Probabilistic Databases under Updates: Boolean Query Evaluation and Ranked Enumeration

Christoph Berkholz
Humboldt-Universität zu Berlin
Institut für Informatik
Berlin, Germany
berkholz@informatik.hu-berlin.de

Maximilian Merz
Humboldt-Universität zu Berlin
Institut für Informatik
Berlin, Germany
maximilian.merz@informatik.hu-berlin.de

ABSTRACT

We consider tuple-independent probabilistic databases in a dynamic setting, where tuples can be inserted or deleted. In this context we are interested in efficient data structures for maintaining the query result of Boolean as well as non-Boolean queries.

For Boolean queries, we show how the known lifted inference rules can be made dynamic, so that they support single-tuple updates with only a constant number of arithmetic operations. As a consequence, we obtain that the probability of every safe UCQ can be maintained with constant update time.

For non-Boolean queries, our task is to enumerate all result tuples ranked by their probability. We develop lifted inference rules for non-Boolean queries, and, based on these rules, provide a dynamic data structure that allows both log-time updates and ranked enumeration with logarithmic delay. As an application, we identify a fragment of non-repeating conjunctive queries that supports log-time updates as well as log-delay ranked enumeration. This characterisation is tight under the OMv-conjecture.

ACM Reference Format:
Christoph Berkholz and Maximilian Merz. 2021. Probabilistic Databases under Updates: Boolean Query Evaluation and Ranked Enumeration. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3452021.3458326

1 INTRODUCTION

In *dynamic query evaluation*, a fixed query is evaluated against a changing database that might be updated, e. g., by inserting or deleting tuples. To avoid evaluating the query from scratch after every update, it is necessary to represent the query result in a dynamic data structure that allows efficient updates as well as quick access.

The drawback of traditional approaches in *incremental view maintenance*, which maintain a full materialisation of the query result, is that the update time is always at least as large as the difference between the old and the new result. In recent years, several methods and data structures have been proposed to generate compressed dynamic representations that allow quick access to the current query result. These data structures support efficient (sublinear, polylogarithmic, or even constant-time) single-tuple updates, even if the change of the represented query result is way larger. Examples include the evaluation of conjunctive queries [4, 5, 15, 16, 19, 20], first-order queries on restricted classes of databases [6, 21], and MSO on tree-structured data [1–3, 22, 27, 28].

In this paper we focus on dynamic evaluation of first-order queries on (finite, tuple-independent) probabilistic databases [9], a formal framework that was introduced to model uncertainty. Probabilistic databases extend relational databases by annotating every ground tuple t in the database with a parameter $p(t) \in [0, 1]$ that expresses its certainty. Formally, they define a distribution where the parameters $p(t)$ are the probabilities of the independent events that a tuple is present in the database. Evaluating a *Boolean* query on a probabilistic database means computing the probability that the query evaluates to true, given the distribution on the ground tuples. In general, this can be a much harder task than evaluating the query on a standard relational database. For instance, $\exists x \exists y\, Sx \wedge Exy \wedge Ty$ can be answered in linear time on non-probabilistic databases, but computing its probability on probabilistic databases is #P-hard [9].

One way to answer tractable first-order queries is to apply *lifted inference rules*.[1] These rules allow to rewrite and decompose a query so that its probability can be computed from simpler subqueries. An easy example of a lifted inference rule is "independent and": If two Boolean queries φ_1 and φ_2 are defined over disjoint sets of relations, then their probabilities are independent and hence we can compute the probability of $\varphi_1 \wedge \varphi_2$ by $\mathbb{P}(\varphi_1 \wedge \varphi_2) = \mathbb{P}(\varphi_1) \cdot \mathbb{P}(\varphi_2)$. It turns out that this approach is quite powerful: the well-known dichotomy theorem of [10] states that any Boolean UCQ is either *safe* and can be evaluated in polynomial time by recursively applying lifted inference rules, or it is unsafe and query evaluation is #P-hard.

We start the investigation of dynamic query evaluation on probabilistic databases by considering *Boolean* first-order queries. Our first result (Theorem 3.8) is that if a query can be solved using a set of known lifted inference rules, then there is a dynamic data structure that allows to evaluate the query with constant update time. A corollary of this result is that the probability of any safe UCQ can be maintained in constant time upon single-tuple updates. Hence, by the dichotomy theorem [10], every UCQ either supports constant update time or requires super-polynomial update time (assuming P ≠ #P).

The result of a *non-Boolean* query $\varphi(x_1, \ldots, x_k)$ on a probabilistic database is a k-ary relation that contains every potential output tuple together with its probability $p > 0$. Clearly, any such query can be solved by answering, for all n^k potential output tuples $(a_1, \ldots, a_k)$, the Boolean query obtained by replacing the free variables by constants $a_1, \ldots, a_k$. Thus, its polynomial-time tractability

[1] This approach is also called *extensional query evaluation*, e. g., in [25].

PODS '21, June 20–25, 2021, Virtual Event, China

reduces to tractability of the underlying Boolean formula. In the dynamic setting, maintaining the materialised output relation under updates might be too expensive. Moreover, it might be that all n^k output tuples appear with a non-zero probability in the query result, but that most of those tuples have a very small probability and hence are of less interest to the user. In particular, this is regularly the case in *open-world probabilistic databases* [7], where every ground tuple that is *not* present in the database is implicitly assigned a small probability. For these reasons, users (of probabilistic database engines) typically want to order output tuples by their probability. This motivates our search for a succinct representation of the query result, that permits efficient updates to the database and allows the user to access high-probability result tuples first.

Our main result identifies a class of first-order queries where this is indeed possible. We provide a dynamic data structure that represents the query result and supports the following:

- When inserting, deleting, or changing the probability of a ground tuple in the current database D, the data structure can be updated in $O(\log \|D\|)$ time.
- Upon request, the data structure immediately reports the result tuple with highest probability and allows to enumerate all result tuples ranked by their probability with $O(\log \|D\|)$ delay between two outputs.

To achieve this result, we establish lifted inference rules for non-Boolean queries that allow a similar recursive reasoning as in the Boolean case. They have the following form: If a query φ can be decomposed in a certain way into subqueries, and if all of those subqueries support efficient dynamic ranked enumeration, then φ also supports efficient dynamic ranked enumeration. For *non-repeating* (also known as *self-join-free*) conjunctive queries we show that our class of queries that are tractable in the above sense is tight: Either, log-time updates and log-delay enumeration are possible, or any dynamic algorithm with $O(\|D\|^{0.5-\epsilon})$ update time and $O(\|D\|^{0.5-\epsilon})$ delay (even unranked) enumeration would violate the OMv-conjecture. A similar dichotomy for dynamic ranked evaluation of non-repeating UCQs remains open. Although it is possible for deterministic (non-probabilistic) databases to characterise those non-repeating UCQs that allow efficient enumeration under updates (assuming the OMv-conjecture), it turns out that this is much harder for probability-ranked enumeration. We illustrate the difficulties that arise by reducing from an open problem in computational geometry in Section 4.3.

1.1 Related work

Query evaluation on tuple-independent probabilistic databases has received a lot attention in the last decade and we refer the reader to the surveys [33] and [36] for a gentle introduction to the area and an overview of recent work. To the best of our knowledge, there is no previous work that studies *dynamic* query evaluation on probabilistic databases.

Olteanu and Wen [29] considered the task of ranking output tuples of non-repeating conjunctive queries in the static setting. Besides not supporting updates, their setting of ranked enumeration differs from ours in the following two aspects. First, their algorithm does not provide guarantees on the delay and computes a full materialisation of the query result. Second, in their framework it is only required to report tuples in a ranked order, without computing the actual probabilities associated to the output tuples. Because of this, their polynomial time fragment (the *head-hierarchical* queries) also includes intractable queries for which it is #P-complete to compute the probability of an output tuple.

Ranked enumeration with guarantees on the delay has recently been investigated in two independent works by Deep and Koutris [11] and Tziavelis et al. [35] for evaluating conjunctive queries on deterministic databases in the static setting. Both papers define their ranked enumeration algorithms using general ranking functions, and for certain queries, their generalizations allow probabilistic ranking: For example, we can use the selective semiring $([0,1], \max, \cdot, 0, 1)$ with the algorithm from [35] to evaluate acyclic non-repeating projection-free conjunctive queries against tuple-independent probabilistic databases with linear preprocessing time and logarithmic delay. However, neither framework seems to generalise to first-order quantification on probabilistic databases or to disjunctions in queries. In addition, both methods support only *static* evaluation of conjunctive queries and also for those classes of queries for which it is known that, under certain algorithmic assumptions, no efficient *dynamic* query evaluation algorithm exists [4].

Ranked enumeration generalises *top-k query evaluation* [17], where only the best k result tuples have to be reported. For this reason, ranked enumeration has also been called "any-k" [35] as it enables the user to decide how many result tuples she wants to receive. We briefly discuss the algorithmic implications of *aborting* the enumeration of our data structure in Section 4.1.5.

On a high level, our algorithm takes a query φ and a probabilistic database D and computes a data structure that represents the query result and provides efficient updates as well as ranked enumeration. In a similar vein, Jha and Suciu [18] studied different static representations for Boolean probabilistic UCQs. Their data structures are based on well-known representation formats from *knowledge compilation*, in particular (with increasing generality): read-once expressions, OBDDs, FBDDs, and d-DNNFs. Monet and Olteanu [26] discussed the use of *deterministic decomposable circuits* (d-Ds) in this setting, which are even more general as they extend d-DNNFs by allowing negations inside the circuit. We believe that with a bit of work, all these representations can actually be made dynamic, so that they can be modified efficiently when a ground tuple in the database changes. However, efficient compilation strategies into these representation formats are only known for a subclass of safe UCQs and it has been conjectured (called the "Intensional vs Extensional Conjecture" in [25]) that not all safe UCQs can be efficiently compiled. This is in contrast to our data structure, which can represent all safe UCQs.

2 PRELIMINARIES

We denote by $\mathbb{N} = \mathbb{N}_0$ the set of non-negative integers, by $\mathbb{N}_1$ the set of positive integers, and we let $[n] := \{1, ..., n\}$ for all $n \in \mathbb{N}_1$. We will denote by ϵ the empty function $\epsilon : \emptyset \to \emptyset$.

Probabilistic Databases. The *domain* **dom** is a finite set of *constants*. We usually denote constants by the letter a and its subscripted variants. A *schema* is a finite set σ of relation symbols R, where each $R \in \sigma$ has a fix *arity* $\mathrm{ar}(R) \in \mathbb{N}_1$. We fix a schema

$\sigma = \{R_1, ..., R_s\}$ and arities $\text{ar}(R_i)$ for $i \in [s]$. A *relational database* D of schema σ (or σ-*db* D) is of the form $D = (\mathbf{dom}; R_1^D, ..., R_s^D)$, where R_i^D is a finite subset of $\mathbf{dom}^{\text{ar}(R_i)}$. The *active domain* $\mathbf{adom}(D)$ is the smallest subset A of $\mathbf{dom}$ such that $R_i^D \subseteq A^{\text{ar}(R_i)}$ for all $i \in [s]$. For any $R \in \sigma$, we call an element $(a_1, ..., a_r) \in R^D$ *ground tuple* of the database instance and denote it by $Ra_1 \cdots a_r$. We let $\text{GT}_\sigma := \{Ra_1 \cdots a_r : R \in \sigma, a_i \in \mathbf{dom}\}$ be the set of all possible ground tuples over a schema σ and domain $\mathbf{dom}$.

A *(finite, tuple-independent) probabilistic database* of schema σ is a σ-db that associates with every ground tuple t contained in the database a probability $p(t) \in [0, 1]$. Unless stated otherwise, we stick to the usual *closed-world semantics* and implicitly assume that any ground tuple that is not present in the probabilistic database has probability 0. Every probabilistic database of schema σ over the domain $\mathbf{dom}$ defines a probability distribution over all *possible worlds* $\mathcal{W}$, which are all relational databases over the same schema σ and domain $\mathbf{dom}$. The probability $\mathbb{P}_D(\mathcal{W})$ of $\mathcal{W}$ is the probability of the event that we end up with the database $\mathcal{W}$ when including every possible ground tuple $t \in \text{GT}_\sigma$ independently with probability $p(t)$, that is,

$$\mathbb{P}_D(\mathcal{W}) := \prod_{t \in \mathcal{W}} p(t) \cdot \prod_{t \in \text{GT}_\sigma \setminus \mathcal{W}} (1 - p(t)).$$

In Section 4.1.6 we show that all our results extend to *open-world semantics* [7], where every ground tuple on $\mathbf{dom}$ that is not contained in the database receives some small positive probability. The *size* $\|D\|$ of a probabilistic database D is the size of a reasonable encoding of the probabilistic database and is linear in the number of ground tuples that it contains.

