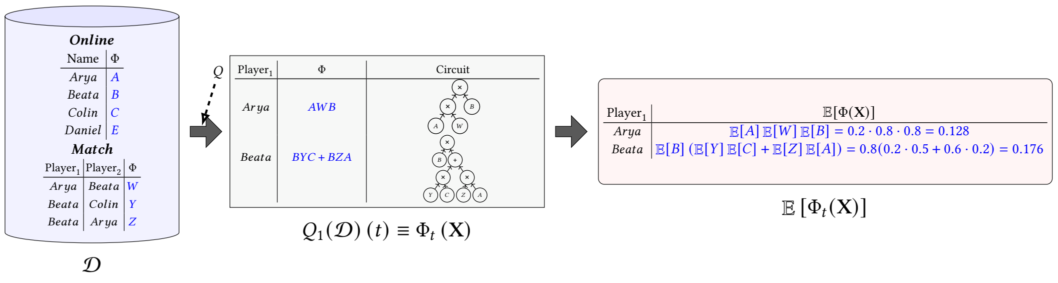
[1]Our accompanying technical report [6] generalizes our results to a broader class of inputs, including a generalization of block-independent databases (BIDBs) and any bag database where tuple multiplicities (in input relations) are bounded by a constant.

Authors' addresses: Aaron Huber, University at Buffalo, Buffalo, USA, ahuber@buffalo.edu; Oliver Kennedy, University at Buffalo, Buffalo, USA, okennedy@buffalo.edu; Atri Rudra, University at Buffalo, Buffalo, USA, atri@buffalo.edu; Zhuoyue Zhao, University at Buffalo, Buffalo, USA, zzhao35@buffalo.edu; Su Feng, Nanjing Tech University, Nanjing, China, sufeng@njtech.edu.cn; Boris Glavic, University of Illinois, Chicago, Chicago, USA, bglavic@uic.edu.

2025. 2836-6573/2025/1-ART041 $15.00
https://doi.org/10.1145/XXXXXXX

	Design Choices		Theoretical Guarantees			Empirical Behavior	
System	Semantics	Intermediate Rep	Exact?	Variance?	Asympt. Fast	Runtime vs Det.	
Trio [7]	Bag	SoM Lineage	✓	0	×	$[5-100]\times$ slower	[16, 42]
MCDB [32]	Bag	Parallel Samples	×	Unbounded	✓	$[10-100]\times$ slower	[18, 33]
Pip [33]	Bag	SoM Lineage	Sometimes	Low	×	$[10-100]\times$ slower	*
GProM-e [4]	Bag	SoM Lineage	✓	0	×	$[1-100]\times$ slower	[46], *
ProvSQL-e [51]	Bag	Circuit Lineage	✓	0	×	$[1-100]\times$ slower	*
ProvSQL-a [51]	Bag	Circuit Lineage	×	Low	✓	$[1-100]\times$ slower	*
FastPDB	Bag	Lineage Sample	× (Anytime)	Unbounded	✓	Anytime (often faster)	*

Fig. 1. Comparison of approaches for implementing PDB; We extend provenance systems GProM and ProvSQL to compute exact (-e) or approximate (-a) expected multiplicities, as well as our proposed system, FastPDB. Systems and results highlighted in blue are new. Asymptotically fast means that the system is capable of asymptotically matching deterministic query runtimes.



Player: $p[A=1]=0.2 \quad p[B=1]=0.8 \quad p[C=1]=0.5 \quad p[E=1]=1$

Match: $p[W=1]=0.8 \quad p[Y=1]=0.2 \quad p[Z=1]=0.6$

Fig. 2. Computing lineage polynomials and calculating expected multiplicities based on the input probabilities and lineage.

input tuple is associated with $\{0,1\}$-valued random variables that represent a tuple's uncertain multiplicity. These variables are assumed to be mutually independent. The motivation for this setting is two-fold: (i) the most commonly studied query semantics for set probabilistic databases is to compute the marginal probability of a query result tuple over TIDBs. As this is equivalent to computing the expectation of a Boolean random variable encoding the tuple's existence, computing expected multiplicities, which is computing the expectation of integer random variable encoding a tuple's multiplicity, is a natural generalization (see Sec. 3 for an overview of work on set-PDBs); (ii) computing expected multiplicities is equivalent to computing expected counts for group-by `count` aggregation queries. Thus, our results also lay the foundation for a further investigation into more complex queries.

In the following we will use bag-PQP (bag probabilistic query processing) to refer to computing expected multiplicities over bag-TIDBs and call systems that implement this semantics bag-PDBs. It has been shown [26] that bag-PQP has PTIME data complexity. The main question we seek to answer in this work is: *is it possible to build efficient* bag-PDB*s*? We investigate this question from both a theoretical as well as a practical perspective. Based on this exploration, we propose a new probabilistic database, FastPDB, which leverages techniques from approximate query processing to implement bag-PQP at interactive speeds.

Lineage. A common approach used in probabilistic databases is to first compute the *lineage* of each query result and then compute expectations from the lineage. Under bag semantics, the lineage of an output tuple is a polynomial over integer-valued random variables. Each such variable represents the multiplicity of an input tuple in the lineage polynomial of an output tuple t. Evaluating t's polynomial for an assignment that sets each variable to the corresponding input tuple's multiplicity yields the multiplicity of t. Lineage is a necessary component of a bag-PQP, as simply annotating output tuples with expected multiplicities is not a closed representation system for query results.

That is, even if tuples in a database D are uncorrelated, a query Q_1 may produce correlated outputs — a fact not captured by the expected multiplicities annotating each tuple. A second query Q_2 applied to the *materialized* output of the first (i.e., $Q_2(Q_1(D))$) does, in general, not produce correct expected multiplicities. Lineage captures correlations, and as such, permits the closed representation necessary for materialized views. Thus, we focus on bag-PQP based on lineage formulas which for bag semantics are polynomials with natural number coefficients.

Example 1.1. *Consider the* bag-TIDB *shown in Fig. 2, which models the player base of a 1-vs-1 game service: table* `Online` *tracks which players are online while each record in* `Match` *indicates a viable player pairing. Each player is annotated with a* $\{0, 1\}$*-valued random variable (*`A-E`*), where a 1 indicates that the player is online. Similarly, each potential match is annotated a variable (*`W,Y,Z`*), where a 1 indicates that* $\texttt{Player}_2$ *is a suitable opponent for* $\texttt{Player}_1$ *(this relationship need not be symmetric). We show the random variables for each input tuple in Fig. 2. This is a* bag-TIDB*, i.e., all random events are independent. Consider the query shown below*

$$Q = \pi_{Player_1}(Online \bowtie_{Name=Player_1} Match \bowtie_{Player_2=Name} Online)$$

which computes the expected number of potential matches available for each player: both players must be logged on, and the match must be viable. For example `Arya` *has a potential match if she is online (*$\mathtt{A} = 1$*), if she has a viable match with* `Beata` *(*$\mathtt{W} = 1$*), and if* `Beata` *is also online (*$\mathtt{B} = 1$*). Observe that the resulting number of potential matches is exactly the product of these random variables (*`AWB`*). Similarly, the number of* `Beata`*'s potential opponents at any given time may be computed as* $\Phi_{Beata}(\mathbf{X}) = \mathtt{BYC} + \mathtt{BZA}$ *(where* $\mathbf{X}$ *is the vector of* $\{0, 1\}$*-valued variables representing input tuples i.e.,* $\mathtt{A}, \mathtt{B}, \ldots$*). We refer to these arithmetic expressions (e.g.,* $\Phi_{Beata}(\mathbf{X})$*) as* lineage polynomials*. Given that tuples are independent events and using linearity of expectation, the expected multiplicity can be computed by pushing the expectation down to the individual variables. For example, for result tuple* (`Beata`)*, we get:*

$$\mathbb{E}[\Phi_{Beata}(\mathbf{X})] = \mathbb{E}[B(YC + ZA)] = \mathbb{E}[B](\mathbb{E}[Y]\mathbb{E}[C] + \mathbb{E}[Z]\mathbb{E}[A])$$

Complexity of Exact Answers. bag-PQP is known to be PTIME in data complexity [26] for positive relational algebra ($\mathcal{RA}^+$). However, this result says little about whether bag-PQP-based systems can be competitive with deterministic databases. We study the problem through the lens of fine-grained complexity. Specifically, we show that the problem of bag-PQP is hard by identifying a class of queries and: (i) determining an asymptotic upper bound on their deterministic runtime (based on existing algorithms), (ii) determining an asymptotic lower bound on the analogous bag-PQP query (based on standard complexity assumptions), and (iii) demonstrating that the ratio of the two is polynomially large. Thus, bag-PDB are necessarily (asymptotically) slower than deterministic databases.

To relate these theoretical results with practice, Fig. 1 shows a comparison of exact and approximate approaches for computing expected multiplicities from related work and approaches introduced in this work (highlighted in blue). Note that any approach for computing lineage (provenance polynomials [25] for bag semantics) can be extended to compute expected multiplicities by implementing the approach outlined in the example above. We present two approaches *GProM-e* (GProM [4]) and *ProvSQL-e* (ProvSQL [51]). GProM uses a flat *sum-of-monomials* (*SoM*) encoding of lineage (e.g., as shown in the Φ column in Fig. 2) while ProvSQL uses a circuit representation (e.g., column "Circuit" in Fig. 2), which can be exponentially more concise in extreme cases. As we will discuss in Sec. 5.1, it is possible to construct a circuit representation of lineage with linear overhead over deterministic query evaluation. Producing exact expected counts in either representation comes at a high cost: as we will demonstrate for some settings the overhead over deterministic query processing is larger than a factor of 10.

Approximating Expected Multiplicities. Given this negative theoretical result and experimental evidence, we investigate approximations. First off, we observe that existing systems that approximate probabilistic queries can be repurposed to compute approximate expected multiplicities. Systems like *Trio* [1, 42] that compute the expectation of `count` aggregate queries can be used to compute expected multiplicities. Similarly, *MCDB* [32] can be applied to sample a set of possible worlds and then estimate multiplicities based on multiplicities computed from each sampled world. However, as shown in Fig. 1 these systems still carry a significant overhead: empirically for both systems, and also asymptotically for Trio and Pip as a result of these system's use of SoM lineage.

Our next contribution is an algorithm that computes an $(1\pm\epsilon)$-approximation of a tuple's expected multiplicity from a circuit encoding of lineage whp. with an asymptotic runtime that is equal to deterministic queries.[2] This algorithm computes the estimation based on a set of monomials sampled from the circuit representation of a tuple's lineage. We implement this algorithm in ProvSQL [51] (*ProvSQL-a*) as a UDF that samples from ProvSQL's circuit representation of lineage. ProvSQL-a has bounded variance whp. as it produces a $(1 \pm \epsilon)$-approximation of expected multiplicities with at least δ probability. However, in practice the major bottleneck of this algorithm is constructing lineage (Fig. 1).

Sampling Monomials Directly using Approximate Query Processing. Based on the observation that our approximation algorithm samples monomials from a tuple's lineage, we next explore whether it is possible to generate such samples without having to first construct a lineage circuit. Our main insight here is that *approximate query processing* (AQP) techniques for estimating the result of an aggregation over the result of a join [29, 35] can be used for this purpose. Specifically, constructing all monomials can be expressed as a join query. Propagating probabilities of tuples as part of this join then provides sufficient information to compute the probability of each monomial and then in turn use `SUM` aggregation to compute the expected multiplicities in a fashion similar to Example 1.1. We develop a system called FastPDB that extends the Wanderjoin implementation in XDB [35] for online aggregation over joins. FastPDB provides anytime approximation for expected multiplicities. Our approach achieves performance that is even better than deterministic query processing in some cases. In contrast to ProvSQL-a, the variance of this approach is unbounded. As in XDB, the use of rejection sampling limits accuracy on queries with highly selective filtering predicates. However, in practice we can compute precise estimates in a fraction of the time needed by ProvSQL-a. Given that we do not construct full lineage, the approach is very robust for queries that generate very large lineage expressions on which both ProvSQL and GProM perform poorly.