Queries. We fix a countably infinite set of variables $\mathbf{var}$, which are usually denoted be x, y, z and variants thereof. A (first-order) formula is built from *atoms* $\varphi = Rx_1 \dots x_r$ of *type* $R \in \sigma$ using negation $\neg\varphi$, conjunction $\varphi_1 \wedge \varphi_2$, disjunction $\varphi_1 \vee \varphi_2$, universal quantification $\forall x\, \varphi$ and existential quantification $\exists x\, \varphi$. As usual, we denote by $\text{free}(\varphi)$ the *free variables* of φ and assume that every quantified variable occurs free in its scope. A formula φ is a *sentence* if $\text{free}(\varphi) = \emptyset$. The *schema of a formula* $\sigma(\varphi) \subseteq \sigma$ is the set of relation symbols that occur in φ. A formula is *non-repeating*, if for every $R \in \sigma$ there exists at most one atom of type R in φ. A *valuation* β is a partial mapping from $\mathbf{var}$ to $\mathbf{dom}$. For a valuation β that is undefined on variable x, we let $\beta_{x \mapsto a} := \beta \cup \{x \mapsto a\}$. Moreover, we let $\beta\restriction_V := \{x \mapsto \beta(x) : x \in \text{dom}(\beta) \cap V\} \subseteq \beta$ be the restriction of β to V and call β' and extension of β if $\beta' \supseteq \beta$.

For a formula φ, a valuation $\beta : \text{free}(\varphi) \to \mathbf{dom}$ and a probabilistic database D, we denote by $\mathbb{P}_{D,\beta}(\varphi)$ the probability that for a randomly chosen possible world $\mathcal{W}$ of D, the first-order interpretation $(\mathcal{W}, \beta)$ satisfies φ:

$$\mathbb{P}_{D,\beta}(\varphi) := \sum_{(\mathcal{W},\beta) \models \varphi} \mathbb{P}_D(\mathcal{W}).$$

If φ is a sentence we may write $\mathbb{P}_D(\varphi)$ instead of $\mathbb{P}_{D,\epsilon}(\varphi)$ and if the database is clear from the context, we will often omit the subscript D. A *(first-order) query* is a pair (φ, W), where φ is a first-order formula and $W \subseteq \text{free}(\varphi)$ is a set of *output variables*. The *result* of a query (φ, W) under a probabilistic database D and a valuation $\beta : \text{free}(\varphi) \setminus W \to \mathbf{dom}$ is the set

$$[\![(\varphi, W)]\!]^{D,\beta} := \big\{(\tau, p) \mid \tau : W \to \mathbf{dom}; p = \mathbb{P}_{D,\tau \uplus \beta}(\varphi) > 0\big\}.$$

If all free variables are output variables, we may just write φ instead of $(\varphi, \text{free}(\varphi))$ and $[\![\varphi]\!]^D$ instead of $[\![\varphi]\!]^{D,\epsilon}$. A query (φ, W) is *Boolean* if it has no output variables, i. e., $W = \emptyset$. By $C(\varphi, W) := \text{free}(\varphi) \setminus W$ we denote the set of variables that need to be fixed by some $\beta : C \to \mathbf{dom}$ when evaluating (φ, W) and we write just C if the query is clear from the context.

Complexity analysis and machine model. In this paper we study *data complexity*, where the query is treated as fixed. Hence, our complexity analysis may hide arbitrary terms that depend on the query size. We use a variant of the *Random Access Machine* (RAM) model with uniform cost measure. In particular, this model enables the construction of lookup tables of polynomial size that can be queried in constant time. Moreover, all probability values (which might get exponentially small in n) can be stored in constant space and arithmetic operations can be done in constant time. The same is true for storing and comparing constants from $\mathbf{dom}$.

Updates. Our data structures support *single-tuple updates*, that is, inserting or deleting a ground tuple $t \in \text{GT}_\sigma$ and a probability $p(t)$ in a probabilistic database D over $\mathbf{dom}$. While we do not consider updates that change the domain $\mathbf{dom}$, our results extend to these kind of updates for UCQs. For first-order queries with $\forall$ and/or $\neg$ an extension to efficient domain changes is not immediate and we leave this for future work.

3 DYNAMIC EVALUATION OF BOOLEAN UCQS

On a conceptual level there are two different approaches of computing the probability of a Boolean query in the static setting. The first one, called *grounded inference*, is to transform the first-order query into an equivalent propositional formula over a set of ground atoms on a fixed domain. This propositional formula, known as *grounding*, *lineage*, or *provenance*, can be obtained by replacing the first-order quantifiers $\exists x\, \varphi$ and $\forall x\, \varphi$ by disjunctions $\bigvee_{a \in \mathbf{dom}} \varphi\frac{a}{x}$ and conjunctions $\bigwedge_{a \in \mathbf{dom}} \varphi\frac{a}{x}$ over all elements in the domain. Here, $\varphi\frac{a}{x}$ is obtained from φ by replacing every free occurrence of x by the constant a and hence the resulting formula is a proposition statement over all ground atoms. The probability of the statement can then be computed using weighted model counting for propositional logic.

A different paradigm is *lifted inference* where one tries to do most of the reasoning on the first-order level. Lifted inference rules allow to decompose or rewrite the first-order query so that its probability can be computed from the probability of simpler (sub-)queries. This approach is particularly appealing to us, as it also helps to design efficient update strategies. In this section we first revisit known lifted inference rules and then show how they can be used to support efficient database updates. For an in-depth discussion of these rules and comparison to grounded inference we refer the reader to the excellent survey [36].

3.1 Lifted Query Evaluation

In this subsection we make the reader familiar with known concepts and definitions for query evaluation on *static* probabilistic databases that can also be found in, e. g., [10, 34, 36]. One thing that can always be done is to rewrite a first-order query φ into an equivalent query

φ' using syntactic transformations. It is clear that this does not change the semantics of a query and hence maintains the probability value when evaluated on a probabilistic database. We may formalize this as the following rule.

(≡) $\mathbb{P}_\beta(\varphi) = \mathbb{P}_\beta(\varphi')$, if $\varphi' \equiv \varphi$ and
- φ' is obtained from φ by renaming a bound variable or
- every atom in φ' appears in φ.

Note that $\varphi' \equiv \varphi$ is a semantic property that is undecidable to check for full first-order logic. Hence, by "applying" this rule we mean applying any computable equivalence transformation. Instead of providing a list of all possible syntactic transformations, we decided to formulate this rule as general as possible. However, for technical reasons we added an additional syntactic property, which ensures that no new atoms are introduced in the formula. This property will be used in the proof of Theorem 3.8. The reader is invited to check that most textbook equivalence transformations for first-order logic satisfy this constraint. Moreover, the well known algorithm of Sagiv and Yannakakis [32] for minimising UCQs, i. e., first-order formulas without negation and universal quantification, is also captured by this rule.

With the (≡) rule we can always transform φ into negation normal form, where negations only occur in front of atoms. If φ is a (negated) atom, we can immediately look up the probability $\mathbb{P}_\beta(\varphi)$ from the database:

(atom) $\quad \mathbb{P}_\beta(Rx_1 \cdots x_r) = p(R\beta(x_1) \cdots \beta(x_r))$

(¬atom) $\quad \mathbb{P}_\beta(\neg Rx_1 \cdots x_r) = 1 - p(R\beta(x_1) \cdots \beta(x_r))$

The next two rules rely on the independence of subqueries. Formally, two queries φ_1 and φ_2 are *independent* if for any probabilistic database D and any valuation $\beta\colon \mathrm{free}(\varphi_1) \cup \mathrm{free}(\varphi_2) \to \mathbf{dom}$ the two probabilities $\mathbb{P}_{D,\beta\restriction_{\mathrm{free}(\varphi_1)}}(\varphi_1)$ and $\mathbb{P}_{D,\beta\restriction_{\mathrm{free}(\varphi_2)}}(\varphi_2)$ are statistically independent.

(ind-∧) $\mathbb{P}_\beta(\varphi_1 \wedge \varphi_2) = \mathbb{P}_{\beta\restriction_{\mathrm{free}(\varphi_1)}}(\varphi_1) \cdot \mathbb{P}_{\beta\restriction_{\mathrm{free}(\varphi_2)}}(\varphi_2)$
if φ_1 and φ_2 are independent.

(ind-∨) $\mathbb{P}_\beta(\varphi_1 \vee \varphi_2) = 1-\big(1-\mathbb{P}_{\beta\restriction_{\mathrm{free}(\varphi_1)}}(\varphi_1)\big)\cdot\big(1-\mathbb{P}_{\beta\restriction_{\mathrm{free}(\varphi_2)}}(\varphi_2)\big)$
if φ_1 and φ_2 are independent.

When applying these rules one has to provide an algorithm that takes care that the subformulas are indeed independent. As it is the case for logical equivalence, testing independence is undecidable and one has to rely on syntactic criteria that guarantee independence. A very simple property, which in fact suffices for UCQs, is that φ_1 and φ_2 are independent if they do not share common relation symbols.

To eliminate the quantifiers $\forall x\, \varphi$ and $\exists x\, \varphi$ using lifted inference rules one has to rely on the independence of all $n = |\mathbf{dom}|$ probabilities $\mathbb{P}_{\beta_{x\mapsto a}}(\varphi)$. Here we first provide a syntactic characterisation that guarantees independence and use this property in the definition of the rules below.

Definition 3.1. A variable x is a *separator variable* in a first-order formula φ if

(1) x occurs in every atom of φ and
(2) for every relational symbol R, there exists a number $i_R \in [\mathrm{ar}(R)]$, such that every atom in φ that refers to R contains x on position i_R.

It is not hard to see that the probabilities $\mathbb{P}_{\beta_{x\mapsto a}}(\varphi)$ obtained by binding a separator variable to a domain element a depend on disjoint sets of ground tuples in the database and hence are mutually independent. This leads to the following two lifted inference rules.

(ind-∀) $\mathbb{P}_\beta(\forall x\, \varphi) = \prod_{a\in\mathbf{dom}} \mathbb{P}_{\beta_{x\mapsto a}}(\varphi)$
if x is a separator variable in φ.

(ind-∃) $\mathbb{P}_\beta(\exists x\, \varphi) = 1 - \prod_{a\in\mathbf{dom}} (1 - \mathbb{P}_{\beta_{x\mapsto a}}(\varphi))$
if x is a separator variable in φ.

If independence cannot be guaranteed for ∨ or ∧, then applying inclusion-exclusion might be helpful in order to switch from ∧ to ∨ or vice versa.

(incl-excl1) $\mathbb{P}_\beta(\bigwedge_{i\in[k]} \varphi_i) = \sum_{\emptyset\neq I\subseteq[k]} (-1)^{|I|+1} \cdot \mathbb{P}_\beta(\bigvee_{i\in I} \varphi_i)$

(incl-excl2) $\mathbb{P}_\beta(\bigvee_{i\in[k]} \varphi_i) = \sum_{\emptyset\neq I\subseteq[k]} (-1)^{|I|+1} \cdot \mathbb{P}_\beta(\bigwedge_{i\in I} \varphi_i)$

Note that when applying inclusion-exclusion further cancellations may occur. For instance, if two queries $\bigvee_{i\in I'} \varphi_i$ and $\bigvee_{i\in I''} \varphi_i$ that appear with different signs in the sum are logically equivalent, then their probabilities cancel and do not need to be computed. If logical implication of formulas can be determined (which is in particular the case for UCQs), then applying the Möbius-inversion formula instead of inclusion-exclusion helps to avoid unnecessary computations. The corresponding lifted inference rule has the following form (we avoid to define the coefficients $\mu_I \in \mathbb{Z}$ in this paper, see [10] for details).

(M) $\mathbb{P}_\beta(\bigwedge_{i\in[k]} \varphi_i) = \sum_{I,\mu_I\neq 0} \mu_I \cdot \mathbb{P}_\beta(\bigvee_{i\in I} \varphi_i)$

Informally, a lifted query evaluation algorithm is a procedure that evaluates a query by recursively applying lifted inference rules. We are now making this intuition more formal. Note that the rules discussed above are data independent, that is, they only depend on the structure of the first-order formula and are independent of the concrete database D and valuation β. This enables us to first define a query plan whose size depends only on the formula. Later we show how to use this plan to evaluate Boolean queries under updates.

Definition 3.2. A *lifted inference plan* for a first-order formula φ is a directed rooted tree $\mathcal{T}$ where every node v is labeled with a formula ψ_v and an inference rule $\mathsf{r}_v \in \{$(ind-∃), (ind-∀), (ind-∧), (ind-∨), (≡), (atom), (¬atom), (incl-excl1), (incl-excl2), (M) $\}$ such that the following holds:

(1) If v is the root node, then $\psi_v = \varphi$.
(2) If v is a leaf, then ψ_v is a (negated) atom and r_v = (atom) or r_v = (¬atom), respectively.
(3) If r_v = (≡), then v has one child w such that $\psi_v \equiv \psi_w$ and
   - ψ_w is obtained from ψ_v by renaming a bound variable or
   - every atom in ψ_w is contained in ψ_v.
(4) If r_v = (ind-∧), then v has two children w_1 and w_2 such that $\psi_v = \psi_{w_1} \wedge \psi_{w_2}$ and ψ_{w_1}, ψ_{w_2} are independent.
(5) If r_v = (ind-∨), then v has two children w_1 and w_2 such that $\psi_v = \psi_{w_1} \vee \psi_{w_2}$ and ψ_{w_1}, ψ_{w_2} are independent.
(6) If r_v = (ind-∀), then v has one child w such that $\psi_v = \forall x\, \psi_w$ and x is a separator variable in ψ_w.
(7) If r_v = (ind-∃), then v has one child w such that $\psi_v = \exists x\, \psi_w$ and x is a separator variable in ψ_w.
(8) If r_v = (incl-excl1), then $\psi_v = \bigwedge_{i\in[k]} \varphi_i$ and v has for every nonempty $I \subseteq [k]$ one child w_I with $\psi_{w_I} = \bigvee_{i\in I} \varphi_i$.

(9) If r_v = (incl-excl2), then $\psi_v = \bigvee_{i\in[k]} \varphi_i$ and v has for every nonempty $I \subseteq [k]$ one child w_I with $\psi_{w_I} = \bigwedge_{i\in I} \varphi_i$.
(10) If r_v = (M), then $\psi_v = \bigwedge_{i\in[k]} \varphi_i$ and v has for every $I \subseteq [k]$ such that $\mu_I \neq 0$ one child w_I with $\psi_{w_I} = \bigvee_{i\in I} \varphi_i$.

We let $\Psi_{\mathcal{T}} = \{\psi_v \colon v \in V(\mathcal{T})\}$ and $\mathsf{R}_{\mathcal{T}} = \{\mathsf{r}_v \colon v \in V(\mathcal{T})\}$ be the set of formulas and rules used by the plan $\mathcal{T}$. A formula φ is *liftable (using rules* R*)* if it has a lifted inference plan $\mathcal{T}$ (with $\mathsf{R}_{\mathcal{T}} \subseteq \mathsf{R}$).

If we have a lifted inference plan $\mathcal{T}$ for a Boolean query φ, then we can easily compute the query result $\mathbb{P}_D(\varphi)$ on a probabilistic database D in polynomial time. For this, we just need to calculate bottom-up for every node v and every valuation $\beta \colon \mathrm{free}(\psi_v) \to \mathbf{dom}$ the probabilities $\mathbb{P}_{D,\beta}(\psi_v)$. The chosen lifted inference rules r_v tell us how to compute these numbers in each node from its children. Since we have to compute $n^{|\mathrm{free}(\psi_v)|}$ probabilities in each node and $|\mathrm{free}(\psi_v)| \leq |\mathrm{vars}(\varphi)|$, the overall running time is $O(f(\varphi)n^{|\mathrm{vars}(\varphi)|})$.

In their fundamental work, Dalvi and Suciu [10] showed that, modulo some preprocessing, we can compute a lifted inference plan for all Boolean UCQs for which a polynomial time query evaluation algorithm exists (assuming P ≠ #P). Before we can state this theorem, we need to introduce the technical notion of *ranked* formulas.

Definition 3.3. A first-order formula φ is *ranked* if there exists a partial order $\preceq$ on its variables such that for every atom $Rx_1 \cdots x_r$ in φ and $1 \leq i < j \leq r$ it holds that $x_i \prec x_j$.

Theorem 3.4 (Dichotomy Theorem for Ranked Boolean UCQs [10]). *Let φ be a ranked Boolean UCQ. Either the query evaluation problem for φ is #P-hard, or there is an algorithm that computes a lifted inference plan for φ that uses the rules* (atom), (≡), (ind-∨), (ind-∧), (ind-∃), *and* (M).

Remark 3.5. The restriction to ranked formulas is not essential. Dalvi and Suciu [10] showed that for every schema σ and σ-formula φ there is a new schema σ' and a *ranked* σ'-formula φ' that can be used instead of φ. In particular, for any domain **dom** there is a computable bijection F between all σ-ground tuples and all σ'-ground tuples over the domain **dom** such that $\mathbb{P}_D(\varphi) = \mathbb{P}_{F(D)}(\varphi')$ for all probabilistic databases D of schema σ [10, Proposition 4.2]. Note that using the bijection F also preserves updates: $\mathbb{P}_D(\varphi)$ can be maintained with $O(g(n))$ update time if, and only if, $\mathbb{P}_{F(D)}(\varphi')$ can be maintained with $O(g(n))$ update time.

3.2 Dynamic Lifted Query Evaluation

In this section we show that if a Boolean query φ has a lifted inference plan $\mathcal{T}$, then it can be maintained with constant update time (Theorem 3.8). As a consequence we obtain that every Boolean UCQ is either #P-hard or can be maintained with constant update time (Corollary 3.9). The basic idea is to maintain for every formula $\psi \in \Psi_{\mathcal{T}}$ and assignment $\beta : \mathrm{free}(\psi) \to \mathbf{dom}$ its probability value $\mathbb{P}_\beta(\psi)$. To ensure constant update time, we need to verify that only few stored values are actually affected from the insertion or deletion of a ground tuple and that each such value can be recomputed in constant time. To this end, we provide the following syntactic definition, which is stated for Boolean and non-Boolean formulas.

Definition 3.6. Let φ be a formula, $C \subseteq \mathrm{free}(\varphi)$ a set of free variables in φ, and $\beta \colon C \to \mathbf{dom}$.[2] We say that (φ, β) is *affected* by a ground atom $Ra_1 \cdots a_r$, if there is an extension $\beta' \colon \mathrm{vars}(\varphi) \to \mathbf{dom}$ of β and an atom $Rx_1 \cdots x_r$ in φ such that $\beta'(x_i) = a_i$ for all $i \in [r]$. Moreover, (φ, β) is *supported* by a probabilistic database D, if it is affected by at least one ground tuple in D. A Boolean formula φ is affected/supported, if (φ, ϵ) is affected/supported.

The important property of this definition is that if (φ, β) is *not* affected by a ground atom $Ra_1 \cdots a_r$, then it is independent of $Ra_1 \cdots a_r$. In particular, inserting, deleting or changing the probability of $Ra_1 \cdots a_r$ in an arbitrary tuple-independent probabilistic database D has no effect on the query result $\mathbb{P}_{D,\beta}(\varphi)$. The other direction does not necessarily hold: Sa affects $\forall x\,(Sx \vee \neg Sx)$, but this tautological query does not depend on Sa. However, for our purposes it suffices that *affecting* a query is a necessary requirement for having an effect to the query.

Lemma 3.7. *Let φ be a first-order formula, $\mathcal{T}$ a lifted inference plan for φ, and $\gamma \colon$ free$(\varphi) \to$ **dom** a valuation. For every ground tuple $Ra_1 \cdots a_r$ and every $\psi \in \Psi_{\mathcal{T}}$ there is at most one $\beta \colon$ free$(\psi) \to$ **dom** with $\beta{\restriction}_{free(\varphi)} \subseteq \gamma$ such that $Ra_1 \cdots a_r$ affects (ψ, β). If such a β exists, then it can be computed in time $O(|\psi|)$.*

Proof. We first argue that for all $\psi_v \in \Psi_{\mathcal{T}}$ the variables in $\mathrm{free}(\psi_v) \setminus \mathrm{free}(\varphi)$ are separator variables. We proceed by induction over the height of v in $\mathcal{T}$. The statement is trivially true for the root, as $\psi_v = \varphi$. For the induction step note that, except for (ind-∃) and (ind-∀), no new free variables are introduced and if a variable is a separator variable in the parent node v, then it is also separator variable in all child nodes v_i. Particular attention is needed for the concrete choice of equivalent transformations in the equivalence rule (≡), as there are equivalent transformations that do not preserve separator variables. However, the syntactic side conditions ensure that, besides renaming bound variables, no new atoms are introduced and hence the separator variable condition is maintained. For (ind-∃) and (ind-∀), the new free variable x is a separator variable by definition.

Now suppose that $Ra_1 \cdots a_r$ affects (ψ, β) for some $\psi \in \Psi_{\mathcal{T}}$, which means that there is an atom $Rx_1 \cdots x_r$ in ψ and an extension $\beta' \supseteq \beta$ such that $\beta'(x_i) = a_i$. Since every free variable in $\mathrm{free}(\psi) \setminus \mathrm{free}(\varphi)$ is a separator variable, we have $\mathrm{free}(\psi) \setminus \mathrm{free}(\varphi) \subseteq \{x_1, \ldots, x_r\}$ and hence $a_1, \ldots, a_r$ fixes $\beta{\restriction}_{\mathrm{free}(\psi)\setminus\mathrm{free}(\varphi)}$. Since $\beta{\restriction}_{\mathrm{free}(\psi)\cap\mathrm{free}(\varphi)} = \gamma{\restriction}_{\mathrm{free}(\psi)}$ it follows that β is uniquely determined. □

Theorem 3.8. *Let φ be a liftable first-order formula and $\mathcal{T}$ a lifted inference plan for φ. For every $\gamma \colon$ free$(\varphi) \to$ **dom** there is an algorithm that computes $\mathbb{P}_{D,\gamma}(\varphi)$ in time $O(\|D\|)$ and maintains this value in constant time on every single-tuple update to D.*

Proof. We fix $\gamma \colon \mathrm{free}(\varphi) \to \mathbf{dom}$ and assume that all valuations β mentioned in this proof satisfy $\beta{\restriction}_{\mathrm{free}(\varphi)} \subseteq \gamma$. For every $\psi \in \Psi_{\mathcal{T}}$ we store the probability values $\mathbb{P}_{D,\beta}(\psi)$ for all $\beta \colon \mathrm{free}(\psi) \to \mathbf{dom}$ such that (ψ, β) is supported by the current database D. In addition, for every $\psi \in \Psi_{\mathcal{T}}$ we store the probability $p_{\mathfrak{u}}(\psi) = \mathbb{P}_{D_{\mathfrak{u}},\beta}(\psi)$ when evaluating ψ with an arbitrary assignment β over the empty

[2] Note that in this section, C will always be equal to $\mathrm{free}(\varphi)$. We will consider $C \subsetneq \mathrm{free}(\varphi)$ in Section 4.

database D_{u}. Note that $p_{\mathrm{u}}(\psi) = \mathbb{P}_{D,\beta}(\psi)$ for all β such that (ψ, β) is not supported by D.

When an update inserts or deletes a ground tuple in the current database, we apply Lemma 3.7 to identify which $\psi_v \in \Psi_{\mathcal{T}}$ are affected and to compute the corresponding unique assignments β_v. Afterwards, we (re)compute all values $\mathbb{P}_{\beta_v}(\psi_v)$ using a bottom-up traversal of the inference tree $\mathcal{T}$. For all rules except (ind-$\exists$) and (ind-$\forall$) we can directly do the recomputation using the corresponding lifted inference rule r_v, as the value $\mathbb{P}_{\beta_v}(\psi_v)$ depends only on the numbers $\mathbb{P}_{\beta_v}(\psi_{v_i})$ for a constant number of child nodes v_i.