Contributions. In summary, we make the following contributions:

Fine-grained Complexity Analysis: In Sec. 4 we show that exact bag-PQP has higher asymptotic runtime than deterministic queries.
A $(1 \pm \epsilon)$-approximation: In Sec. 5 we extend ProvSQL [51] with an algorithm that computes a $(1 \pm \epsilon)$-approximation of multiplicities with the same asymptotic runtime as a deterministic query.
Push-Down Sampling: In Sec. 6 we show that the sampling step of our approximation algorithm can be 'pushed down' into the lineage construction process. Based on this insight, we present FastPDB, a bag-PDB that leverages WanderJoin [35]. To our knowledge, this is the first PDB using AQP techniques.
Experimental Evaluation: In Sec. 7, we contrast the performance of FastPDB against (i) other probabilistic database systems [32, 33], (ii) lineage-based solutions [4, 51], and (iii) deterministic query processing. In general, FastPDB produces (approximate) results significantly faster than

[2]As query execution engines and optimizers differ widely across systems, we use a "reasonable" model for deterministic query runtime. See [6] for more details.

$$(\sigma_\theta R)(t) = \begin{cases} R(t) & \textbf{if } \theta(t) \\ 0 & \textbf{otherwise} \end{cases} \qquad (\pi_A R)(t) = \sum_{t':\ t=\pi_A t'} R(t')$$

$$(R_1 \bowtie R_2)(t) = R_1(\pi_{sch(R_1)}(t)) \cdot R_2(\pi_{sch(R_2)}(t))$$
$$(R_1 \cup R_2)(t) = R_1(t) + R_2(t)$$

Fig. 3. Semantics for Bag-$\mathcal{RA}^+$ [25]

competing approaches and scales to datasets that are several orders of magnitude larger than the baselines.

2 BAG PROBABILISTIC DATABASES

Bag Semantics. We use $\{\ldots\}$ to denote sets, $\{\!|\ldots|\!\}$ to denote multisets (bags), $\langle\ldots\rangle$ to denote tuples, and $\mathbb{D}$ to denote a universal domain of values. We will use $|\cdot|$ to denote the number of elements in sets, bags, or tuples. A bag relation R with $sch\,(R) \coloneqq \langle A_1, \ldots, A_\ell \rangle$ with arity $arity\,(R) = |sch(R)|$ is a bag of tuples $t \in \mathbb{D}^\ell$. A bag relation can be modeled as a function $R : \mathbb{D}^{arity(R)} \longrightarrow \mathbb{N}$ that associates with each tuple t a multiplicity (the number of duplicates of t in R). Then $R(t)$ denotes the multiplicity of t in R. A database $D \coloneqq \{R_1, \ldots, R_m\}$ is a set of m bag relations (and $sch\,(D) \coloneqq \{sch\,(R_i)\,|1 \leq i \leq m\}$). We use $t \in R$ to denote that tuple t exists in R, i.e., $R(t) > 0$. We also apply the same notation for databases. Fig. 3 shows the semantics of positive relational algebra $\mathcal{RA}^+$ over bags: the relation (function) returned by an operator is defined point-wise for each tuple that may exist in the operator's output. For instance, the multiplicity of a tuple t in the result of union is the sum of the multiplicities of t in both inputs; for natural join we multiply the multiplicities of two input tuples that join (agree on the common attributes); and projection sums up the multiplicities of all inputs t' that after projection are equal to the result tuple t.

Bag Probabilistic Databases and Expected Multiplicities. A bag-PDB $\mathcal{D}$ is a pair $(\Omega, \mathcal{P})$ where $\Omega = \{D_1, \ldots, D_n\}$ is a set of bag databases called *possible worlds* and $\mathcal{P}$ is a probability distribution over Ω. We use $\overline{D}$, called the bounding deterministic database of a bag-PDB $\mathcal{D}$, to denote the set of tuples that appear in at least one world: $\overline{D} = \{\ t \mid \exists D \in \Omega : t \in D\ \}$. Given a bag-PDB $\mathcal{D}$ we want to compute for a $\mathcal{RA}^+$ query Q the *expected multiplicity* of a result tuple t: summing up over $D \in \Omega$ the product of the multiplicity of the tuple $Q(D)(t)$ with the probability of the world $(\mathcal{P}(D))$.

Definition 2.1 (Expected Multiplicity). *Given a bag* PDB $\mathcal{D} = (\Omega, \mathcal{P})$, *query* Q, *and tuple* t, *the* expected multiplicity *of* t *is:* $\mathbb{E}\left[Q\,(D)\,(t)\right] = \sum_{D \in \Omega} Q\,(D)\,(t) \cdot Pr\,[D]$.

bag-TIDBs. As noted above, we use an adaptation of TIDBs to bags, with input tuples having multiplicity in $\{0, 1\}$. A further generalization to block-independent databases and another variant of bag-TIDBs where each input tuple is associated with a probability distribution over possible multiplicities from 0 to some constant c can be found in [6].

Definition 2.2 (TIDB). *A bag tuple independent database (*bag-TIDB*)* $\mathcal{D}$ *is a* bag-PDB *that is specified as a pair* $(\overline{D}, \mathbf{p})$ *where* $\overline{D}$ *is a bounding database and* $\mathbf{p}$ *associates each tuple* $t \in \overline{D}$ *with the marginal probability* $\mathbf{p}(t) = p_t$ *of existing with multiplicity* 1. *The worlds of a* bag-TIDB *are all bag databases* D *generated by selecting a subset* $S \subseteq \overline{D}$ *and setting* $D(t) = 1$ *if* $t \in S$ *and* $D(t) = 0$ *otherwise. Such a world* D *for* $S \subseteq \overline{D}$ *has probability:*

$$\mathcal{P}\,[D] = \prod_{t \in S} p_t \cdot \prod_{t \notin S} (1 - p_t)$$

We will sometimes specify the probabilities $\mathbf{p}$ of a bag-TIDB with $\overline{D} = \{t_1, \ldots, t_n\}$ as a vector $\mathbf{p} \in [0, 1]^n$.

Example 2.3. *Consider the* bag-TIDB $\mathcal{D}$ *with a single relation* R *shown below with four possible worlds* $\Omega = \{D_1, D_2, D_3, D_4\}$ *with probabilities as shown (in the left table) below:*

$$D_1 = \emptyset \qquad D_2 = \{\!|\langle 1,2\rangle|\!\} \qquad D_3 = \{\!|\langle 1,3\rangle|\!\} \qquad D_4 = \{\!|\langle 1,2\rangle, \langle 1,3\rangle|\!\}$$

$$\mathcal{P}[D_1] = 0.7 \cdot 0.4 = 0.28 \qquad \mathcal{P}[D_2] = 0.3 \cdot 0.4 = 0.12$$

$$\mathcal{P}[D_3] = 0.7 \cdot 0.6 = 0.42 \qquad \mathcal{P}[D_4] = 0.3 \cdot 0.6 = 0.18$$

Evaluating query $Q := \pi_A(R)$ *over* $\mathcal{D}$ *returns a single result* $\langle 1\rangle$. *The expected multiplicity of this tuple is shown (in the right table) below and derived like this:*

$$\mathbb{E}[Q(\mathcal{D})(\langle 1\rangle)] = \sum_{D \in \Omega} Q(D)(\langle 1\rangle) \cdot \mathcal{P}[D]$$
$$= 0 \cdot 0.28 + 1 \cdot 0.12 + 1 \cdot 0.42 + 2 \cdot 0.18 = 0.9$$

bag-TIDB $\mathcal{D}$

A	B	p_t
1	2	0.3
1	3	0.6

Query Results with Expected Multiplicities

A	$\mathbb{E}[Q(\mathcal{D})(t)]$
1	0.9

2.1 Provenance Polynomials

For bag-PDBs, the lineage of a tuple is a so-called provenance polynomial [25]: a polynomial with over integer-valued random variables representing tuples in the input PDBs. Importantly, the expectation of a provenance polynomial for a tuple t is equal to the expected multiplicity of the tuple. Before discussing provenance polynomials, we first introduce the standard monomial basis (SMB).

General polynomials. We use $[0, K]$ to denote $\{0, 1, \ldots, K\}$ and $[0, K]^n$ to denote the set of vectors of length n with values from $[0, K]$. Consider a set $\mathbf{X} = \{X_i\}_{i=1}^n$ of variables. A *general polynomial* ϕ with *degree* $K \in \mathbb{N}$ is of the form:

$$\phi(\mathbf{X}) = \sum_{\mathbf{d} \in [0,K]^n} c_{\mathbf{d}} \cdot \prod_{i \in [0,n]} X_i^{\mathbf{d}[i]} \qquad \text{where } c_{\mathbf{d}} \in \mathbb{N}. \tag{1}$$

That is, in this representation every possible monomial over $\mathbf{X}$ is assigned a coefficient in $\mathbb{N}$. Given a vector of exponents $d \in [0, K]^n$, a monomial M over $\mathbf{X}$ is of the form $c \cdot \prod_{t \in S} X_t^{d_t}$ for some constant c. We sometimes write $\phi(\mathbf{X})$ to emphasize that ϕ is over variables $\mathbf{X}$ and use $\phi(\mathbf{c})$ for a vector of constants $\mathbf{c}$ with $|\mathbf{c}| = |\mathbf{X}|$ to denote evaluating ϕ over the assignment $\mathbf{X}[i] = \mathbf{c}[i]$ for $i \in [1, |\mathbf{X}|]$.

Definition 2.4 (Standard Monomial Basis). *Polynomial* $\phi(\mathbf{X})$ *is in* standard monomial basis *(SMB) if it is the result of dropping all monomials with* $c_{\mathbf{d}} = 0$ *from a general form polynomial (Eq.* (1)*).*[3]

Provenance polynomials. Consider a bag-PDB $\mathcal{D}$ with bounding database $\overline{D}$ and a $\mathcal{RA}^+$ query Q. Let $\mathbf{X} = \{\, X_t \mid t \in \overline{D} \,\}$ be a set of variables, one per tuple in $\overline{D}$. A lineage expression (provenance polynomial) for a query result tuple t is a polynomial over $\mathbf{X}$ that captures the relationship between the output tuple and the input tuples that were combined (via query Q) to produce t. A provenance polynomial is denoted $\Phi(\mathbf{X})$ or $\Phi[Q, \overline{D}, t](\mathbf{X})$ if we want to specify the query, database, and result tuple.

Fig. 4 defines $\Phi[Q, \overline{D}, t](\mathbf{X})$ inductively for $\mathcal{RA}^+$ queries. Note that, in contrast to Fig. 3, where $R(t)$ denotes the actual multiplicity of t in R, here we use $R(t)$ to denote the *variable* representing the multiplicity of t. The correspondence between probability query answers and lineage for set PDBs generalizes to expected multiplicities over bag-PDBs.

[3]Any polynomial can be rewritten in SMB.

$$\Phi[\pi_A(Q), \overline{D}, t] = \sum_{t' : \pi_A(t') = t} \Phi[Q, \overline{D}, t'] \qquad \Phi[\sigma_\theta(Q), \overline{D}, t] = \begin{cases} \Phi[Q, \overline{D}, t] & \text{if } \theta(t) \\ 0 & \text{otherwise} \end{cases}$$

$$\Phi[Q_1 \cup Q_2, \overline{D}, t] = \Phi[Q_1, \overline{D}, t] + \Phi[Q_2, \overline{D}, t] \qquad \Phi[R, \overline{D}, t] = R(t)$$

$$\Phi[Q_1 \bowtie Q_2, \overline{D}, t] = \Phi[Q_1, \overline{D}, \pi_{sch(Q_1)} t] \cdot \Phi[Q_2, \overline{D}, \pi_{sch(Q_2)} t]$$

Fig. 4. Computing lineage polynomials for a $\mathcal{RA}^+$ query Q over $\mathcal{D}$ with bounding database $\overline{D}$ and variables $\mathbf{X} = (X_t)_{t \in \overline{D}}$.

Lemma 2.5 ([6]). *Given $\mathcal{D} = (\Omega, \mathcal{P})$ with bounding database $\overline{D}$, query Q, tuple $t \in Q(\overline{D})$. $\mathbb{E}\left[\Phi[Q, \overline{D}, t]\right] = \mathbb{E}\left[Q\left(\mathcal{D}\right)(t)\right]$.*

Lem. 2.5 implies that expected multiplicities can be computed from lineage. As we will see in Sec. 4.2, expectations of lineage polynomials can be computed for bag-TIDBs by evaluating a reduced forms of lineage polynomials over the probabilities of input tuples.