If $\mathsf{r}_v = $ (ind-$\forall$) or $\mathsf{r}_v = $ (ind-$\exists$) and hence $\psi_v = \forall x\, \psi_w$ or $\psi_v = \exists x\, \psi_w$, we maintain the following sets of domain elements for all $\beta\colon \mathrm{free}(\psi_v) \to \mathbf{dom}$ where $(\psi_w, \beta_{x\mapsto a})$ is supported by D for some $a \in \mathbf{dom}$:

$$S(\beta) := \{a \in \mathbf{dom} :\ (\psi_w, \beta_{x\mapsto a}) \text{ is supported by } D\},$$
$$T_0(\beta) := \{a \in S(\beta) :\ \mathbb{P}_{\beta_{x\mapsto a}}(\psi_w) = 0\},$$
$$T_1(\beta) := \{a \in S(\beta) :\ \mathbb{P}_{\beta_{x\mapsto a}}(\psi_w) = 1\}.$$

By Lemma 3.7 this can be done efficiently as every single-tuple update causes at most one element to be inserted to or deleted from these sets. We also maintain the corresponding auxiliary variable

$$p_\forall(\beta) := \textstyle\prod_{a \in S(\beta)\setminus(T_0(\beta)\cup T_1(\beta))} \mathbb{P}_{\beta_{x\mapsto a}}(\psi_w) \quad \text{or}$$
$$p_\exists(\beta) := \textstyle\prod_{a \in S(\beta)\setminus(T_0(\beta)\cup T_1(\beta))} (1 - \mathbb{P}_{\beta_{x\mapsto a}}(\psi_w)).$$

These variables can also be maintained in constant update time by multiplying with or dividing by the corresponding probability when an update causes a to be inserted to or deleted from $S(\beta) \setminus (T_0(\beta) \cup T_1(\beta))$ for some β. With this auxiliary information we are ready to compute the new probabilities of the query:

$$\mathbb{P}_{\beta_v}(\forall x\, \psi_w) = \begin{cases} 0, \text{ if } T_0(\beta_v) \neq \emptyset \\ p_\forall(\beta_v) \cdot p_{\mathrm{u}}(\psi_w)^{|\mathbf{dom}| - |S(\beta_v)|}, \text{ else.} \end{cases}$$

$$\mathbb{P}_{\beta_v}(\exists x\, \psi_w) = \begin{cases} 1, \text{ if } T_1(\beta_v) \neq \emptyset \\ 1 - p_\exists(\beta_v) \cdot p_{\mathrm{u}}(\psi_w)^{|\mathbf{dom}| - |S(\beta_v)|}, \text{ else.} \end{cases}$$

This finishes the proof that the data structure can be maintained with constant update time, after which we can read off the query result $\mathbb{P}_{D,\gamma}(\varphi) = \mathbb{P}_{D,\beta}(\psi_v)$ in the root node v. Note that the data structure can be initialized in constant time over the empty database, which implies linear time initialisation for a database D after inserting all tuples. □

Corollary 3.9. *Every Boolean UCQ is either #P-hard or can be maintained with constant update time.*

Proof. Follows immediately from Theorem 3.4, Remark 3.5 and Theorem 3.8 □

4 DYNAMIC RANKED ENUMERATION

In the last section, we studied the dynamic evaluation of Boolean queries, where the query result is a single probability value. The goal of this section is to provide dynamic algorithms for evaluating non-Boolean queries, where the query result is a relation in which every tuple is annotated with a probability. In particular, we want to enumerate the output tuples ranked by their probability and with bounded delay.

Recall from Section 2 that a *(first-order) query* is a pair (φ, W), where φ is a first-order formula and $W \subseteq \mathrm{free}(\varphi)$ the set of *output variables*. Moreover, the *result set* $[\![(\varphi, W)]\!]^{D,\beta}$ of a query (φ, W) under a probabilistic database D and a valuation $\beta\colon C \to \mathbf{dom}$, where $C = C(\varphi, W) := \mathrm{free}(\varphi) \setminus W$, is the set of all pairs (τ, p) of result tuples $\tau\colon W \to \mathbf{dom}$ that have a positive probability $p = \mathbb{P}_{D,\tau\uplus\beta}(\varphi) > 0$.

We now provide a formal description of our algorithmic problem: A *dynamic ranked enumeration algorithm* for a query (φ, W) and an assignment $\beta\colon C \to \mathbf{dom}$ computes the query result $[\![(\varphi, W)]\!]^{D,\beta}$ on a probabilistic database D and implements two modes: *Update mode*, in which it receives updates to the database, and *enumeration mode*, in which it is asked to list the elements of $[\![(\varphi, W)]\!]^{D,\beta}$, ranked by their probabilities and without repetitions, where D is the current database after all previous updates. The algorithm starts in update mode on the empty database and must handle the following calls:

(1) `insert`$(Ra_1 \ldots a_r, p)$, upon which the ground tuple $Ra_1 \ldots a_r$ is inserted into the database with probability p.
(2) `delete`$(Ra_1 \ldots a_r)$, upon which the ground tuple $Ra_1 \ldots a_r$ is removed from the database.
(3) `enumerate`(), upon which the algorithm must enter enumeration mode and return the first element of the output set.

In enumeration mode, the algorithm must handle repeated calls to the following method:

(4) `next`(), upon which it is to return the next element of the output, if there is any; or, if there is no element left, it must return the special end-of-enumeration symbol `EOE` and switch into update mode.
(5) `abort`(), upon which it must abort the enumeration and switch back into update mode.

If the distinction is not important, we will write "`update`(t, p)" to mean either one of `insert`(t, p) or `delete`(t). We call "update time" the time the algorithm needs for a call to `update`(t, p). We call "delay" the maximum time needed to handle a call to `enumerate`() and `next`().

Our goal is to design dynamic ranked enumeration algorithms that achieve $O(\log \|D\|)$ update time and $O(\log \|D\|)$ delay for a large class of queries. Below we define the class of *r-liftable* queries for which this is possible. This class extends the class of liftable Boolean queries from Definition 3.2, which can be evaluated with constant update time.

Definition 4.1. The class of *r-liftable* queries is recursively defined as follows:

(1) If φ is liftable first-order formula (according to Definition 3.2), then $(\varphi, \emptyset)$ is r-liftable .
(2) If (φ, W) is r-liftable and x is a separator variable in φ, then $(\varphi, W \cup \{x\})$ is r-liftable.
(3) If (φ_1, W_1) and (φ_2, W_2) are r-liftable, φ_1 and φ_2 are independent, and $W_1 \cap W_2 = \emptyset$, then $(\varphi_1 \wedge \varphi_2, W_1 \cup W_2)$ is r-liftable.

(4) If (φ_1, W_1) and (φ_2, W_2) are r-liftable, φ_1 and φ_2 are independent, and $W_1 \cap W_2 = \emptyset$, then $(\varphi_1 \vee \varphi_2, W_1 \cup W_2)$ is r-liftable.

Note that the rules (3) and (4) are similar to the lifted inference rule (ind-$\wedge$) and (ind-$\vee$), respectively. Moreover, the introduction of output variables (2) relies on the same syntactic property as the quantifier rules (ind-$\forall$) and (ind-$\exists$). In Section 4.3 we discuss the difficulties that arise when trying to achieve a non-Boolean variant of the inclusion-exclusion rules (incl-excl1), (incl-excl2), and (M).

4.1 A dynamic ranked enumeration algorithm for liftable queries

In this subsection, we present a dynamic ranked enumeration algorithm for the class of r-liftable queries and prove the following main theorem.

Theorem 4.2. *For any r-liftable query (φ, W), database D, and valuation β: free$(\varphi) \setminus W \to$ **dom**, there is a data structure that supports dynamic ranked enumeration of the query result $[\![(\varphi, W)]\!]^{D,\beta}$ with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay.*

We prove these rules by describing, for each rule, a recursive data structure $\mathbf{E}$ that can enumerate the probability-ranked elements of $[\![(\varphi, W)]\!]^{D,\beta}$. We will call these data structures *enumerators*. In the following, we first give a high-level introduction, then introduce the enumerator data structures for each rule, and finally bring everything together to prove the theorem. Note that we will omit the `abort()` function at first and consider it in Subsection 4.1.5. We begin with some notation and definitions.

Recall from Section 2 that elements of a query result $[\![(\varphi, W)]\!]^{D,\beta}$ are of the form (τ, p), where $\tau\colon W \to$ **dom** and $p \in [0,1] \subset \mathbb{Q}$. When we want to explicitly state the order of different elements of $[\![(\varphi, W)]\!]^{D,\beta}$ under the probability ranking, we use the upper index and write $(\tau, p)^1, \ldots, (\tau, p)^m$. We interchangeably write (τ^i, p^i) or $(\tau, p)^i$.

Definition 4.3. We extend Definition 3.6 by the following: We say that a result $[\![(\varphi, W)]\!]^{D,\beta}$ is *affected* by a ground tuple t (or by a call `update`(t, p)) iff the pair (φ, β) is affected by t. A result $[\![(\varphi, W)]\!]^{D,\beta}$ is *supported* iff the pair (φ, β) is supported over D; the result is *unsupported* otherwise. An enumerator `Enum`(φ, β) is affected (supported, unsupported) if the corresponding query result $[\![(\varphi, W)]\!]^{D,\beta}$ is affected (supported, unsupported).

It is easy to see that if we let D_u be the empty database, then all elements in the result $[\![(\varphi, W)]\!]^{D_\mathrm{u},\beta}$ (for any φ, W, β) have the same probability p_shared; and this probability is independent of W and β.

Definition 4.4. We define $p_\mathrm{u}(\varphi)$:

$$p_\mathrm{u}(\varphi) := \begin{cases} 0 & \text{if } [\![(\varphi, W)]\!]^{D_\mathrm{u},\beta} = \emptyset, \\ p_\mathrm{shared} & \text{otherwise.} \end{cases}$$

4.1.1 High-Level Description. For the dynamic ranked enumeration of a result set $[\![(\varphi, W)]\!]^{D,\beta}$, we will describe an *enumerator* data structure. More precisely, for every rule in Definition 4.1, we will define one *type* of enumerator: For example, an `AndEnumerator` will be able to "break down" the task of enumerating a query result $[\![(\varphi_1 \wedge \varphi_2, W_1 \cup W_2)]\!]^{D,\beta}$ into the separate tasks of enumerating $[\![(\varphi_1, W_1)]\!]^{D,\beta\restriction_{C_1}}$ and $[\![(\varphi_1, W_2)]\!]^{D,\beta\restriction_{C_2}}$. This leads to an overall structure of a tree of enumerators, see Figure 1: For example, enumerator $\mathbf{E}_0^x$ relies on enumerators $\mathbf{E}_1^\wedge, \mathbf{E}_2^\wedge, \ldots$ for their partial solutions.

For the remainder of this subsection, we denote by D the current database instance. Note also that the set $W = \text{free}(\varphi) \setminus \text{dom}(\beta)$ is implicitly given by β. We will therefore mostly omit D and W from $[\![(\varphi, W)]\!]^{D,\beta}$ and simplify the notation to $[\![\varphi]\!]^\beta$. With the same reasoning, we will denote the enumerator for a result set $[\![\varphi]\!]^\beta = [\![(\varphi, W)]\!]^{D,\beta}$ by `Enum`(φ, β).

Invariants. The enumerator data structures will implement the two modes of a dynamic ranked enumeration algorithm: the update mode and the enumeration mode. We present the *update invariant* and the *enumeration invariant*, which are maintained by the enumerator data structure in the respective modes:

Update Invariant: The enumerator has pre-computed its first element under the ranking and can produce it in $O(1)$ time.

Enumeration Invariant: The enumerator has pre-computed the element it needs to return on the next `next()` call, and can produce it in $O(1)$ time.

Because these invariants will hold, our enumerators offer the functions `first()` (in update mode) and `preview_next()` (in enumeration mode) in addition to the required functions for dynamic ranked enumeration algorithms.

Consider an enumerator at the top of the enumerator tree and its descendant enumerators. Calls of `update`(t, p), `enumerate()`, or `next()` force the enumerator to re-establish the invariants. This re-establishing process is performed recursively in all descendant enumerators that are *affected* by the call. We will show that the number of descendant enumerators that are affected by each call is independent of the size of the database, and that the re-establishing

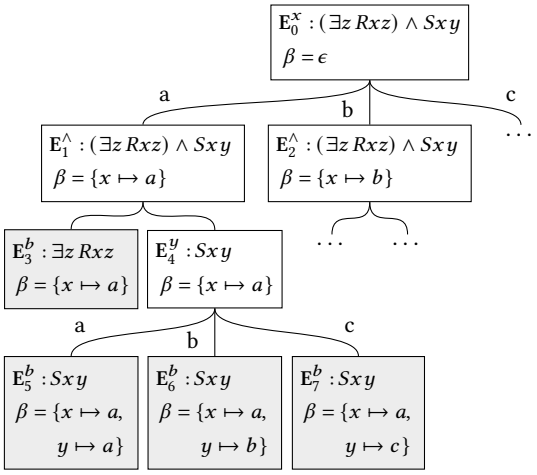

Figure 1: An enumerator tree. Every box represents an enumerator. The upper exponent of E indicates the type of the enumerator, e.g. E^b for a Rule (1) enumerator. Enumerators of type E^b (highlighted in light grey) end the recursion.

procedure takes at most logarithmic time in each affected enumerator. (Mostly, re-establishing the invariants consist of deleting from and inserting into a priority queue; this requires logarithmic time.)

Result sets $[\![\varphi]\!]^{\beta}$ consist of tuples (τ, p). To designate the end of an enumeration, our enumerators return the special result tuple $(\texttt{EOE}, 0)$, where EOE is the "end of enumeration" symbol. The probability 0 will ensure that when we rank possible output tuples by their probability, the special $(\texttt{EOE}, 0)$ will always be ranked last.

We define the $\texttt{update}(t, p)$ call to return a pair (s, o), where s is one of the two symbols SUP or UNSUP (which designate that the updated enumerator is supported or unsupported, respectively) and o is the first output tuple of the updated enumerator (i.e. either some (τ, p) or $(\texttt{EOE}, 0)$).

In the enumeration tree, an enumerator is only initialized when it becomes supported. This "pruning" or "omission" of unsupported enumerators is a key part of why we are able to provide logarithmic updates. If an $\texttt{insert}(t, p)$ call lets previously unsupported enumerators become supported, these enumerators are first initialized and then passed the insert call. If a $\texttt{delete}(t)$ call leads to an enumerator becoming unsupported, the enumerator returns the special UNSUP message to its parent; this instructs the parent to delete the enumerator. In both of these cases, only a constant number of enumerators are affected.

We can enumerate unsupported $[\![\varphi]\!]^{\beta}$ by computing, in constant time, the probability $p_{\mathfrak{u}}(\varphi)$, and then outputting either $(\texttt{EOE}, 0)$ (if $p_{\mathfrak{u}}(\varphi) = 0$), or $(\tau, p_{\mathfrak{u}}(\varphi))$ for every $\tau \colon W \to \mathbf{dom}$ otherwise. The latter can be reduced to enumerating the cartesian product $\mathbf{dom}^{|W|}$.

4.1.2 Enumerator for Rule (2). Before we describe the enumerator data structure, we show how to maintain a set of unsupported constants in a dynamic data structure. We assume that there is a total order on **dom**, and that we can, given a constant a, access the preceding and succeeding constants under the order in constant time, and that we can test $a \leq a'$ for two constants in constant time.

Lemma 4.5. *Let $U \subseteq \mathbf{dom}$ be a subset of* **dom** *that is initially set to $U = \mathbf{dom}$, and let $n_U := |\mathbf{dom} \setminus U|$. There is a data structure E_U that can enumerate the elements of U with $O(\log n_U)$ delay and provides the following $O(\log n_U)$ functions to update U:*

(1) $\texttt{skip}(a)$*, which removes the constant $a \in \mathbf{dom}$ from U, and*
(2) $\texttt{unskip}(a)$*, which adds the constant $a \in \mathbf{dom}$ back into U.*

Proof. The set U is maintained as follows: When a constant a is skipped, we initialize a triple (a_p, T_a, a_s), where $a_p := \texttt{pred}(a)$ and $a_s := \texttt{succ}(a)$ are the predecessor and successor wrt. the total order on **dom**, and T_a is an AVL tree that contains only a. Under repeated skip and unskip calls to E_U, the AVL trees designate continuous areas of skipped constants. We can join adjacent areas in $O(\log n_U)$, and we can split an area in $O(\log n_U)$ if a constant it contains is unskipped.

In enumeration mode, whenever E_U needs to output the successor of an element b, E_U checks whether there is a triple that contains b in the first position. If it finds such a triple (b, T, c), it outputs as successor the constant c that is in the third position of the triple; if there is no such triple, it simply outputs $\texttt{succ}(b)$. □

The next lemma formalizes the inductive step for Rule (2) in Definition 4.1.

Lemma 4.6. *Let (φ, W_1) be a query where x is a separator variable in φ and $x \notin W_1$. Assume that for any $\beta_1 : C_1 \to \mathbf{dom}$, there exists a dynamic ranked enumeration algorithm E_1 that enumerates $[\![(\varphi, W_1)]\!]^{D,\beta_1}$ with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay.*

Then, there exists a dynamic ranked enumeration algorithm **E** *of type* $\texttt{WhichEnumerator}(\varphi, \beta)$ *that enumerates $[\![(\varphi, W)]\!]^{D,\beta}$ with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay, where $W = W_1 \uplus \{x\}$ and $\beta = \beta_1 \restriction_C$.*

Proof of Lemma 4.6. Let us fix the query (φ, W) and a valuation $\beta : C \to \mathbf{dom}$, and assume that there exist enumerators E_a for every $[\![(\varphi, W_1)]\!]^{D,\beta_{x \mapsto a}}$. To prove the lemma, we need to show how to construct the enumerator **E**.

We first describe the **update mode** (see Algorithm 1). The enumerator **E** of type $\texttt{WhichEnumerator}(\varphi, \beta)$ is initialized with a priority queue Q. This priority queue will contain triples $(a, E_a, (\tau, p)^1_a)$ for $a \in \mathbf{dom}$, where E_a is the enumerator that can enumerate $[\![(\varphi, W_1)]\!]^{\beta_{x \mapsto a}}$, and $(\tau, p)^1_a$ is E_a's first output tuple. The queue Q will be sorted in descending order by the values p^1_a. This means that the first triple in the queue, let it be $(b, E_b, (\tau, p)^1_b)$ for some $b \in \mathbf{dom}$, will contain the first output tuple $(\tau, p)^1_b$ of **E**.

The queue Q is initialized with one special triple $(\mathfrak{u}, E_{\mathfrak{u}}, p_{\mathfrak{u}})$, where $\mathfrak{u} \notin \mathbf{dom}$ is the special "unsupported" symbol and $p_{\mathfrak{u}} := p_{\mathfrak{u}}(\varphi)$.

```
1  Class WhichEnumerator:
2      function WhichEnumerator(φ, β):
3          E_u ← init_unsup_for(φ);
4          Q ← a new, empty priority queue;
5          Q.enqueue((u, E_u, E_u.first()));
6          E ← new Object(φ, β, Q, E_u);
7          return E;
8      end
9      function E.update(t = Ra_1...a_r, p):
10         a ← the unique a ∈ dom, s.t. E_a is affected by t;
11         if (a, ...) ∉ E.Q then
12             E_a ← new Enumerator(φ, β_{x↦a});
13             E.E_u.skip(a);
14         else
15             (a, E_a, (τ'_a, p'_a)) ← E.Q.remove((a, ...));
16         end
17         switch E_a.update(t, p) do
18             case SUP, (τ'_a, p'_a) do
19                 E.Q.enqueue((a, E_a, (τ'_a, p'_a)));
20             case UNSUP, ... do
21                 E.E_u.unskip(a);
22         end
23         s ← (SUP if |E.Q| > 1; UNSUP otherwise);
24         (τ, p)^1 ← take the first tuple from E.Q;
25         return s, (τ, p)^1;
26     end
27 end
```

Algorithm 1: Update mode of $\texttt{WhichEnumerator}(\varphi, \beta)$

This special triple represents all unsupported child enumerators – because all unsupported result tuples have the same probability, we can treat them as one. When an `insert`(t, p) call makes a child enumerator E_a supported, the child enumerator is initialized, its triple $(a, E_a, \ldots)$ is added to the priority queue Q, and the constant a is "skipped" in $E_{\mathfrak{u}}$. When a `delete`(t) call removes the support of a child enumerator E_a, this is reversed. Thus, the queue Q always contains triples for the supported enumerators, and the unsupported enumerators are handled by $E_{\mathfrak{u}}$.

When the enumerator **E** receives a call `update`(t, p), it computes the unique constant a, such that the child enumerator E_a is affected by t (a is unique because x is a separator variable). The enumerator propagates the `update`(t, p) call to E_a, upon which the child enumerator returns its new first output tuple $(\tau, p)'_a$. The triple $(a, E_a, \ldots)$ in Q is updated with the new first output tuple of E_a, and **E** re-establishes the ranking of the queue and returns the first element in Q as its own first output tuple.

It is clear that this algorithm maintains the update invariant, as the first output tuple is always contained in the first triple in Q.

We now describe the **enumeration mode** (see Algorithm 2). The basic idea is that we use the priority queue Q to rank the next output tuple of each child enumerator E_a. When the enumerator **E** receives a call `next`, we pop the first triple $(a, E_a, (\tau, p)^k_a)$ from the priority queue Q and return $(\tau, p)^k_a$ as our next output tuple to our parent. We then call `next` on the child enumerator, and add $(a, E_a, (\tau, p)^{k+1}_a)$ back in the priority queue Q.

If we would do exactly this, we would "use up" the priority queue Q, and we would have to rebuild it at the end of the enumeration procedure. Therefore, we instead do the following: At the start of the enumeration procedure, we create a second, temporary priority queue Q_{enum}. We do not pop enumerators from Q, but simply iterate with a pointer j_Q through it. If the pointer j_Q points to a triple $(a, E_a, (\tau, p)^1_a) \in Q$ that has higher probability than the first element in Q_{enum}, we use that triple as described above (but without removing it from Q), move the pointer j_Q forward by one, and add the new triple $(a, E_a, (\tau, p)^2_a)$ to Q_{enum}.

If the special triple $(\mathfrak{u}, E_{\mathfrak{u}}, p_{\mathfrak{u}})$ is at the top of the queue(s), it outputs all unsupported output tuples (as an example, consider the query $\varphi(x) = \neg Rx$ on the empty database). As discussed above, the enumerator $E_{\mathfrak{u}}$ must log-delay enumerate all possible valuations of W_1 (or: the cartesian product $\mathbf{dom}^{|W|} \times U$, where U is the set of all *unsupported* constants $a \in \mathbf{dom}$, that is, those that are *not* represented in the priority queue Q). The set of unsupported constants U is maintained and enumerated as described in Lemma 4.5. Note that if U is the set of unsupported constants, then the number n_U in the lemma is at most $|\mathbf{adom}|$.

Note that we left the triples $(a, E_a, (\tau, p)^1_a)$ in Q unchanged, so that returning to update mode after an exhaustive enumeration is as easy as deleting Q_{enum} and j_Q. We did modify all of our child enumerators E_a by repeatedly calling `next()` on them, but since they all turned themselves back into update mode before sending `EOE` to us, we do not need to clean up after them.