Example 2.6. *Consider the polynomial encoding of the* bag-TIDB *from Example 2.3 shown below where each tuple is associated with a variable (see below). The lineage polynomial for result tuple $\langle 1 \rangle$ is:*

$$\Phi[Q, \overline{D}, \langle 1 \rangle] = X + Y$$

To compute $\mathbb{E}\left[\Phi[Q, \overline{D}, \langle 1 \rangle]\right] = \mathbb{E}\left[X + Y\right]$ we can use linearity of expectation to get $\mathbb{E}\left[X + Y\right] = \mathbb{E}\left[X\right] + \mathbb{E}\left[Y\right]$. Now substituting the definition of expectation we get:

$$\sum_{x \in \{0,1\}} x \cdot Pr\left[X = x\right] + \sum_{y \in \{0,1\}} y \cdot Pr\left[Y = y\right] = 1 \cdot 0.3 + 1 \cdot 0.6 = 0.9$$

As guaranteed by Lem. 2.5, we get same answer as in Example 2.3.

A	B	Φ	$Pr\left[\Phi = 1\right]$
1	2	X	0.3
1	3	Y	0.6

A	Φ	$\mathbb{E}\left[\Phi\right]$
1	$X + Y$	0.9

3 RELATED WORK

Probabilistic databases have been studied extensively; [52, 55] provide comprehensive surveys. The classical problem studied for set-PDBs is to compute the marginal probability of a tuple to exist in the result of a query. As this can be viewed as computing the expectation of a Boolean variable (is the tuple in the result or not), the natural generalization for bag semantics that we study in this work is to compute the *expected multiplicity* of a result tuple. In contrast to the set case where this problem is #P-hard in data complexity even for TIDBs [24], computing expected multiplicities is in PTIME [26]. For the set case, multiple tractable classes of queries [11–13, 21, 48, 49] and database instances [2] have been identified. However, evaluating $\mathcal{RA}^+$ queries in general requires approximation techniques to be feasible [14, 19, 20, 23, 44]. As one example, Gatterbauer and Suciu [23] propose the use of extensional query evaluation — an evaluation strategy that assumes full independence between monomials in a set-PDB — as a bound. Fink et. al. [19] propose an anytime algorithm that gradually shrinks bounds on result probabilities. Although much work in this space focuses on set-PDBs, several efforts explore bag semantics. As previously noted, MCDB [5, 32] and Pip [33] are two major approaches. However, without approximate query processing, this approach uses a SoM encoding that carries significant overhead. Grohe et. al., [26] explored the complexity of queries that compute the probability that the multiplicity of a result tuple exceeds some constant c, relating it the complexity of set-PDB. Our accompanying technical report [6] handles computing expected multiplicities for arbitrary but a fixed constant c while this paper states the problem for $c = 1$.

Feng et. al. [16–18], and Guagliardo and Libkin [27] propose an approach to incomplete bag databases, bounding the space of certain / possible answers. Orr et. al., [45] explore the dual of our work, using a probabilistic data model to formalize AQP. [53] studied the problem of approximating certain and possible answers over databases with missing data.

In terms of AQP, most closely related to our work are approximations for aggregates over join results. The most common approach is use a random sample from join results to compute an unbiased estimation of the aggregation. Ripple join [29], the first algorithm for general joins, uniformly samples from all tables in a round-robin fashion, but is inefficient for joins with low selectivity. Deng et al. [15] and Kyoungmin et. al. [34] simultaneously found an optimal combinatorial uniform join sampling algorithm, albeit with linear-time query-time preprocessing. Wander Join [35] samples non-uniformly over k-ary joins based on random walks, is faster than ripple join, and does not require query-time pre-processing (see Sec. 6.2 for further discussion). See [37, 40, 41] for AQP surveys.

4 ON THE COMPLEXITY OF BPQP

In this section, we demonstrate that there is a fundamental complexity gap between bag-PQP and deterministic query evaluation. We will show that there exists a class of $\mathcal{RA}^+$ queries Q^k_{hard} for which exactly computing the *expected multiplicity* of result tuples over a bag-TIDB has a higher (fine-grained) complexity than deterministic query processing using an existing query evaluation algorithm. Specifically, in Sec. 4.3, we present reductions from well-studied sub-graph counting problems to bag-PQP for Q^k_{hard}. These reductions establish a lower bound on the complexity of bag-PQP for Q^k_{hard} and contrast it with a deterministic algorithm.

4.1 Subgraph Counting and Q^k_{hard}

Our hardness results are based on (exactly) counting the number of (not necessarily induced) subgraphs in a graph G isomorphic to a graph H, and known standard hardness results and assumptions. Let $\#(G, H)$ denote the number of subgraphs isomorphic to H in graph G. H is considered as being of constant size and G is the input. Let $\#(G, \vdots \cdots \vdots^k)$ be the number of k-matchings[4] and $T_{match}(k, G)$ the optimal runtime of computing $\#(G, \vdots \cdots \vdots^k)$ exactly. Our results are based on the following *known* (conditional) hardness results:

THEOREM 4.1 ([9]). *Given a positive integer k and undirected graph $G = (V, E)$ with no self-loops or parallel edges, $T_{match}(k, G) \geq \omega\left(f(k) \cdot |E|^c\right)$ for any function f and any constant c independent of $|E|$ and k (assuming $\#W[0] \neq \#W[1]$).*

THEOREM 4.2 ([10]). *Given a positive integer $k > 1$ and undirected graph $G = (V, E)$, $T_{match}(k, G) \geq |V|^{\Omega(k/\log k)}$ assuming the Exponential Time Hypothesis (ETH).*

$\#W[0]$ and $\#W[1]$ are parameterized complexity classes for counting problems [22], where the complexity is analyzed separately for classes of instances parameterized by k, e.g., the matching size. The exponential time hypothesis claims that any algorithm solving 3-SAT has a runtime of at least $2^{c \cdot n}$ for some fixed $c > 0$ [31].

The hard query. Consider the following variation of the query from Fig. 2, using relation $V(U)$ in place of `Online` and $E(U_1, U_2)$ in place of `Match`.

$$Q_1 := \pi_{\emptyset}\left(V \bowtie_{U=U_1} E \bowtie_{U_2=U} V\right).$$

[4] A k-matching is a set of k edges where no pair of edges has the same endpoint.

Noting that Q_1 always emits exactly one tuple, we define a family of queries Q^k_{hard} for arbitrary *join width k* as follows:

$$Q^k_{hard} \coloneqq \pi_\emptyset(\underbrace{Q_1 \bowtie \cdots \bowtie Q_1}_{\text{k times}}). \tag{2}$$

Let graph $G = (V_G, E_G)$ be a set of n vertices, and let D_G (resp., $\mathcal{D}_G$) be the database (resp. bag-TIDB) with relations V and E as defined next. D_G consists of $V(U) = V_G$ and $E(U_1, U_2) = E_G$. The query $Q_1(D_G)$ simply computes the number of edges in E where both endpoints exist in V. Query $Q^k_{hard}(D_G)$ computes this value, raised to the k'th power. Given a probability p, bag-TIDB $\mathcal{D}_G$ is defined as follows: every tuple $u \in V$ has probability $p_u = p$ and every tuple $e \in E$ has probability $p_e = 1$. In Sec. 4.3, we demonstrate that the expected multiplicity of tuples in the result of queries in the Q^k_{hard} family evaluated over variants of $\mathcal{D}_G$ for different values of p is related to $\#(G, ⚯ \cdots ⚯^k)$. Using hash join, an index on V, and projection pushdown, a deterministic database engine can evaluate Q^k_{hard} in no worse than $O(k\,|E|)$ time:

Lemma 4.3. *There exists a algorithm that computes Q^k_{hard} in time $O(k\,|E|)$ over D_G for any graph $G = (V, E)$ with $|V| = O(|E|)$.*

Comparison with set semantics hardness. We note that Dalvi and Suciu [13], show that under *set semantics* computing $Q^k_{hard} = Q_1$ is #P-hard (there is a technical subtlety, which we address in the next paragraph). Under bag semantics (which is our focus), one can compute the expected multiplicity of Q^k_{hard} for $k \in \{1, 2\}$ in $O(|E|)$ time.[5] As we show in accompanying technical report [6], the problem takes super linear time for $k \geq 3$.

We finally address a technical subtlety mentioned about the #P-hardness result of computing Q_1 under set semantics. The reduction in [13] starts off with a *bipartite* graph $\overline{G} = (V_1, V_2, \overline{E})$ where $\overline{E} \subseteq V_1 \times V_2$ where the hard query for set semantics is $\overline{Q}_1 \coloneqq \pi_\emptyset\left(V_1 \bowtie_{U=U_1} \overline{E} \bowtie_{U_2=U} V_2\right)$ (and $p = \frac{1}{2}$).[6] The reason this does not match Q_1 is because $V_1 \cap V_2 = \emptyset$. To show that Q_1 is also hard in set semantics, consider $G = (V \coloneqq V_1 \cup V_2, E)$ where for each $(i, j) \in \overline{E}$ both (i, j) and (j, i) are in E. We note that the marginal probability of computing Q_1 is the same as that of $\overline{Q}_1$.[7]

4.2 Reduced Polynomials

Recall that in Sec. 2.1 (specifically in Lem. 2.5), we showed that the final answer that we are after is the expectation of the corresponding lineage polynomial, i.e. $\mathbb{E}[\Phi]$. We now introduce a mechanical transformation that transforms a polynomial Φ into a "reduced" form $\widetilde{\Phi}$ that, for specific inputs, is exactly equivalent to the expectation $\mathbb{E}[\Phi]$. The insight behind this reduction is central to the arguments in Sec. 4.3 and algorithms in Sec. 5.

Consider a monomial $M \in \Phi$. We define the reduced monomial $\widetilde{M}$ by setting each *non-zero* exponent of a variable in M to 1. Let us consider the case where Φ is applied to a vector of random variables $\mathbf{X}$, such that each X_t independently has value 1 with probability p_t, i.e., Φ models a lineage

[5]The claimed result from $k = 1$ follows from the observation that the expected multipliciy for Q_1 is $p^2\,|E|$. The claim for $k = 2$, follows from the observation that in $O(|E|)$ time we can compute the number of copies of all two edge subgraphs in G.

[6]In this case the corresponding lineage polynomial is $\vee_{(i,j)\in\overline{E}} X_i \wedge X_j$ and we are interested in computing the probability that this polynomial evaluates to a 1. We note that there should be variables $X_{(i,j)}$ corresponding to the edges $(i, j) \in \overline{E}$ in the lineage polynomial. However, the reduction in [13] assigns edges in $\overline{E}$ a probability of 1, so we can drop the variables $X_{(i,j)}$ for the marginal probability computation.

[7]Note that the lineage polynomial for Q_1 is $\vee_{(i,j)\in E} X_i \wedge X_j = \vee_{(i,j)\in\overline{E}} \left(X_i \wedge X_j\right) \vee \left(X_j \wedge X_i\right) = \vee_{(i,j)\in\overline{E}} \left(X_i \wedge X_j\right)$, where the last equality follows from the fact for any Boolean variable b, we have $b \vee b = b$.

polynomial over a bag-TIDB. The key insight behind reduced polynomials is that, for an input monomial M, *the reduced monomial* $\widetilde{M}$*, evaluated on the probability vector* $\mathbf{p}$ *is exactly the expected value of the original monomial evaluated on the vector of variables.* (i.e., $\widetilde{M}(\mathbf{p}) = \mathbb{E}[M(\mathbf{X})]$) [6].

Example 4.4. *Consider the lineage monomial* $M(A, B) := A^2B^2$*. Because distinct variables are independent, we can push expectation through them yielding* $\mathbb{E}\left[A^2B^2\right] = \mathbb{E}\left[A^2\right]\mathbb{E}\left[B^2\right]$*. Since* $A, B \in \{0, 1\}$ *we can simplify to* $\mathbb{E}[A]\,\mathbb{E}[B]$ *by the fact that for any* $X \in \{0, 1\}, X^2 = X$*. We then have* $\mathbb{E}[A]\,\mathbb{E}[B] = p_A \cdot p_B$ *(where* p_A *and* p_B *denote* $Pr[A = 1]$ *and* $Pr[B = 1]$*). Thus,* $\mathbb{E}[M] = \mathbb{E}[A]\,\mathbb{E}[B] = \widetilde{M}(p_A, p_B)$*.*

Through linearity of expectation, the same argument applies to the entire polynomial: when Φ is evaluated over binary variables $\mathbf{X}$, then $\widetilde{\Phi}(\mathbf{p}) = \mathbb{E}[\Phi(\mathbf{X})]$. Moving forward, we will use the expectation and reduced polynomial interchangeably.

4.3 Hardness Reduction

We are now ready to present our main hardness results:

Lemma 4.5. *Let* $p_0, \ldots, p_{2k}$ *be* $2k + 1$ *distinct values in* $(0, 1]$*. and* $\Phi_G^k(\mathbf{X}) := Q_{hard}^k(\mathcal{D}_G)$ *where* $\mathcal{D}_G$ *is constructed as in Sec. 4.1. Then computing* $\widetilde{\Phi}_G^k(p_i, \ldots, p_i)$ *(for all* $i \in [2k + 1]$*) for arbitrary* $G = (V, E)$ *needs time* $\Omega(T_{match}(k, G))$*, if* $T_{match}(k, G) \geq \omega(|E|)$*.*

First, observe that $\widetilde{\Phi}_G^k(p, \ldots, p)$ must have the form $\sum_{i=0}^{2k} c_i p^i$, where only c_i depends on the specific graph for the following reason. Each monomial in Φ_G^1 is a product of exactly 2 variables (i.e., its degree is 2). It follows that each monomial of Φ_G^k is a product of $2k$ (non-distinct) variables (i.e., degree $2k$). By definition $\widetilde{\Phi}_G^k(\mathbf{X})$ sets every exponent $d_t > 1$ to $d_t = 1$, which means that $\deg(\widetilde{\Phi}_G^k) \leq \deg(\Phi_G^k) = 2k$. Thus, if we think of p as a variable, then $\widetilde{\Phi}_G^k(p, \ldots, p)$ is a polynomial of degree at most $2k$.

We observe that, by construction of $\widetilde{\Phi}_G^k$, each c_i is *exactly* the number of monomials in the sum-of-products expansion of $\widetilde{\Phi}_G^k(\mathbf{X})$ (its representation according to Definition 2.4) that are composed of i *distinct* variables. Given that we have $2k + 1$ distinct values of $\widetilde{\Phi}_G^k(p_i, \ldots, p_i)$ for $0 \leq i \leq 2k$, it follows that we have a linear system of the form $M \cdot c = b$, where the ith row of M is $\left(p_i^0, \ldots, p_i^{2k}\right)$, c is the coefficient vector $(c_0, \ldots, c_{2k})$, and b is the vector such that $b[i] = \widetilde{\Phi}_G^k(p_i, \ldots, p_i)$. This linear system can always be solved in $O(k^3)$ time [6] to compute values for each c_i.

We claim that c_{2k} is $k! \cdot \#(G, \text{⫶} \cdots \text{⫶}^k)$. This can be seen by looking at the expansion of the original factorized representation:

$$\Phi_G^k(\mathbf{X}) = \sum_{(i_1, j_1), \ldots, (i_k, j_k) \in E} X_{i_1} X_{j_1} \cdots X_{i_k} X_{j_k}$$

Only a unique k-matching, with k distinct (i_t, j_t) index pairs can contribute to the coefficient of p^{2k}. Moreover, every such distinct k-matching will be produced $k!$ times. Thus, given c_{2k}, we can obtain $\#(G, \text{⫶} \cdots \text{⫶}^k)$ in constant time.

4.4 Summary

Lemma 4.6. *Computing the expected multiplicity for* Q_{hard}^k *needs at least time* $\omega\left(f(k) \cdot (|E|)^c\right)$ *and* $|E|^{\Omega(k/\log k)}$ *under the complexity assumptions in Thm. 4.1 and Thm. 4.2 respectively.*

Assume that we can compute $Q_{hard}^k(\mathcal{D}_G)$ in time T. Thus, we can compute it $2k + 1$ times for different values of p in $O(kT)$ time. We can then obtain c_{2k} by solving a system of linear equations of size $O(k^2)$ in $O(k^3)$. We achieve a contradiction by showing that if T does not satisfy the claimed lower bounds, then $O\left(kT + k^3\right)$ is smaller than the lower bounds in Thm. 4.1 and Thm. 4.2.

Theorem 4.7. *Let $T_{det}(k, G)$ be the runtime of $Q^k_{hard}(D_G)$, and $T_{prob}(k, G)$ be the runtime of $Q^k_{hard}(\mathcal{D}_G)$ where D_G and $\mathcal{D}_G$ are as defined in Sec. 4.1. We have $T_{prob}(k, G) = \omega\left(T_{det}(k, G)\right)$.*

This result follows from Lem. 4.3 and Lem. 4.6. The proofs of our technical results can be found in our accompanying technical report [6]. In [6], we further relate the complexity of bag-PQP to a conservative model of query runtimes.

5 SAMPLING ALGORITHM

To summarize Sec. 4, there exists at least one family of queries and probabilistic database instances for which computing expected multiplicities for the result of a bag-TIDB query is necessarily (under standard complexity assumptions) a polynomial factor slower (in the size of the database) than computing multiplicities for the result of a corresponding deterministic query. In other words, *it is not possible for* bag-TIDB *databases to be competitive with deterministic databases if exact expectations are required.* Since exact multiplicities are out of the question, we turn to approximation.

Roadmap. We first analyze the bottlenecks of existing algorithms for bag-PQP in Sec. 5.1, isolating the theoretical bound to the problem of computing the expected count from a lineage circuit. In Sec. 5.2 and specifically Alg. 2, we establish a framework for approximating the expected count using the standard Horvitz and Thompson estimator. The balance of the section focuses on Alg. 3, our algorithm for computing approximate multiplicities in $Q(\mathcal{D})$. This algorithm first performs a pre-computation pass over the circuit (Sec. 5.3.2, Alg. 4), and then subsequently samples a series of monomials from the prepared circuit (Sec. 5.3.3, Alg. 5). We show that the runtime of Alg. 3 is bounded by $|\mathsf{C}|$, the size of the lineage circuit for a query result. As the asymptotic runtime to construct the lineage circuit for a deterministic query is proportional to that of constructing the query output (see Thm. 5.1), we adopt the $|\mathsf{C}|$ as our target complexity. *For ease of presentation, in this section we will assume that there is exactly one result tuple t.*

Algorithm 1 `expectSumOfProducts` $(Q, \mathbf{R}, \mathbf{p})$

```
Input: Query $Q$, relations $\mathbf{R}$, prob. $\mathbf{p} = (p_1, \ldots, p_n) \in [0,1]^n$
1: $\mathcal{S} \leftarrow$ buildPolynomial $(Q, \mathbf{R})$        ▷ e.g., ProvSQL [51]
2: acc $\leftarrow \sum_{M \in \mathcal{S}} \prod_{X \in \textsc{Distinct}(M)} p_X$
3: return acc
```

5.1 Exact Expectations

As discussed in Sec. 4.2, the expectation $\mathbb{E}[\Phi]$ (i.e., $\mathbb{E}[Q(\mathcal{D})(t)]$) is exactly $\widetilde{\Phi}(\mathbf{p})$. The naive algorithm for computing the exact expectations (Alg. 1), follows the derivation of $\widetilde{\Phi}$: it uses buildPolynomial to compute Φ represented in its *Sum-of-Monomials* (*SoM*) encoding (defined below), e.g., using ProvSQL [51]. Then for each monomial $M \in \Phi$, it obtains the *set* of distinct variables in $X \in M$ and computes the product of their respective probabilities (i.e., p_X). The correctness of Alg. 1 follows from linearity of expectation, and it runs in time linear in the number of monomials in the SoM encoding of Φ. However, using arithmetic circuits, polynomials can be encoded more concisely. As the following example shows, the use of circuits is necessary for achieving runtime competitive with deterministic query processing as just outputting a SoM encoding of lineage can have superlinear overhead over deterministic queries.

Example 5.1. *To understand why this is, consider the query $Q := \pi_\emptyset(R \times S)$ which evaluated deterministically returns the empty tuple $\langle\rangle$ with multiplicity $|R| \cdot |S|$. The deterministic version of this query can be evaluated in $O(|R| + |S|)$ by counting the number of tuples in R and S and then* *multiplying these counts. However, $\Phi = \sum_{r \in R,\ s \in S} X_r \cdot X_s$, the SoM lineage for the single result tuple, has $O(|R| \cdot |S|)$ monomials. Thus, just generating the lineage requires at least $O(|R| \cdot |S|)$ time. Using a circuit, Φ can be encoded using linear space through factorization: $(\sum_{r \in R} X_r) \cdot (\sum_{s \in S} X_s)$.*

As shown in the example above, using SoM (Alg. 1) we will incur superlinear overhead over deterministic queries just for constructing lineage. As a result, any approximation algorithm with linear overhead over deterministic querying requires the use of a more succinct representation of lineage, e.g., through arithmetic circuits.

Lineage Circuits. We now formally define arithmetic circuits and demonstrate that circuits for lineage polynomials can be constructed with linear overhead over deterministic query processing.

Definition 5.2 (Circuit). *A circuit C is a Directed Acyclic Graph (DAG) with source gates (in degree of 0) drawn from $\mathbf{X} = (X_t)_{t \in D}$ and one sink gate for each result tuple. Internal gates have binary input and are either sum (+) or product (×) gates. C_L and C_R denote the left and right inputs of C, respectively.*

Where it is clear from context, we allow C to denote a gate rather than an entire circuit. Operators that replicate tuple lineages (e.g., joins) may reuse the *same* gate for each replica, sharing (rather than copying) the sub-circuit for multiple output tuples. Using a model for the runtime of deterministic queries where the runtime of a query Q over a bag-PDB $\mathcal{D}$ with bounding deterministic database $\overline{D}$ in this model is denoted as $T_{det}(Q, \overline{D})$, we demonstrate in [6] that circuits can be generated efficiently. Intuitively, this is possible by reusing partial circuits during circuit construction. Each intermediate result tuple will be associated with a pointer to the root node of a circuit. Relational operators construct circuits for result tuples by adding new nodes and connecting them to the roots of existing circuits. For instance, the circuit for a join result tuple is constructed by creating a multiplication node and connecting it to the root nodes of the circuits for the two input tuples that were joined. In this manner, a constant runtime and space overhead is paid for each tuple that is processed by an operator.

Theorem 5.1 (See [6]). *There exists an algorithm* buildCircuit *that for any query Q, bounding database $\overline{D}$, and tuple t outputs a lineage circuit $\mathsf{C}_{t,Q,D}$ with logarithmic depth such that its size and the runtime of* buildCircuit *are both bounded by $O(T_{det}(Q, \overline{D}))$.*

Although its authors do not prove its runtime, ProvSQL [51] realizes buildCircuit with this bound. A formal proof can be found in our accompanying technical report [6], but intuitively, there is a one-to-one mapping between each intermediate tuple in a join plan, and the gate representing the tuple (recalling that gates may be re-used).