We now argue that calls `update`(t, p), and `next()` are handled by the enumerator **E** in logarithmic time. The enqueue and pop operations of the priority queue are logarithmic in the length of the queue, and this length is bounded by the size of the active domain $n_a := |\mathbf{adom}|$. Skipping and unskipping constants in $E_{\mathfrak{u}}$ is also in logarithmic time by Lemma 4.5. The child enumerator's `update`(t, p), and `next()` need logarithmic time by assumption. This gives us the lemma. □

```
1  Class WhichEnumerator:
2      function E.enumerate():
3          E.Q_enum ← shallow copy of E.Q;
4          E.j_Q ← 1;
5          return E.next();
6      end
7      function E.next():
8          switch E.Q, E.Q_enum do
9              case E.Q_enum[1].p = E.Q[E.j_Q].p = 0 do
10                 restore default state;
11                 return (EOE, 0);
12             case E.Q_enum[1].p ≥ E.Q[E.j_Q].p do
13                 (a, E_a, (τ^k, p^k)) ← E.Q_enum.pop();
14             case E.Q_enum[1].p < E.Q[E.j_Q].p do
15                 (a, E_a, (τ^k, p^k)) ← E.Q[E.j_Q];
16                 E.j_Q ← E.j_Q + 1;
17         end
18         (τ^k, p^k) ← E_a.next();
19         (τ^{k+1}, p^{k+1}) ← E_a.preview_next();
20         E.Q_enum.enqueue((a, E_a, (τ^{k+1}, p^{k+1})));
21         return (τ^k, p^k);
22     end
23 end
```

Algorithm 2: Enumeration mode of `WhichEnumerator`(φ, β). We use the notation $Q[i].p$ to denote the probability of the i'th triple in the queue Q, which we define to be 0 if the index i is out of bounds or if the queue is empty. The (τ^k, p^k) in line 13 (or 15), and line 18 are equal – we call `next()` in line 18 not for its return value, but for its side effects on E_a.

4.1.3 *Enumerator for Rules (3) and (4).*

We only present the enumerator data structure for Rule (3) in detail, as the enumerator for Rule (4) is similar (see Lemma 4.8).

Lemma 4.7. *Let (φ_1, W_1) and (φ_2, W_2) be two queries where φ_1 and φ_2 are independent and $W_1 \cap W_2 = \emptyset$. Assume that for any $\beta_1 : C_1 \to \mathbf{dom}$, $\beta_2 : C_2 \to \mathbf{dom}$, there exist dynamic ranked enumeration algorithms E_1, E_2 that enumerate $[\![(\varphi_1, W_1)]\!]^{D,\beta_1}$ and $[\![(\varphi_2, W_2)]\!]^{D,\beta_2}$, respectively, with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay.*

Then, there exists a dynamic ranked enumeration algorithm **E** *of type* `AndEnumerator`(φ, β) *that enumerates $[\![(\varphi, W)]\!]^{D,\beta}$ with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay, where $\varphi = \varphi_1 \wedge \varphi_2$, $W = W_1 \cup W_2$, and any $\beta : C \to \mathbf{dom}$.*

Proof of Lemma 4.7. Let us fix the query (φ, W) and a valuation $\beta : C \to \mathbf{dom}$, and assume that there exist enumerators E_1, E_2 with $\beta_1 = \beta\restriction_{C_1}$ and $\beta_2 = \beta\restriction_{C_2}$. To prove the lemma, we need to show how to construct the enumerator **E**.

We first describe the **update mode** (see Algorithm 3). The enumerator **E** of type `AndEnumerator`$(\varphi_1 \wedge \varphi_2, \beta)$ is initialized with

a priority queue Q and two cache arrays, $\mathcal{A}_1$ and $\mathcal{A}_2$. The enumerator **E** lazily initializes two child enumerators, E_1 and E_2, that enumerate the results $[\![(\varphi_1, W_1)]\!]^{D,\beta_1}$ and $[\![(\varphi_2, W_2)]\!]^{D,\beta_2}$, respectively, on the first `insert` call that affects them; and it deletes them when they send the UNSUP signal. While they are unsupported, they are substituted by special $E_{\mathrm{u},1}$ and $E_{\mathrm{u},2}$ enumerators, respectively.

If any of the two submits $(\mathtt{EOE}, 0)$ as its first output tuple, the result set of the enumerator is also empty. For two output tuples $(\tau_1, p_1) \in E_1$ and $(\tau_2, p_2) \in E_2$, we let $(\tau_1, p_1) \odot (\tau_2, p_2) := (\tau_1 \cup \tau_2, p_1 \cdot p_2)$. If the child enumerators submit $(\tau, p)_1^1$ and $(\tau, p)_2^1$, then the enumerator sets each pair as the first entry in the corresponding cache list $\mathcal{A}_i$, and it adds the pair $((1,1), (\tau, p)_1^1 \odot (\tau, p)_2^1)$ to the priority queue Q.

Next, we describe the **enumeration mode** (see Algorithm 4). For the following, we will denote by t_i^j the jth output tuple of E_i, so that $t_i^j = (\tau_i^j, p_i^j)$. The output set $[\![\varphi_1 \wedge \varphi_2]\!]^\beta$ consists of the cartesian product of the outputs of E_1 and E_2, that means all $T^{j_1,j_2} := t_1^{j_1} \odot t_2^{j_2}$, where $t_1^{j_1} \in [\![\varphi_1]\!]^{\beta\restriction_{C_1}}$ and $t_2^{j_2} \in [\![\varphi_2]\!]^{\beta\restriction_{C_2}}$.

Because both child enumerators are ranked, we can ranked-enumerate all T^{j_1,j_2} with a simple search: Let us consider the cartesian product of output tuples T^{j_1,j_2} as a matrix, where indices of enumerator E_1 index the rows, and indices of E_2 the columns. (See Figure 2 for an example.) Let J be a set of $|E_1|$ many pointers J_k, each of them pointing to a certain element T^{k,J_k} in row k. We initialize all of these pointers to $J_k \leftarrow 1$ for all k and add all T^{k,J_k} to the priority Q with their probability as priority. On a call to `next()`, we now pop the first T^{k,J_k} from the queue, output it as our next tuple, then set $J_k \leftarrow J_k + 1$ and insert the new T^{k,J_k} into the queue.

Because the elements of E_1 are ranked as well, we know that $T^{k,1} \geq T^{k+1,1}$ under our ranking. This means that we only need to initialize the pointer J_{k+1} when we output the tuple $T^{k,1}$; and that we only need to initialize J_1 at the beginning.

We now argue that calls `update`(t, p), and `next()` are handled by the enumerator **E** in logarithmic time. The child enumerator's `update`(t, p), and `next()` need logarithmic time by assumption. This gives us directly that the update function of **E** is also logarithmic. During enumeration, the queue Q of candidates will have

$[\![Sy]\!]^\epsilon \rightarrow$	$(b, 0.9)$	$(e, 0.4)$	$\cdots$	$(d, 0.2)$	$(\mathtt{EOE}, 0)$
$[\![Rx]\!]^\epsilon \downarrow$					
$(a, 1)$	$(ab, 0.9)$	$(ae, 0.2)$	$\cdots$	$(ad, 0.2)$	
$(c, 0.5)$	$(cb, 0.45)$	$(ce, 0.2)$	$\cdots$	$(cd, 0.1)$	
$(b, 0.1)$	$(bb, 0.09)$	$(be, 0.04)$	$\cdots$	$(bd, 0.02)$	
$(\mathtt{EOE}, 0)$					

Figure 2: An example of a cartesian product of two enumerators E_1, E_2 in a formula $\varphi(x, y) = Rx \wedge Sy$. The entry $(ab, 0.9)$ means that the formula $\varphi(a, b) = Ra \wedge Sb$ has probability 0.9. In this example, underlines mark the two tuples that have already been output, dashed underlines mark the position of the index J_k in every row k.

```
1  Class AndEnumerator:
2      function AndEnumerator(φ = φ1 ∧ φ2, β):
3          for i ∈ {1, 2} do
4              E_i ← new Enumerator(φ_i, β↾_{φ_i});
5              A_i ← new Array(E_i.first());
6          end
7          E ← new Object(φ1, φ2, β, A1, A2, E1, E2);
8          return E;
9      end
10     function E.update(t = Ra1...ar, p):
11         A ← {i ∈ {1, 2} | t affects E_i};
12         for i ∈ A do
13             s_i, (τ_i^1, p_i^1) ← E_i.update(t, p);
14             replace the only tuple in A_i with (τ_i^1, p_i^1);
15         end
16         for i ∉ A do
17             s_i, (τ_i^1, p_i^1) ← E_i.first();
18         end
19         s ← (UNSUP if s1 = s2 = UNSUP; SUP otherwise);
20         return s, (τ_1^1, p_1^1) ⊙ (τ_2^1, p_2^1);
21     end
22 end
```

Algorithm 3: Update mode of `AndEnumerator`(φ, β)

```
1  Class AndEnumerator:
2      function E.enumerate():
3          E.Q ← a new, empty priority queue;
4          E.Q.enqueue(((1, 1), E.A1[1] ⊙ E.A2[1]));
5          return E.next();
6      end
7      function E.next():
8          ((k, J_k), T^{k,J_k}) ← E.Q.pop();
9          if T^{k,J_k}.p = 0 then
10             restore default state;
11             return (EOE, 0);
12         end
13         if J_k = 1 then
14             t_1^{k+1} ← E.E1.next();
15             E.A1[k + 1] ← t_1^{k+1};
16             t_2^1 ← E.A2[1];
17             E.Q.enqueue(((k + 1, 1), t_1^{k+1} ⊙ t_2^1));
18         end
19         if |E.A2| < J_k + 1 then
20             E.A2[J_k + 1] ← E.E2.next();
21         end
22         t_1^k ← E.A1[k];
23         t_2^{J_k+1} ← E.A2[J_k + 1];
24         E.Q.enqueue(((k, J_k + 1), t_1^k ⊙ t_2^{J_k+1}));
25         return T^{k,J_k};
26     end
27 end
```

Algorithm 4: Enumeration mode of `AndEnumerator`(φ, β)

length at most $m_1 := |[\![\varphi_1]\!]^{\beta_1}|$. During each next() call, we need at most one pop and at most two insertions. The trivial bound of $m_1 \leq n_a^{|W_1|}$ where $n_a := |\mathbf{adom}|$ suffices to show that the operations on the queue Q need only time $O(\log n_a) = O(\log n_a^{|W|})$ Therefore, we can give the time bound of $O(\log ||D||)$ per next() call. This gives us the lemma. □

Lemma 4.8. *Let (φ_1, W_1) and (φ_2, W_2) be two queries where φ_1 and φ_2 are independent and $W_1 \cap W_2 = \emptyset$. Assume that for any $\beta_1 : C_1 \to \mathbf{dom}$, $\beta_2 : C_2 \to \mathbf{dom}$, there exist dynamic ranked enumeration algorithms E_1, E_2 that enumerate $[\![(\varphi_1, W_1)]\!]^{D,\beta_1}$ and $[\![(\varphi_2, W_2)]\!]^{D,\beta_2}$, respectively, with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay.*

Then, there exists a dynamic ranked enumeration algorithm **E** *of type* AndEnumerator(φ, β) *that enumerates $[\![(\varphi, W)]\!]^{D,\beta}$ with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay, where $\varphi = \varphi_1 \vee \varphi_2$, $W = W_1 \cup W_2$, and any $\beta : C \to \mathbf{dom}$.*

The proof of Lemma 4.8 is completely analog to Lemma 4.7, only the output probability is computed as $p^{(1,1)} := 1 - (1 - p_1) \cdot (1 - p_2)$.

4.1.4 Proving the Theorem.

Proof of Theorem 4.2. We prove the theorem by structural induction over r-liftable formulas (Definition 4.1). The base case (1) follows immediately from Theorem 3.8. The induction step for the rules (2), (3), and (4) follows from Lemma 4.6, 4.7, and 4.8, respectively. □

4.1.5 Top-k Enumeration.

Often, users of probabilistic databases are only interested in the most probable output tuples. This motivates the research area of *top-k enumeration*, where users do not request the full query result $[\![\varphi]\!]^{\beta}$, but only the k output tuples with the highest probability. Here, $k \in \mathbb{N}_1$ is a user-provided parameter.

We show that our enumerators support top-k enumeration. Let abort() be a new method (in addition to next()) that our enumerators are to implement in enumeration mode. Upon a call to abort(), an enumerator should clean up the enumeration state, go back into update mode, and return the special end-of-enumeration tuple (EOE, 0). We let "abort time" be the time that is needed to handle a call to abort(). It is clear that this allows users to perform top-k enumeration, even without explicitly giving us the parameter k beforehand: They just start the enumeration as usual and abort it after they have seen enough output tuples.