5.2 Approximate$\widetilde{\Phi}$

Alg. 1 is computing a sum of a large number of monomials. A natural strategy for approximating the sum of N items is to compute the sum of s uniform samples and rescale by $\frac{N}{\mathsf{s}}$. The algorithm, Approximate$\widetilde{\Phi}$ (Alg. 2) implements a variant proposed by Horvitz and Thompson [30] that allows for biased sampling. We consider biased sampling, as we will use an approach based on biased sampling for FastPDB in Sec. 6.2. Approximate$\widetilde{\Phi}$ is parameterized by a function genSamples. We discuss implementations of this function below, but we require that it generates a collection of sampled monomials M, each annotated with the probability of drawing that specific monomial $Pr[\mathsf{M}_i]$. If monomials are sampled uniformly, then $Pr[\mathsf{M}_i] = \frac{1}{|Q(\overline{\mathbf{R}})|} = \frac{1}{N}$. The following result follows from the standard analysis of the Horvitz and Thompson estimator:

Algorithm 2 $\text{APPROXIMATE}\widetilde{\Phi}(Q, \mathbf{R}, \mathbf{p}, \mathsf{s})$

```
Input: Query $Q$, Relations $\mathbf{R}$, #samples $\mathsf{s}$, $\mathbf{p} = (p_1, \ldots, p_n) \in [0,1]^n$
1: $\mathcal{S} \leftarrow \text{GENSAMPLES}(Q, \mathbf{R}, \mathbf{p}, \mathsf{s})$        ▷ $\mathcal{S} = \{\langle \mathsf{M}_i, Pr[\mathsf{M}_i]\rangle\}$
2: $\mathsf{acc} \leftarrow \sum_{(\mathsf{M}, Pr[\mathsf{M}]) \in \mathcal{S}} \prod_{X \in \text{DISTINCT}(\mathsf{M})} p_X \cdot \frac{1}{Pr[\mathsf{M}]}$
3: return $(\mathsf{acc} \cdot \frac{1}{\mathsf{s}})$
```

Algorithm 3 $\text{GENSAMPLES}_{\text{C}}(Q, \mathbf{R}, \mathsf{s})$

```
Input: Query $Q$, relations $\mathbf{R}$, #samples $\mathsf{s}$
1: $\mathcal{S} \leftarrow []$
2: $\mathsf{C} \leftarrow \text{BUILDCIRCUIT}(Q, \mathbf{R}, \mathsf{s})$        ▷ Thm. 5.1 / ProvSQL
3: $(\mathsf{size}, \mathsf{C}_p) \leftarrow \text{PREPARE}(\mathsf{C})$        ▷ Alg. 4
4: for i in s do
5:     $\mathcal{S} \leftarrow \mathcal{S} \circ \left[\text{SAMPLEMONOMIAL}(\mathsf{C}_p), \frac{1}{\mathsf{size}}\right]$        ▷ Alg. 5
6: return $\mathcal{S}$
```

PROPOSITION 5.2. *Assume the* GENSAMPLES $(Q, \mathbf{R}, \mathbf{p}, \mathsf{s})$ *has the following property: For each monomial M in the SoM encoding of $\Phi[Q, \mathbf{R}, t]$, the probability of M being sampled is strictly greater than* 0*, i.e. $Pr[M] > 0$. Then for every $s \geq 1$:*

$$\mathbb{E}\left[\text{APPROXIMATE}\widetilde{\Phi}(Q, \mathbf{R}, \mathbf{p}, \mathsf{s})\right] = \texttt{expectSumOfProducts}(Q, \mathbf{R}, \mathbf{p}).$$

The only remaining task is to design an appropriate GENSAMPLES. For $\text{APPROXIMATE}\widetilde{\Phi}$ to be an efficient approximation algorithm: (i) GENSAMPLES needs to run in time $O(|\text{BUILDCIRCUIT}(Q, \mathbf{R}, \mathsf{s})|)$ (Alg. 3) and (ii) $Pr[M]$ needs to be as close to $\frac{1}{|Q(\mathbf{R})|}$ as possible.

5.3 Circuit based GENSAMPLES

We realized a circuit based implementation of GENSAMPLES, named name $\text{GENSAMPLES}_{\text{C}}$, that achieves this bound. The algorithm is summarized in Alg. 3. $\text{GENSAMPLES}_{\text{C}}$ first produces a circuit of the output lineage polynomial via a call to BUILDCIRCUIT (from Thm. 5.1, and implemented in ProvSQL [51]). Because the circuit size and the cost of its creation are bounded by $O(|\mathsf{C}|) \leq O\left(T_{det}(Q, \overline{D})\right)$, the circuit creation and subsequent approximation algorithm combine to form an end-to-end approximation of the expected multiplicity of tuples in $Q(D)$ that meets our complexity target. Before discussing $\text{GENSAMPLES}_{\text{C}}$, we introduce the notion of the *monomial expansion* of a circuit, which we denote by $\mathsf{E}(\mathsf{C})$. $\mathsf{E}(\mathsf{C})$ encodes the SoM representation of C as a list of monomials with repetitions for monomials that have a coefficient larger than 1. Each monomial is represented as a bag of variables, e.g., $X^2Y^3 = \{\!|X, X, Y, Y, Y|\!\}$.

EXAMPLE 5.3. *The SoM encoding of the polynomial of the circuit of Fig. 5 is $X^2W^2 + XWYZ + XWYZ + WY^2Z$. In other words,*

$$\mathsf{E}(\mathsf{C}) = [\{\!|X, X, W, W|\!\}, \{\!|X, W, Y, Z|\!\}, \{\!|X, W, Y, Z|\!\}, \{\!|W, Y, Y, Z|\!\}].$$

Essentially, algorithm $\text{GENSAMPLES}_{\text{C}}$ samples monomials uniformly from $\mathsf{E}(\mathsf{C})$ as follows. First, BUILDCIRCUIT is used to build the circuit to sample from. Second, a call to PREPARE augments C (i.e. C_p) with sampling weights to enable uniform sampling. Then, using C_p, sampling takes place via calls to SAMPLEMONOMIAL, which has the following guarantee (see [6] for the proof):

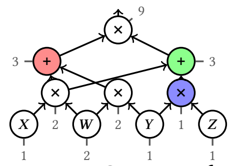


Fig. 5. Circuit for $(XW+WY)(XW+YZ)$.

Lemma 5.4. SampleMonomial *returns in time* $O(|\mathsf{C}|)$ *an* independent uniform monomial *from* $\mathsf{E}(\mathsf{C})$.

5.3.1 Guarantee on Approximate$\widetilde{\Phi}$. We now have all the pieces in place to present our final guarantee on Approximate$\widetilde{\Phi}$. We get the following bounds (see [6] for the proof):

Theorem 5.3. *For every query* Q*, relations* $\mathbf{R}$ *and probability vector* $\mathbf{p}$, Approximate$\widetilde{\Phi}$ $\left(Q, \mathbf{R}, \mathbf{p}, \left\lceil \frac{2\log\frac{2}{\delta}}{\epsilon^2} \right\rceil\right)$ *using* genSamples$_{\mathsf{C}}$ *(in time* $O_{\epsilon,\delta}(|\mathsf{C}|) \leq O_{\epsilon,\delta}\left(T_{det}(Q, \overline{D})\right)$*) outputs an estimate* acc *of* $\widetilde{\Phi}(p_1, \ldots, p_n)$ *such that (where* C *is as in Line 2 of* genSamples*):*

$$Pr\left[\left|\mathsf{acc} - \widetilde{\Phi}(p_1, \ldots, p_n)\right| \geq \epsilon \cdot \mathsf{C}(1, \ldots, 1)\right] \leq \delta.$$

Proof Sketch. From Proposition 5.2, $\mathbb{E}[\mathsf{acc}] = \widetilde{\Phi}(p_1, \ldots, p_n)$. Lem. 5.4 then allows us to use Chernoff's bound, which gives the claimed bound (note that by definition $\mathsf{C}(1, \ldots, 1)$ is the number of monomials in the SoM encoding of the polynomial represented by C). The claim on the run time follows from Lem. 5.4 and Thm. 5.1.

Finally, we note that if there exists a constant $p_0 > 0$ such that $p_i \geq p_0$ for all $i \in n$, i.e., all probabilities are larger than p_0, then we can replace the guarantee of Thm. 5.3 with a multiplicative $1 \pm \epsilon$ approximation guarantee:

$$Pr\left[\left|\mathsf{acc} - \widetilde{\Phi}(p_1, \ldots, p_n)\right| \geq \epsilon \cdot \widetilde{\Phi}(p_1, \ldots, p_n)\right] \leq \delta.$$

The above follows by noting that under the condition $p_i \geq p_0$ we have $\widetilde{\Phi}(p_1, \ldots, p_n) \geq \Omega_{p_0,n}(\mathsf{C}(1, \ldots, 1))$ and so adjusting ϵ by an appropriate constant in Thm. 5.3 gives the bound. Next, we explain algorithms Prepare and SampleMonomial.

Algorithm 4 Prepare (C)

```
Input: C: Circuit
Output: C: Circuit annotated with Lweight, Rweight, partial.
Output: sum ∈ ℕ
 1: for g ∈ TopOrd(C) do                      ▷ TopOrd(·) is C in topological order
 2:     if g is a var gate then
 3:         g.partial ⟵ 1
 4:     else if g is a × gate then
 5:         g.partial ⟵ $g_L$.partial × $g_R$.partial
 6:     else if g is a + gate then
 7:         g.partial ⟵ $g_L$.partial + $g_R$.partial
 8:         g.Lweight ⟵ $\frac{g_L.\text{partial}}{g.\text{partial}}$
 9:         g.Rweight ⟵ $\frac{g_R.\text{partial}}{g.\text{partial}}$
10:     sum ⟵ g.partial
11: return (sum, C)
```

5.3.2 Prepare. Alg. 4 (which implements Prepare) computes, for each gate C of a circuit, the the number of monomials in the corresponding sub-circuit (i.e., $\mathsf{C}(1, \ldots, 1)$). Samples in SampleMonomial are taken specifically at + gates, so we also pre-compute shorthand values for these cells (Lweight, Rweight) that correspond to the proportion of the number of monomials under each of the gate's children. The algorithm's runtime is $O(|\mathsf{C}|)$, modulo a log factor for the topological sort, as it visits each gate (g) exactly once and performs constant work for each.

Algorithm 5 SampleMonomial (C)

```
Input: C: Circuit annotated by Alg. 4
Output: List of Variables
1: if C's root is a + gate then
2:     $C_{samp}$ ← Sample $C_L$ (w.p. C.Lweight) or $C_R$ (w.p. C.Rweight)
3:     return SampleMonomial($C_{samp}$)
4: else if C's root is a × gate then
5:     v ← SampleMonomial(C.left)
6:     return v ∘ SampleMonomial(C.right)
7: else if C's root is a var gate then
8:     return [C.val]
```

5.3.3 SampleMonomial. Alg. 5 implements SampleMonomial and utilizes a recursive definition of E(C) (see [6]). Concretely, a + gate combines two sets of monomials; thus, the sampling process picks, at random, a set to explore with a probability proportional to the sum of the number of monomials computed in the Prepare step. For a × gate, the process samples a monomial uniformly from each of the gate's children and computes their product. The source gates each define a variable $X \in \mathbf{X}$. SampleMonomial may visit the same gate multiple times (e.g., if multiple gates share a parent). Our proof [6] of its $O(|C|)$ runtime is based on the observation that the degree of a *lineage* circuit is bounded by the *join width* $(k-1)$ of the query that created it. Thus, the number of × gates that SampleMonomial encounters in any single invocation is likewise bounded by k. Because only one branch of each + gate is taken, we can bound the total runtime of SampleMonomial by $O(k|C|)$.

6 IMPLEMENTATION

In this section, we present two implementations of Approximate$\widetilde{\Phi}$. First, we present a direct implementation using genSamples$_C$ (Alg. 3). While this implementation can guarantee low variance, per our experiments (Sec. 7), constructing the circuit that the first implementation relies on can have high runtime overhead. This high overhead motivates our second implementation, which uses approximate query processing (WanderJoin [35], specifically) to "push down" lineage sampling into query evaluation.

6.1 Approximate$\widetilde{\Phi}$ with genSamples$_C$

To implement the exact algorithm, we use ProvSQL [51], an extension for PostgreSQL that instruments query execution to produce lineage circuits for result tuples. ProvSQL persists a DAG structure with immutable nodes for every tuple that the Postgres instance materializes. This architecture allows multiple tuples, and even multiple relations in the system to share circuit nodes. We implement Approximate$\widetilde{\Phi}$ with genSamples$_C$ as a UDF that takes a ProvSQL circuit node as input, and returns the expected multiplicity.