Lemma 4.9. *Let* **E** *be one of the enumerator data structures we introduced for the proof of Theorem 4.2, and let $k \in \mathbb{N}$ be the number of output tuples that was produced prior to a call to* abort(). *If all child enumerators of* **E** *support top-k enumeration with $O(k)$ abort time, then* **E** *also supports top-k enumeration with $O(k)$ abort time.*

In uninterrupted enumeration, the enumeration state is built up and built back down over the course of multiple next() calls; this is why we can give logarithmic delay, but need abort time linear in k. Typically, values of k will be small and independent of the size of the database, e.g. $k = 10$ or $k = 100$.

Proof. We prove the bound on the abort time. Enumerators of all types need only constant time to reset their own state on an abort() call. However, Rule (2)-based enumerators need to propagate the abort() call to $s := |Q_{\text{enum}}|$ many child enumerators. Let $k_1, ..., k_s$ be the number of output tuples that each of the s child enumerators produced. Note that $s \leq k$, and in particular that $k = \sum_{c \in [s]} k_c$. It follows that the abort time of the Rule (2)-based enumerator is in $O(\sum_{c \in [s]} k_c) = O(k)$. □

Remark 4.10. Note that the space required by our data structure is $O(||D||)$ in update mode. In the enumeration mode, the new priority queues grow linear in the number of generated output tuples and are cleaned up afterwards. It follows that top-k enumeration requires total space $O(||D|| + k)$, which is another incentive to avoid running the enumeration for prohibitively large k.

4.1.6 Open-World Semantics.

In tuple-independent probabilistic databases, the conventional semantics are "closed-world", which means that ground tuples t that are not present in the database are assumed to have probability $p(t) = 0$. Ceylan et al. [7] describe several queries where output probabilities derived with closed-world semantics go against intuition, and propose *open-world* semantics to address this problem: For open-world semantics, a probabilistic database instance D is coupled with a *threshold probability* $\lambda \in [0, 1]$. A *λ-completion* of a probabilistic database D is another probabilistic database based on D, where we add every ground tuple that is not present in D with some probability $p \in [0, \lambda]$. Now, a (finite)[3] open-world probabilistic database (D, λ) defines a set of probability distributions $\mathcal{P}$ such that a distribution P is contained in $\mathcal{P}$ iff P is induced by some λ-completion of D. Queries (φ, W) on (D, λ) yield result valuations τ that do not have a single probability value assigned to them, but an interval $[\underline{p}, \overline{p}]$, where

$$\underline{p}(\varphi) := \min_{P \in \mathcal{P}} P(\tau) \quad \text{and} \quad \overline{p}(\varphi) := \max_{P \in \mathcal{P}} P(\tau).$$

We will write $[\![\varphi]\!]_c^{\beta}$ for the result of query (φ, β) under closed-world semantics and $[\![\varphi]\!]_o^{\beta}$ for the result of the same query under open-world semantics. For further discussion, we refer the reader to [7].

When we rank the result tuples (τ, p) of $[\![\varphi]\!]_o^{\beta}$ with $p = [\underline{p}, \overline{p}] \subset \mathbb{Q}$ by their probabilities, we rank them (in descending order) by their upper-bound probability $\overline{p}$.

Theorem 4.11. *If a query (φ, W) is r-liftable with our rules under closed-world-semantics (see Theorem 4.2), then it is also r-liftable under open-world-semantics.*

We prepare the proof with the following definition:

Definition 4.12. If we evaluate a query (φ, W) over the empty database D_{u} under open-world semantics, then all elements in the result $[\![\varphi]\!]_o^{D_{\mathrm{u}},\beta}$ have the same probability interval $p_{\text{shared}} \subset \mathbb{Q}$ (which is independent of β). We define the function $p_{\mathrm{u}}(\varphi)$ under open-world semantics to be an interval $p_{\mathrm{u}}(\varphi) \subset \mathbb{Q}$; and we define $p_{\mathrm{u}}(\varphi) := [0, 0]$ if $[\![\varphi]\!]^{D_{\mathrm{u}},\beta} = \emptyset$, and $p_{\mathrm{u}}(\varphi) := p_{\text{shared}}$ otherwise.

Proof. Our algorithm from Theorem 4.2 supports open-world semantics with only slight modifications: For computing the probability intervals of unsupported queries (φ, W), instead of using $p_{\mathrm{u}}(\varphi)$ under closed-world semantics, we will use $p_{\mathrm{u}}(\varphi)$ under open-world semantics (as defined in Definition 4.12). The probability

[3] A different notion of open-world semantics has recently been proposed for probabilistic databases over an *infinite* domain [12, 13]. However, in this work we only consider finite domains.

interval of a supported ground tuple $t = Ra_1...a_r$ with probability $p_t \in \mathbb{Q}$ will be the single-element interval $[p_t, p_t] = \{p_t\} \subset \mathbb{Q}$.

In all output tuples (τ, p), p will now be an interval $p = [\underline{p}, \overline{p}] \subset \mathbb{Q}$, instead of a scalar probability value. The arithmetics we performed on probability values can be transferred to intervals with constant overhead: For example, for two intervals p, p' we can compute $p \cdot p' := [\underline{p} \cdot \underline{p'}, \overline{p} \cdot \overline{p'}]$ and $1 - p := [1 - \overline{p}, 1 - \underline{p}]$. We let all priority queues sort in descending order, based on the upper bound probability. The rest of the proof for closed-world semantics is not affected by the change to open-world semantics; this gives us the theorem. □

4.2 Application to Non-Repeating CQs

In this subsection, we show that our lifted inference rules for dynamic ranked enumeration (Definition 4.1) are complete for the class of non-repeating conjunctive queries: For every non-repeating conjunctive query we can either apply Theorem 4.2 and efficiently maintain the ranked query result, or (under the assumption that the OMv conjecture holds) there is *no* efficient dynamic ranked enumeration algorithm for this query.

A *conjunctive query* (CQ) of schema σ is a query (φ, W), where

$$\varphi = \exists y_1 \cdots \exists y_\ell \ (\psi_1 \wedge \cdots \wedge \psi_d),$$
$$W = \text{free}(\varphi),$$

$\ell \in \mathbb{N}_0$, $d \in \mathbb{N}_1$, and ψ_j is an atomic query of schema σ for every $j \in [d]$. Since $W = \text{free}(\varphi)$, we abbreviate a conjunctive query (φ, W) as φ. A CQ φ is *non-repeating* or *self-join-free* if no two atoms of φ share the same relational symbol.

Berkholz et al. [4] introduced the notion of a *q-hierarchical CQ*, which will perfectly capture those non-repeating CQs that are efficiently enumerable:

Definition 4.13 ([4]). A CQ φ is *q-hierarchical* if, for any two variables $x, y \in \text{vars}(\varphi)$, the following is satisfied:

(1) $\text{atoms}(x) \subseteq \text{atoms}(y)$ or $\text{atoms}(y) \subseteq \text{atoms}(x)$ or $\text{atoms}(x) \cap \text{atoms}(y) = \emptyset$, and
(2) if $x \in \text{free}(\varphi)$ and $\text{atoms}(x) \subsetneq \text{atoms}(y)$, then $y \in \text{free}(\varphi)$,

where $\text{atoms}(x)$ denotes the set of atoms in φ that contain x.

We first show that all non-repeating q-hierarchical CQs are efficiently enumerable.

Lemma 4.14. *Let φ be a non-repeating CQ. The following statements are equivalent:*

(1) φ can be transformed into an equivalent r-liftable query φ'.
(2) φ is q-hierarchical.

Proof. Consider a rooted syntax tree $\mathcal{T}$ for the conjunction $\psi_1 \wedge \cdots \wedge \psi_d$ in the formula φ. $\mathcal{T}$ has internal nodes $\wedge$ that have two children each, and leaf nodes ψ_i. Note that because of the commutativity and associativity of $\wedge$, any two such trees $\mathcal{T}, \mathcal{T}'$ that have the same leaf set are equivalent. For any node t in $\mathcal{T}$, let $D(t)$ be the set of descendants of t in $\mathcal{T}$ (including t itself). We also define $\text{atoms}(t) \subseteq D(t)$ to be the set of leaves (i.e. atoms ψ_i) in $D(t)$. Let us define the *q-condition* for such a syntax tree $\mathcal{T}$: The q-condition is satisfied on $\mathcal{T}$ if

(1) for every variable $x \in \text{vars}(\varphi)$, there is a node t_x in $\mathcal{T}$, such that $\text{atoms}(x) = \text{atoms}(t_x)$, and
(2) if $x \in \text{free}(\varphi)$ and $y \in \text{vars}(\varphi) \setminus \text{free}(\varphi)$, then t_x is no descendant of t_y in $\mathcal{T}$.

It is easy to see that if the q-condition is satisfied on a syntax tree $\mathcal{T}$ for a CQ φ, then φ is q-hierarchical. The converse also holds: If φ is q-hierarchical, then we can rearrange the syntax tree $\mathcal{T}$ of φ into an equivalent syntax tree $\mathcal{T}'$ that satisfies the q-condition.

It remains to show the equivalence between (a) a syntax tree satisfying the q-condition, and (b) the corresponding formula being r-liftable. Assume that a syntax tree $\mathcal{T}$ satisfies the q-condition. Because φ is non-repeating, x is a separator variable in the subformula rooted in node t_x, and every two sibling nodes t_1, t_2 represent independent subformulas. What remains in order to apply our rules is only to "pull in" all existential quantifiers $\exists x$ to the position of node t_x. For the other direction, see that if our rules were applied successfully to a formula with syntax tree $\mathcal{T}$, then (after we "pulled out" all existential quantifiers) that tree must satisfy the q-condition. □

It remains to show that non-repeating conjunctive queries that are *not* q-hierarchical are not efficiently enumerable. The following theorem from Berkholz et al. [4] provides a lower bound on the delay and update time for enumerating the result of a non-q-hierarchical CQ on a standard (non-probabilistic) database. It relies on the OMv-conjuncture [14], an algorithmic assumption on the hardness of the online matrix-vector multiplication problem.

Theorem 4.15 ([4]). *Fix a number $\varepsilon > 0$ and a non-repeating CQ φ. If φ is not q-hierarchical, then there is no algorithm with arbitrary preprocessing time and $O(\|D\|^{\frac{1}{2}-\varepsilon})$ update time that enumerates the query result on a deterministic database D in any order with $O(\|D\|^{\frac{1}{2}-\varepsilon})$ delay, unless the OMv-conjecture fails.*

If there is no algorithm that can enumerate a result set on a deterministic database in any order, then adding the requirements of probabilistic inference and ranking will not make the problem easier. We directly get our theorem:

Theorem 4.16. *Let φ be a non-repeating CQ, D a probabilistic database, and $\varepsilon > 0$. The following dichotomy holds:*

(1) If φ is q-hierarchical, then there is a data structure that supports dynamic ranked enumeration of the query result $[\![\varphi]\!]^D$ with $O(\log ||D||)$ update time and $O(\log ||D||)$ delay.
(2) If φ is not q-hierarchical, then there is no dynamic ranked enumeration algorithm with $O(\|D\|^{\frac{1}{2}-\varepsilon})$ update time, and $O(\|D\|^{\frac{1}{2}-\varepsilon})$ delay, unless the OMv-conjecture fails.

Proof. The first statement follows directly from Lemma 4.14 and Theorem 4.2, the second directly from Theorem 4.15. □

4.3 On Dynamic Ranked Enumeration for Inclusion-Exclusion Lifting

We have shown that the lifted inference rules (ind-$\vee$) and (ind-$\wedge$) for Boolean queries translate to dynamic ranked enumeration (rule (4) and (3) in Definition 4.1). In addition, the rule (2) for free variables bases on the same syntactic criteria as (ind-$\exists$) and (ind-$\forall$) in the Boolean setting.