Prepare. Prepare traverses the input circuit, precomputing the cumulative weights for each inner node of the circuit. We extended the ProvSQL circuit structure to store these weights. One challenge is that ProvSQL implements inner nodes as n-ary, as opposed to the binary nodes we assume in our theoretical development. Although this allows ProvSQL to store a circuit without introducing a logarithmic overhead to its size, the $O(|C|)$ runtime of SampleMonomial now depends on sampling in constant time from a + node's n children. To achieve this bound, the Prepare algorithm additionally generates a Walker Alias Table [56], a data structure that allows

$$\llbracket R \rrbracket_G = R_{aug} \qquad \llbracket \sigma_\theta(Q) \rrbracket_G = \sigma_\theta \, \llbracket Q \rrbracket_G$$

$$\llbracket \pi_A(Q) \rrbracket_G = \pi_{A,\mathsf{vsch}(Q)} \, \llbracket Q \rrbracket_G \qquad \llbracket Q_1 \bowtie Q_2 \rrbracket_G = \llbracket Q_1 \rrbracket_G \bowtie \llbracket Q_2 \rrbracket_G$$

$$\begin{aligned}\llbracket Q_1 \uplus Q_2 \rrbracket_G = {} & \pi_{sch(Q_1),\mathsf{vsch}(Q_1),\mathsf{vsch}(Q_2)\leftarrow \mathbb{1}} \, \llbracket Q_1 \rrbracket_G \\ & \uplus \, \pi_{sch(Q_2),\mathsf{vsch}(Q_1)\leftarrow \mathbb{1},\mathsf{vsch}(Q_2)} \, \llbracket Q_2 \rrbracket_G\end{aligned}$$

$$\mathsf{vsch}(R) = \{\mathcal{X}_R, \mathcal{P}_R\} \qquad \mathsf{vsch}(\sigma_\theta(Q)) = \mathsf{vsch}(\pi_A(Q)) = \mathsf{vsch}(Q)$$

$$\mathsf{vsch}(Q_1 \bowtie Q_2) = \mathsf{vsch}(Q_1 \uplus Q_2) = \mathsf{vsch}(Q_1) \cup \mathsf{vsch}(Q_2)$$

Fig. 6. GProM [4] semantics for propagating tuple annotations. $\mathsf{vsch}(\cdot)$, as defined above denotes the set of variable attributes in a relation (R) or query (Q).

weighted sampling from a fixed set of elements in constant time. An alias table for n elements can be created in $O(n)$ time (i.e., constant time per-child), preserving the $O(|C|)$ runtime of Prepare.

SampleMonomial. SampleMonomial is implemented exactly as Alg. 5, modulo the alias table as described above. Samples are computed using the Mersenne Twister [39] with a uniform real distribution in $(0, 1)$ for each internal + gate.

Approximate$\widetilde{\Phi}$. Approximate$\widetilde{\Phi}$ is a UDF that generates an expected multiplicity from a lineage circuit by: (i) invoking Prepare on the input circuit, and (ii) repeatedly invoking SampleMonomial as needed per Thm. 5.3 to achieve a user-provided error bound.

6.2 FastPDB: Sampling Pushdown with AQP

We observe that the $O(|\mathsf{C}|)$ runtime bound for SampleMonomial is due to the degenerate case when the depth of C, denoted by depth (C), is of the same order as $|\mathsf{C}|$. For typical lineage circuits C where depth $(\mathsf{C}) \ll |\mathsf{C}|$ (e.g., for queries not exclusively over small tables), SampleMonomial *visits only a small fraction of the overall circuit.* This suggests a more efficient approach: sampling directly from the source data without fully materializing C. In the following, we first explain how to generate SoM lineage in SQL using standard provenance techniques and then how to compute exact expected multiplicities from the lineage. We observe that the query computing the exact multiplicities for an input SPJ (select-project-join) query generated in this way is an SPJ query followed by a single aggregation.[8] Such queries can be approximated using existing AQP techniques. The net result is an approximation algorithm for bag-PQP that avoids generation of lineage circuits.

Generating SoM Lineage in SQL. In FastPDB we store a bag-TIDBs $\mathcal{D} = (\Omega, \mathcal{P})$ by augmenting every table R in the database to get R_{aug} with two additional annotation columns: $\mathcal{X}_R$ stores an identifier for the variable X associated with an input tuple and p stores $Pr[X = 1]$. The instance of this database contains exactly the tuples from $\overline{D}$. Using existing provenance systems, we can generate a SoM representation of $\Phi[Q, \overline{D}, t]$ $(\mathbf{X})$ for each result tuple t of a query Q over a bag-PDB $\mathcal{D}$ by rewriting Q. In the result of the rewritten query, the lineage of a single result tuple is encoded as multiple result tuples — one for each monomial. The tuple representing a monomial includes both the variables appearing in the monomial as well as their probabilities. For instance, for a query $\pi_A(R \bowtie S)$ over relations $sch(R) = \langle A \rangle$ and $sch(S) = \langle B \rangle$, the schema of the rewritten query is $\langle A, \mathcal{X}_R, \mathcal{P}_R, \mathcal{X}_S, \mathcal{P}_S \rangle$. A result tuple $\langle a \rangle$ with lineage $\Phi = X_1X_3 + X_1X_5$ where $p_1 = 0.3$, $p_3 = 0.8$, and $p_5 = 1.0$ would be encoded as two tuples $\langle a, 1, 0.3, 3, 0.8 \rangle$ and $\langle a, 1, 0.3, 5, 1.0 \rangle$.

We summarize a simplified form of GProM's rewrite semantics [4] for $\mathcal{RA}^+$ in Fig. 6. Here $\mathbb{1}$ denotes a tuple encoding an appropriate number of copies of a dummy variable $X_\perp$ with probability

[8]Using linearity of expectation, union operations can be pulled through the aggregation resulting in a union of join-aggregate queries followed by a final aggregation to combine the expectations produced by the individual join-aggregate queries.

1.0, e.g., for a subquery over relations R and S with annotation attributes $\langle \mathcal{X}_R, \mathcal{P}_R, \mathcal{X}_S, \mathcal{P}_S \rangle$ we get $\mathbb{1} = \langle X_\perp, 1.0, X_\perp, 1.0 \rangle$.

The rewrite $[\![Q]\!]_G$, rewrites Q to include a set of additional annotation attributes $\mathtt{vsch}(Q)$ that contain the $\mathcal{X}_R$ and $\mathcal{P}_R$ attributes from each relation R accessed by the query. In other words, for any row of $Q(D)$, the attributes in $\mathtt{vsch}(Q)$ (excluding those with values of $\mathbb{1}$) collectively define one monomial M of the corresponding tuple's lineage and store the probabilities of variables used in M.

Computing Exact Expectations in SQL. Recall that the expected multiplicities of a result tuple t for a query evaluated over a bag-TIDB can be computed from SoM lineage by summing up the expectation of each monomial M which in turn can be computed by multiplying the probability of all distinct variables in M (i.e., ignoring exponents). Given the result of the rewritten query $[\![Q]\!]_G$, this can be implemented in SQL using `sum` and a function:

$$\textsc{MonomialProb}\left((X, p_X)_{X \in M}\right) := \prod_{X \in \textsc{Distinct}(M)} p_X$$

Using this function we can computed expected multiplicities as shown below.

```
SELECT sch(Q).*, sum(MonomialProb(vsch(Q).*))
FROM ⟦Q⟧_G GROUP BY sch(Q).*
```

Approximating Multiplicities with AQP. As in Alg. 2, this query $[\![Q]\!]_G$ may be approximated. Approximating sums over join outputs is a primary focus of AQP. As first observed by Olken [43], the key challenge in sampling from join results is sparsity: two tuples selected uniformly at random are unlikely to join with each other. One approach to AQP called Wander Join [35] addresses this limitation by (i) sampling first from one input relation, and then (ii) using pre-built indices (that include weights for computing sampling probabilities like those computed by Alg. 4) to iteratively sample only joinable tuples from the remaining inputs. We refer the interested reader to [35] for a detailed overview of the algorithm. For our purposes, it suffices that WanderJoin meets the criteria of genSamples: (i) it generates a stream of tuples sampled from the result of arbitrary non-aggregate joins, and (ii) each tuple includes the probability with which it was drawn. The important advantage of using AQP is that it enables us to compute expected multiplicities without having to materialize lineage.

Implementation. FastPDB's implementation stores bag-TIDBs in PostgreSQL. User queries are rewritten with GProM to pass through variable identifiers and probabilities, and evaluated by XDB [35], an implementation of WanderJoin in PostgreSQL to produce approximated expected multiplicities.

7 EXPERIMENTS

7.1 Setup

Our experiments were conducted on a 12th generation Intel Core i5-1240P version 6.154.3 4400 MHz with 8 core processors, 32 GB of RAM, and 500 GB SSD. We compare the following systems (i) **FastPDB**: as described in Sec. 6; (ii) **ProvSQL-e**: `expectSumOfProducts` (Alg. 1) implemented as a ProvSQL [51]-compatible UDF that takes a lineage circuit constructed by ProvSQL as input and computes the exact expected multiplicity; (iii) **ProvSQL-a**: $\textsc{Approximate}\widetilde{\Phi}$ (Alg. 2) with genSamples $_{\mathrm{C}}$ (Alg. 3) implemented as a ProvSQL-compatible UDF that takes the lineage circuits constructed by PostSQL as input; (iv) **GProM-e**: `expectSumOfProducts` (Alg. 1) implemented using GProM [4] to compute a sum-of-products representation of the lineage formula and propagate probabilities and uses `sum` to calculate the expected multiplicity of each output tuple based

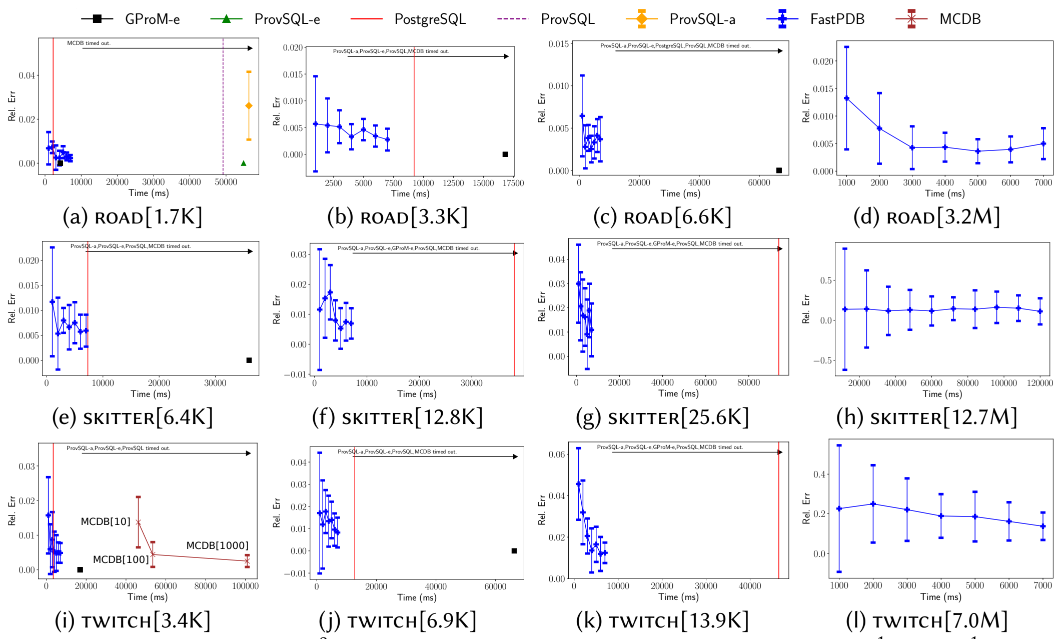


Fig. 7. Canonical hard query Q^2_{hard} on the graph datasets, varying dataset size from $\frac{1}{2000}$th to $\frac{1}{250}$ of the original dataset size.

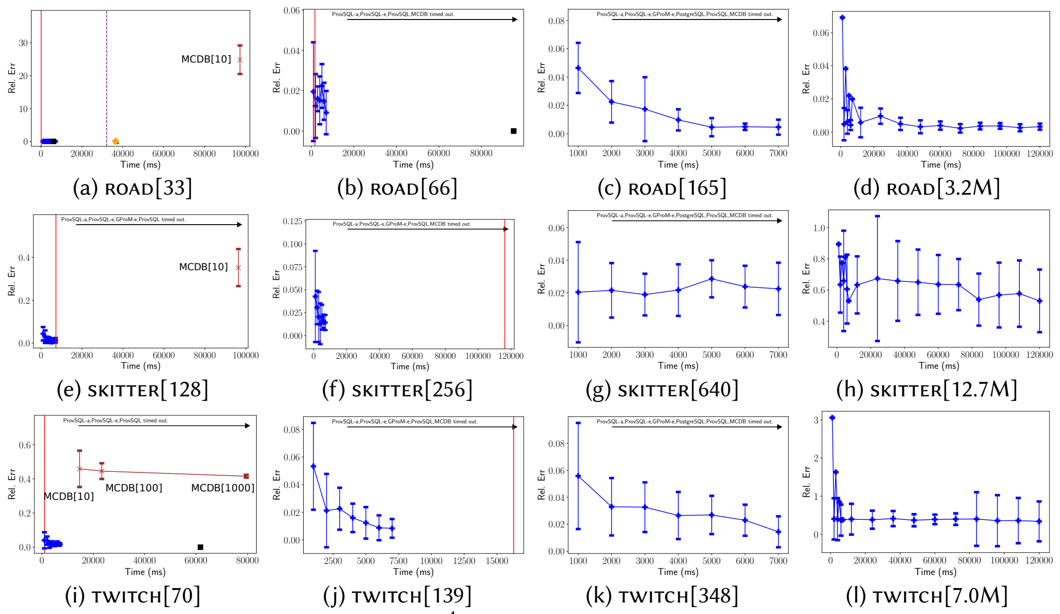
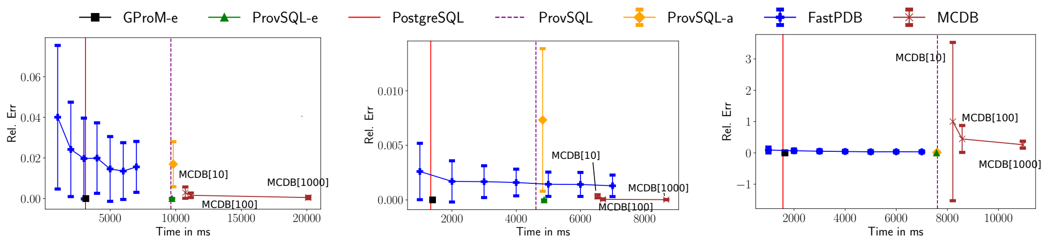


Fig. 8. Canonical hard query Q^4_{hard} on the graph datasets, varying dataset size.

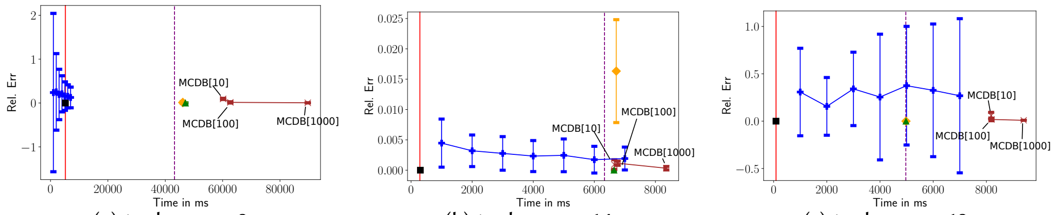
on this SoM representations using the propagated probabilities for input tuples; (v) **MCDB**: A re-implementation of MCDB [32]'s tuple bundles included in Pip [33][9], the expected multiplicity is computed based on a set of sampled worlds, MCDB[n] denotes the version which generates n worlds. As comparison points, we also include the deterministic query runtime of **PostgreSQL**

[9]MCDB's original implementation is not generally available.



(a) urel-1 (b) urel-2 (c) urel-3

Fig. 9. Relative approximation error and runtime comparing FASTPDB against competitors for pdbench [3] queries.



(a) tpch query 9 (b) tpch query 14 (c) tpch query 19

Fig. 10. Hierarchical TPC-H queries Q9, Q14, Q19 (without aggregation) on SF1.

and the time taken by **ProvSQL** to construct a lineage circuit without calculating any expected multiplicities.

FASTPDB is implemented on XDB [36], a probabilistic database built using PostgreSQL 9.4.2. Experiments with GProM and PostgreSQL use the same version of Postgres. ProvSQL requires more recent features, and so these experiments use PostgreSQL 15.1. Pip is not compatible with versions of PostgreSQL more recent than 8.4.22, and so experiments with Pip's MCDB implementation use this version. In all cases, each system is supplied with an appropriate index on each of the attributes appearing in selection predicates across all test queries. For ProvSQL-a, we set $\epsilon = 0.05$ and $\delta = 0.97$ that is with 97% probability the estimated multiplicity will be within 5% of the real value. For MCDB, we use 10, 100, and 1000 samples and report results for each (MCDB[10], MCDB[100], MCDB[1000]). As XDB is the only system capable of anytime estimates, we allow FASTPDB to run for 7 seconds and report predictions available after each second. All experiments were repeated 10 times. We report median runtimes and median and standard deviation for relative estimation error for systems that produce approximate results.

Datasets and Queries. pdbench: PDBENCH [3] extends the TPC-H benchmark data generator [54] to introduce attribute-level uncertainty into records. Except where noted, test data was generated with a scale factor of 1.0 (i.e., one possible world consumes $\sim$ 1.0 GB in uncompressed CSV form), using the default uncertainty ratio. The generated database uses about 3.2 GB of space in uncompressed CSV form due to the addition of uncertainty. Experiments use the standard PDBENCH test queries [3], identified as **urel-1**, **urel-2**, and **urel-3**. These are versions of TPC-H queries 3, 6, and 7 respectively, modified by removing aggregation. We modify these queries only to compute multiplicities rather than existentials. **road, skitter, twitch:** We also use several graph datasets from https://snap.stanford.edu/data/index.html. The Texas road network dataset (**road**) has $\sim 1.3M$ nodes and $\sim 1.9M$ edges. The **skitter** dataset is the internet topology graph generated from traceroutes run daily in 2005 ($\sim 1.7M$ nodes an $\sim 11M$ edges). The twitch dataset (**twitch**) is a social networks of Twitch users ($\sim 168K$ nodes and $\sim 6.8M$ edges). For each of these datasets we choose random probability values for each of the tuples sampled from a uniform distribution over $[0, 1]$. We created smaller versions of these datasets by randomly selecting joining tuples from the original tables, preserving the relative size difference between the node and edge tables. We use ROAD[N] (SKITTER[N], and TWITCH[N]) to denote the scaled down version with $\sim n$ rows. We evaluate variants of the canonical hard query Q^k_{hard} (see Sec. 4.1) for $k \in \{2, 4\}$ to stress test systems on a hard query.

7.2 Hard Queries on Graph Datasets

We use two versions of the canonical hard query Q^k_{hard} on the graph dataset to evaluate the runtime and accuracy of systems for queries that are known to be hard. We use a timeout of 120 seconds and vary the dataset size. The results for Q^2_{hard} are shown in Fig. 7. While most systems are able to compute approximate expected multiplicities for the smallest dataset (ROAD[1.6K]), constructing the linage circuit is expensive (50 seconds) and sampling from the circuit is not beneficial, taking more time than exactly computing the result. FastPDB computes a more accurate estimate than ProvSQL-a in less time. While GProM-e can still produce a result for datasets with less than 10K tuples, FastPDB is the only one to scale to large datasets with millions of tuples and produces estimates that are within a few percent of the correct result the smaller datasets. For the full datasets, for Q^2_{hard} the error is around 10% for skitter and twitch which are more skewed and around 1% for road. This is impressive given that Q^2_{hard}'s lineage for the full skitter has $\sim 1.2 \times 10^{14}$. MCDB, the only competitor that does not produce full lineage, suffers from having to compute large join results. Even MCDB[10] timed out for the smallest dataset.

Fig. 8 shows the results for Q^4_{hard}. As this query has extremely large lineage formulas ($\sim 1.5 \times 10^{28}$ monomials for the full skitter dataset), we use smaller versions of the datasets to compare against the baselines. Most systems only finish execution for ROAD[32] (32 tuples) and FastPDB is the only system that produces results for the full datasets, reaching less than 1% relative errors within seconds for the smaller datasets. For skitter and twitch the relative error is large for the full dataset. This is to be expected given that the size of the lineage. In summary, these experiments clearly demonstrate the need to avoid materialization of lineage circuits.

7.3 pdbench and TPC-H Queries

Fig. 9 shows the total runtimes of all systems for the pdbench (probabilistic) TPC-H data & queries.

urel-1 (modified TPC-H Q3). urel-1 is a 3-way foreign-key join over the customer, orders, and lineitem tables. FastPDB achieves a relative error below 0.5% within 1s, about five times faster than PostgreSQL deterministic execution that does not compute any expectations. Because urel-1 uses exclusively foreign-key joins, the number of monomials is linear in the data size. As a result, GProM's SoM lineage encoding introduces minimal overhead compared to PostgreSQL. ProvSQL-a and ProvSQL-e's runtimes are dominated by lineage construction costs, which roughly doubles their runtime relative to deterministic PostgreSQL. For this simple query, MCDB shines, achieving a nearly perfect answer with only 10 samples, albeit still at a significant higher cost than both exact approaches.

urel-2 (modified TPC-H Q6). This is a single-table query with multiple selections and low selectivity, making the query a poor fit for systems like XDB. Still, FastPDB achieves $\sim 0.3\%$ relative error within 1s, twice as fast as deterministic query evaluation. As in urel-1, the number of monomials is linear in the data size, and GProM performs competitively with PostgreSQL. While MCDB achieves accurate estimates, it is again slower than both exact approaches.

urel-3 (modified TPC-H Q7). This is a 5-way cyclic join. After 1s, FastPDB achieves 0.5% relative error. Due to the greater number of joins, the majority of tuples in a sample generated by MCDB do not join, leading to poor estimation accuracy. Lineage construction continues to dominate the runtime for the ProvSQL variants. .

Hierarchical TPC-H Queries. We also ran experiments with TPC-H queries 9, 14, and 19 without aggregation which are supported by all of the compared systems. Note that these are hierarchical queries, which are computational easy even for set probabilistic databases. Fig. 10 shows the results of this experiment. For these queries, deterministic query evaluation is fast, requiring less than a second for all three queries. GProM-e performs well for all queries. Nonetheless, FastPDB produces accurate estimates within or slightly above the deterministic runtime of these queries. Q19 is challenging for XDB and, thus, also FastPDB, as it contains a disjunctive selection condition that is very selective: only ~ 0.001 of the join results fulfill the condition. Thus, most sampled monomials (join results) have to be rejected as they do not fulfill the selection condition resulting in inaccurate estimates. As the final provenance circuit is very small, ProvSQL-a produces very accurate estimates and the overhead of ProvSQL-e over just constructing the circuit (ProvSQL-e) is neglectable. MCDB is accurate, but slower than the exact approaches.

Sampling Rate vs Variance. XDB's [36] sampling is biased (e.g., towards source tuples with high join fan-outs). Although the expectation computation corrects for this bias (see Alg. 2), this does result in an increase in variance, especially for datasets with highly skewed distributions of the number of join partners such as skitter. In practice this means that ProvSQL-a has significantly higher sample efficiency, i.e., it often produces an order of magnitude more accurate results for the same amount of samples. However, FastPDB typically already converges on an accurate estimate in the time it takes ProvSQL-a to construct the lineage circuit, negating this disadvantage. We show a more detailed analysis in [6].

Summary. Overall, FastPDB benefits from its anytime approximation approach based on XDB that avoids materializing the full lineage formulas for a query. For many settings, FastPDB can produce accurate results before the competitors have even finished materializing the lineage. This is critical for queries with large lineage formulas such as Q^k_{hard} where often FastPDB is the only system that is able to generate estimated expected multiplicities within the allocated time and easily scales to dataset sizes several orders of magnitude larger than any of the baselines. Computing estimates over a full lineage circuit produced by ProvSQL (ProvSQL-a) is typically not beneficial as the runtime savings compared to exact computation are either neglectable or, even worse, this is slower than exact computation (ProvSQL-e).

Even though GProM-e does use the SMB representation of polynomials, GProM-e outperforms ProvSQL-e on the settings presented here, because of its use of simpler relational operations instead of constructing circuits as UDTs. Using sampling with MCDB can have very large error for some queries and is in general slower than lineage-based solutions. FastPDB is the most stable in terms of runtime being the only approach that can finish all queries under all settings of consideration, and in most cases produces estimates faster than alternative approaches. The only exception is TPC-H Q19 where the extremely low selectivity results in many samples being rejected. For such queries it may be beneficial to consider a hybrid approach that uses the database to estimate the selectivity of a query and falls back to use an exact approach or ProvSQL-a if the selectivity is very low.

As there is an obvious trade-off between latency and approximation errors and the choice between an exact approach and FastPDB, the users of FastPDB may want to accurately estimate the total cost given a desired error bound and decide between an exact and approximate approach. Furthermore, the frequency for reporting updated predictions in FastPDB should ideally be adjusted based on the total runtime of a query, i.e., whether it is a hard query or not. Unfortunately, as we have not established a dichotomy for computing expected multiplicities, we do not know how to determine whether a query will be efficient when evaluated using an exact approach. Furthermore, accurate estimation of statistical information about the data to compute pessimistic or conservative confidence intervals is hard [28, 38]. Leveraging the join order optimization with trial samples in XDB, it is possible to obtain a rough estimation of the cost under the best join order to reach a certain error bound. However, such approach is also susceptible to extremely low selectivity of predicates, resulting in inaccurate cost estimation, which we leave for future work to further improve.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we investigate computing expected multiplicities over bag probabilistic databases for $\mathcal{RA}^+$ from both a theoretical as well as systems perspective. We show using fine-grained complexity that computing exact expectations, while in PTIME, is provably less efficient than deterministic query evaluation. While we present an approximation scheme that returns a $(1 \pm \epsilon)$-approximation with high probability with linear overhead for the class of hard queries we identify, in practice a main overhead is the generation of lineage. We overcome this bottleneck by exploiting approximate query processing for sampling monomials from a tuple's lineage without having to materialize it. The resulting anytime approximation scheme works well for both hard and "easy" queries scaling to several orders of magnitude larger databases than both exact and approximate competitors while often achieving equal approximation in less time. This paper opens several avenues of follow-up work. Given that computing expected multiplicities is equivalent to computing expectations of group-by count queries, we conjecture that our approach could be extended to support other aggregates. We also leave open the question of whether this approach generalizes beyond the bag-TIDB model and $\mathcal{RA}^+$ language we explored here — for example, whether the approach generalizes to uncertainty expressed at the level of attributes. Similarly, for many queries, we observe that GProm-e, an evaluation strategy that directly generates monomials for query results, performs surprisingly well. However, for hard queries, only FastPDB can answer these queries in reasonable time. If a dichotomy for $\mathcal{RA}^+$ on bag-probabilistic databases exists and testing whether a query is PTIME is efficient, then a hybrid approach would be effective that uses GProM-e for easy queries and uses FastPDB for hard queries.. Similar approaches applied to linear algebra could be used to develop high-performance, uncertainty-aware neural network inference.

ACKNOWLEDGMENTS

This work was supported by grants NSF Award IIS-1956149, NSF Award IIS-2420577 and NSF Award IIS-2420691.

REFERENCES

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB*. 1151–1154.

[2] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. 2015. Provenance Circuits for Trees and Treelike Instances. In *ICALP*. 56–68.

[3] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. 2008. Fast and Simple Relational Processing of Uncertain Data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, USA, 983–992. https://doi.org/10.1109/ICDE.2008.4497507

[4] Bahareh Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Eng. Bull.* 41, 1 (2018), 51–62.

[5] Subi Arumugam, Ravi Jampani, Luis Leopoldo Perez, Fei Xu, Christopher M. Jermaine, and Peter J. Haas. 2010. MCDB-R: Risk Analysis in the Database. *Proc. VLDB Endow.* 3, 1 (2010), 782–793.

[6] ANONYMOUS AUTHORS. 2023. Probabilistic Databases Don't Have to Be Slow. https://anonymous.4open.science/r/2024_Bag_PDBs_Reproducibility-FA3F/tech_report.pdf

[7] Omar Benjelloun, Anish Das Sarma, Chris Hayworth, and Jennifer Widom. 2006. An Introduction to ULDBs and the Trio System. *IEEE Data Eng. Bull.* 29, 1 (2006), 5–16.

[8] George Beskales, Ihab F. Ilyas, and Lukasz Golab. 2010. Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *Proc. VLDB Endow.* 3, 1 (2010), 197–207.
[9] Radu Curticapean. 2013. Counting Matchings of Size k Is W[1]-Hard. In *ICALP*, Vol. 7965. 352–363.
[10] Radu Curticapean and Dániel Marx. 2014. Complexity of Counting Subgraphs: Only the Boundedness of the Vertex-Cover Number Counts. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS '14)*. IEEE Computer Society, USA, 130–139. https://doi.org/10.1109/FOCS.2014.22
[11] Nilesh Dalvi and Dan Suciu. 2007. The Dichotomy of Conjunctive Queries on Probabilistic Structures. In *PODS*. 293–302.
[12] N. Dalvi and D. Suciu. 2007. Efficient query evaluation on probabilistic databases. *VLDB* 16, 4 (2007), 544.
[13] Nilesh Dalvi and Dan Suciu. 2012. The dichotomy of probabilistic inference for unions of conjunctive queries. *JACM* 59, 6 (2012), 30.
[14] Maarten Van den Heuvel, Peter Ivanov, Wolfgang Gatterbauer, Floris Geerts, and Martin Theobald. 2019. Anytime Approximation in Probabilistic Databases via Scaled Dissociations. In *SIGMOD*. 1295–1312.
[15] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. On Join Sampling and the Hardness of Combinatorial Output-Sensitive Join Algorithms. In *PODS*. ACM, 99–111.
[16] Su Feng, Boris Glavic, Aaron Huber, and Oliver Kennedy. 2021. Efficient Uncertainty Tracking for Complex Queries with Attribute-level Bounds. In *SIGMOD*.
[17] Su Feng, Boris Glavic, and Oliver Kennedy. 2023. Efficient Approximation of Certain and Possible Answers for Ranking and Window Queries over Uncertain Data. *Proc. VLDB Endow.* 16, 6 (2023), 1346–1358.
[18] Su Feng, Aaron Huber, Boris Glavic, and Oliver Kennedy. 2019. Uncertainty Annotated Databases - A Lightweight Approach for Approximating Certain Answers. In *SIGMOD*.
[19] Robert Fink, Jiewen Huang, and Dan Olteanu. 2013. Anytime approximation in probabilistic databases. *VLDBJ* 22, 6 (2013), 823–848.
[20] Robert Fink and Dan Olteanu. 2011. On the optimal approximation of queries using tractable propositional languages. In *ICDT*. 174–185.
[21] Robert Fink and Dan Olteanu. 2016. Dichotomies for Queries with Negation in Probabilistic Databases. *TODS* 41, 1 (2016), 4:1–4:47.
[22] Jörg Flum and Martin Grohe. 2002. The Parameterized Complexity of Counting Problems. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS '02)*. IEEE Computer Society, USA, 538.
[23] Wolfgang Gatterbauer and Dan Suciu. 2017. Dissociation and Propagation for Approximate Lifted Inference With Standard Relational Database Management Systems. *VLDB J.* 26, 1 (2017), 5–30.
[24] Erich Grädel, Yuri Gurevich, and Colin Hirsch. 1998. The Complexity of Query Reliability. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 227–234. https://doi.org/10.1145/275487.295124
[25] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. 31–40.
[26] Martin Grohe, Peter Lindner, and Christoph Standke. 2023. Probabilistic Query Evaluation with Bag Semantics. In *ICDT*, Floris Geerts and Brecht Vandevoort (Eds.), Vol. 255. 20:1–20:19.
[27] Paolo Guagliardo and Leonid Libkin. 2017. Correctness of SQL Queries on Databases with Nulls. *SIGMOD Rec.* 46, 3 (2017), 5–16.
[28] P.J. Haas. 1997. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings. Ninth International Conference on Scientific and Statistical Database Management (Cat. No.97TB100150)*. 51–62. https://doi.org/10.1109/SSDM.1997.621151
[29] P. J. Haas and J. M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD*. 287–298.
[30] D. G. Horvitz and D. J. Thompson. 1952. A Generalization of Sampling Without Replacement from a Finite Universe. *J. Amer. Statist. Assoc.* 47, 260 (1952), 663–685. https://doi.org/10.1080/01621459.1952.10483446 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1952.10483446
[31] R. Impagliazzo and R. Paturi. 1999. Complexity of k-SAT. In *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference) (Cat.No.99CB36317)*. 237–240. https://doi.org/10.1109/CCC.1999.766282
[32] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. 2008. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD*.
[33] Oliver Kennedy and Christoph Koch. 2010. PIP: A Database System for Great and Small Expectations. In *ICDE*.
[34] Kyoungmin Kim, Jaehyun Ha, George Fletcher, and Wook-Shin Han. 2023. Guaranteeing the Õ(AGM/OUT) Runtime for Uniform Sampling and Size Estimation over Joins. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Floris Geerts, Hung Q. Ngo, and Stavros Sintos (Eds.). ACM, 113–125. https://doi.org/10.1145/3584372.3588676

[35] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 615–629. https://doi.org/10.1145/2882903.2915235

[36] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2017. Wander Join and XDB: Online Aggregation via Random Walks. *SIGMOD Rec.* 46, 1 (May 2017), 33–40. https://doi.org/10.1145/3093754.3093763

[37] Kaiyu Li and Guoliang Li. 2018. Approximate Query Processing: What is New and Where to Go? - A Survey on Approximate Query Processing. *Data Sci. Eng.* 3, 4 (2018), 379–397.

[38] Stephen Macke, Maryam Aliakbarpour, Ilias Diakonikolas, Aditya Parameswaran, and Ronitt Rubinfeld. 2021. Rapid Approximate Aggregation with Distribution-Sensitive Interval Guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1703–1714. https://doi.org/10.1109/ICDE51399.2021.00150

[39] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30.

[40] Barzan Mozafari. 2017. Approximate query engines: Commercial challenges and research opportunities. In *SIGMOD*. 521–524.

[41] Barzan Mozafari and Ning Niu. 2015. A Handbook for Building an Approximate Query Engine. *IEEE Data Eng. Bull.* 38, 3 (2015), 3–29.

[42] Raghotham Murthy, Robert Ikeda, and Jennifer Widom. 2011. Making Aggregation Work in Uncertain and Probabilistic Databases. *IEEE Trans. Knowl. Data Eng.* 23, 8 (2011), 1261–1273.

[43] Frank Olken and Doron Rotem. 1986. Simple Random Sampling from Relational Databases. In *VLDB*. Morgan Kaufmann, 160–169.

[44] Dan Olteanu, Jiewen Huang, and Christoph Koch. 2010. Approximate confidence computation in probabilistic databases. In *ICDE*. 145–156.

[45] Laurel J. Orr, Magdalena Balazinska, and Dan Suciu. 2020. EntropyDB: a probabilistic approach to approximate query processing. *VLDB J.* 29, 1 (2020), 539–567.

[46] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (2018), 719–732.

[47] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201.

[48] Christopher Ré, Nilesh N. Dalvi, and Dan Suciu. 2007. Efficient Top-k Query Evaluation on Probabilistic Data. In *ICDE*. 886–895.

[49] Christopher Ré and Dan Suciu. 2009. The trichotomy of HAVING queries on a probabilistic database. *VLDBJ* 18, 5 (2009), 1091–1116.

[50] Christopher De Sa, Alexander Ratner, Christopher Ré, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang. 2017. Incremental knowledge base construction using DeepDive. *VLDB J.* 26, 1 (2017), 81–105.

[51] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proc. VLDB Endow.* 11, 12 (aug 2018), 2034–2037. https://doi.org/10.14778/3229863.3236253

[52] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Morgan & Claypool Publishers.

[53] Bruhathi Sundarmurthy, Paraschos Koutris, Willis Lang, Jeffrey Naughton, and Val Tannen. 2017. m-tables: Representing Missing Data. In *ICDT*, Vol. 68.

[54] The Transaction Processing Performance Council. [n. d.]. The TPC-H Benchmark. http://www.tpc.org/tpch/.

[55] Guy Van den Broeck and Dan Suciu. 2017. Query Processing on Probabilistic Data: A Survey. *Foundations and Trends in Databases* (2017).

[56] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (1977), 253–256.

[57] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, Dieter Gawlick, and Oliver Kennedy. 2015. Lenses: An On-Demand Approach to ETL. *PVLDB* 8, 12 (2015), 1578–1589.

Received July 2024; revised September 2024; accepted November 2024