The question that we address in this section is whether an analog of the inclusion-exclusion rules (incl-excl1), (incl-excl2), and (M)

for ranked enumeration might be possible as well. Note that for Boolean UCQs using inclusion-exclusion is essential to characterise the queries that can be maintained under updates. We argue that for dynamic ranked enumeration applying inclusion-exclusion is much more challenging. To illustrate this, we consider the following simple family of non-repeating UCQs (φ_k, W_k) that can be evaluated in polynomial time:

$$\varphi_k = \bigvee_{i\in[k]} (S_i x \wedge T_i y), \quad W_k = \{x, y\}.$$

By inclusion-exclusion we get that

$$[\![(\varphi_k, W_k)]\!]^D = \{((a,b), p_{a,b}) : a, b \in \mathbf{dom};\ p_{a,b} > 0\}, \tag{1}$$

$$\text{where } p_{a,b} = \textstyle\sum_{\emptyset\neq I\subseteq[k]} (-1)^{|I|+1} \cdot \prod_{i\in I} \big(p(S_i a) \cdot p(T_i b)\big). \tag{2}$$

There is a simple algorithm for dynamic ranked evaluation that achieves constant delay and $O(n \log n)$ update time by maintaining an ordered list of output pairs and inserting/deleting all $O(n)$ affected pairs after an update. Moreover, a data structure supporting *unordered* constant-delay enumeration of result tuples can be maintained with constant update time. Achieving logarithmic update time for dynamic ranked enumeration seems to be much harder and can be related to the following intriguing problem in computational geometry.

In the d-dimensional dynamic linear programming problem (also known as *dynamic convex hull* or *linear optimization query*) the task is to maintain a set U of d-dimensional points in a data structure that allows the following operations.

(1) *Update*: Insert or remove single point from U.
(2) *LP-query*: Given a vector $(v_1, \ldots, v_d)$, output a point $(u_1, \ldots, u_d) \in U$ such that $\sum_i v_i u_i$ is maximal.

Solving an LP-query is equivalent to solving a linear program with d variables over the convex hull of the point set. For $d = 2$ and $d = 3$ this problem can be solved with polylogarithmic (amortized) update and query time by maintaining a representation of the convex hull of the point set [8, 30]. While for $d > 3$ different dynamic data structures exist [23, 24, 31], it is open, whether there is a data structure that achieves polylogarithmic update and query time [8].

Note that the objective function $\sum_i v_i u_i$ is already quite similar to the expression (2) when $|I| = 1$. For a formal reduction we want to neglect the other summands for $|I| > 1$ and for this reason we reduce from a *discrete* variant of the problem, where the points are taken from $[N]^d$ for some fixed integer N and query vector has integral values between $-N$ and N. This formal restriction is equivalent to assuming that the point set and the query are represented with bounded precision.

Theorem 4.17. *Let $N \in \mathbb{N}$. If there is a ranked enumeration algorithm for φ_d with logarithmic update time and logarithmic delay, then there is a data structure that maintains a point set $U \subset [N]^d$ with $O(\log |U|)$ update time and supports LP-queries $\overline{v} \in [-N, N]^d$ in time $O(\log |U|)$.*

Proof. To solve dynamic linear programming queries using probabilistic query evaluation we maintain 2^d databases $D_{\overline{s}}$ with $\overline{s} = (s_1, \ldots, s_d) \in \{-1, 1\}^d$ that represent the point set and will be used for answering LP-queries. With each point $\overline{u} = (u_1, \ldots, u_d) \in U$ we identify an element $a_{\overline{u}} \in \mathbf{dom}$ and maintain the ground tuples $S_1(a_{\overline{u}}), \ldots, S_d(a_{\overline{u}})$ in every database $D_{\overline{s}}$ with probabilities

$$p(S_i(a_{\overline{u}})) = \begin{cases} \frac{u_i}{N^2 \cdot 2^d}, & \text{if } s_i = 1 \\ \frac{1}{N2^d} - \frac{u_i}{N^2 \cdot 2^d}, & \text{if } s_i = -1. \end{cases} \tag{3}$$

Now let $(v_1, \ldots, v_d)$ by an LP-query and set $s_i := \operatorname{sign}(v_i)$. We insert into $D_{\overline{s}}$ the ground atoms $T_1(b), \ldots, T_d(b)$ with probabilities $p(T_i(b)) = \frac{|v_i|}{N^2 \cdot 2^d}$. Now we start the ranked enumeration for φ_d on $D_{\overline{s}}$ and let $(a_{\overline{u}}, b)$ be the first tuple. We report $\overline{u}$ as an answer to the LP-query and delete the ground tuples $T_1(b), \ldots, T_d(b)$ from $D_{\overline{s}}$.

It is clear that the process guarantees logarithmic update and query time and it remains to prove correctness. By construction we have

$$\begin{aligned} p_{a_{\overline{u}},b} &= \sum_{i\in[d]} p(S_i a_{\overline{u}}) \cdot p(T_i b) \quad - r(\overline{u}, \overline{v}) \\ &= \sum_{i\in[d]} \tfrac{u_i}{N^2 \cdot 2^d} \cdot \tfrac{v_i}{N^2 \cdot 2^d} + \sum_{i:\ s_i=-1} \tfrac{1}{N2^d} \cdot \tfrac{|v_i|}{N^2 \cdot 2^d} \quad - r(\overline{u}, \overline{v}) \end{aligned}$$

Where $r(\overline{u}, \overline{v}) \leq 2^d \cdot (\frac{1}{N2^d})^{-4} = N^{-4} 2^{-3d}$ as every remaining summand is a product of at least four probabilities smaller than $\frac{1}{N2^d}$. Now we can conclude the proof by showing that if $\overline{u}$ has a larger score than some other point $\overline{w} \in U$, it will receive a larger probability.

$$\begin{aligned} & \textstyle\sum_i v_i u_i > \sum_i v_i w_i \\ \Leftrightarrow \quad & \textstyle\sum_i v_i u_i \geq \sum_i v_i w_i + 1 \\ \Leftrightarrow \quad & \tfrac{1}{N^4 \cdot 2^{2d}} \textstyle\sum_i v_i u_i \geq \tfrac{1}{N^4 \cdot 2^{2d}} \sum_i v_i w_i + \tfrac{1}{N^4 \cdot 2^{2d}} \\ \Leftrightarrow \quad & p_{a_{\overline{u}},b} \geq p_{a_{\overline{w}},b} + r(\overline{w}, \overline{v}) - r(\overline{u}, \overline{v}) + \tfrac{1}{N^4 \cdot 2^{2d}} \\ & > p_{a_{\overline{w}},b} \end{aligned}$$

□

5 CONCLUSION

We studied the problem of probabilistic query evaluation under updates for both Boolean and non-Boolean queries. We first considered the known lifted inference rules for Boolean queries and demonstrated that we can maintain, in constant update time, the query result for all Boolean queries that are liftable using these rules. Then, we extended the lifted inference rules to non-Boolean queries. Here, our interest was to rank the set of output tuples by their probability. We showed that we can enumerate the ranked output tuples with logarithmic delay, and that we can efficiently maintain this ranked query result under updates.

The overall goal of this line of research is to understand for which classes of queries efficient dynamic query evaluation is possible. While for Boolean UCQs a complete characterisation is inherited from Dalvi and Suciu's dichotomy theorem [10], we only have partial results for dynamic ranked enumeration for non-Boolean UCQs. In particular, we showed that our method is complete for the class of non-repeating conjunctive queries, where we can either apply our method, or (conditioned on the OMv-conjecture) no algorithm with polylogarithmic delay and polylogarithmic update time is possible.

One obvious open problem is to extend this characterisation to larger classes of queries, either by proving tight (conditional) lower bounds or by extending our methods.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 414325841; 431183758.

REFERENCES

[1] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, pages 111:1–111:15, 2017. URL https://doi.org/10.4230/LIPIcs.ICALP.2017.111.

[2] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. In Benny Kimelfeld and Yael Amsterdamer, editors, *Proceedings of the 21st International Conference on Database Theory (ICDT 2018)*, pages 5:1–5:18, 2018. URL https://doi.org/10.4230/LIPIcs.ICDT.2018.5.

[3] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2019)*, pages 89–103, 2019. URL https://doi.org/10.1145/3294052.3319702.

[4] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2017)*, pages 303–318, 2017. URL https://doi.org/10.1145/3034786.3034789.

[5] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs under updates and in the presence of integrity constraints. In Benny Kimelfeld and Yael Amsterdamer, editors, *Proceedings of the 21st International Conference on Database Theory (ICDT 2018)*, pages 8:1–8:19, 2018. URL https://doi.org/10.4230/LIPIcs.ICDT.2018.8.

[6] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018. URL https://doi.org/10.1145/3232056.

[7] İsmail İlkan Ceylan, Adnan Darwiche, and Guy van den Broeck. Open-world probabilistic databases. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR 2016)*, pages 339–348, 2016. URL http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12908.

[8] Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *Journal of the ACM*, 57(3):16:1–16:15, 2010. URL https://doi.org/10.1145/1706591.1706596.

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

[10] Nilesh N. Dalvi and Dan Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *Journal of the ACM*, 59(6):30:1–30:87, 2012. URL https://doi.org/10.1145/2395116.2395119.

[11] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. In Ke Yi and Zhewei Wei, editors, *Proceedings of the 24th International Conference on Database Theory (ICDT 2021)*, pages 5:1–5:19, 2021. URL https://doi.org/10.4230/LIPIcs.ICDT.2021.5.

[12] Martin Grohe and Peter Lindner. Probabilistic databases with an infinite open-world assumption. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2019)*, pages 17–31, 2019. URL https://doi.org/10.1145/3294052.3319681.

[13] Martin Grohe and Peter Lindner. Infinite probabilistic databases. In Carsten Lutz and Jean Christoph Jung, editors, *Proceedings of the 23rd International Conference on Database Theory (ICDT 2020)*, pages 16:1–16:20, 2020. URL https://doi.org/10.4230/LIPIcs.ICDT.2020.16.

[14] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC 2015)*, pages 21–30, 2015. URL http://doi.acm.org/10.1145/2746539.2746609.

[15] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 2017)*, pages 1259–1274, 2017. URL https://doi.org/10.1145/3035918.3064027.

[16] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *Proc. VLDB Endow.*, 11(7):733–745, 2018. doi: 10.14778/3192965.3192966. URL http://www.vldb.org/pvldb/vol11/p733-idris.pdf.

[17] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11:1–11:58, 2008. URL https://doi.org/10.1145/1391729.1391730.

[18] Abhay Kumar Jha and Dan Suciu. Knowledge compilation meets database theory: Compiling queries to decision diagrams. *Theory Comput. Syst.*, 52(3):403–440, 2013. URL https://doi.org/10.1007/s00224-012-9392-5.

[19] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020. URL https://doi.org/10.1145/3396375.

[20] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2020)*, pages 375–392, 2020. URL https://doi.org/10.1145/3375395.3387646.

[21] Dietrich Kuske and Nicole Schweikardt. Gaifman normal forms for counting extensions of first-order logic. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, pages 133:1–133:14, 2018. URL https://doi.org/10.4230/LIPIcs.ICALP.2018.133.

[22] Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS 2014)*, pages 67:1–67:10, 2014. URL https://doi.org/10.1145/2603088.2603137.

[23] Jirí Matousek. Linear optimization queries. *Journal of Algorithms*, 14(3):432–448, 1993. URL https://doi.org/10.1006/jagm.1993.1023.

[24] Jirí Matousek and Otfried Schwarzkopf. Linear optimization queries. In David Avis, editor, *Proceedings of the 8th Annual Symposium on Computational Geometry*, pages 16–25, 1992. URL https://doi.org/10.1145/142675.142683.

[25] Mikaël Monet. Solving a special case of the intensional vs extensional conjecture in probabilistic databases. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2020)*, pages 149–163, 2020. URL https://doi.org/10.1145/3375395.3387642.

[26] Mikaël Monet and Dan Olteanu. Towards deterministic decomposable circuits for safe queries. In Dan Olteanu and Barbara Poblete, editors, *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2018)*, 2018. URL http://ceur-ws.org/Vol-2100/paper19.pdf.

[27] Matthias Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2018)*, pages 769–778, 2018. URL https://doi.org/10.1145/3209108.3209144.

[28] Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2018)*, pages 179–191, 2018. URL https://doi.org/10.1145/3196959.3196961.

[29] Dan Olteanu and Hongkai Wen. Ranking query answers in probabilistic databases: Complexity and efficient algorithms. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE 2012)*, pages 282–293, 2012. URL https://doi.org/10.1109/ICDE.2012.61.

[30] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981. URL https://doi.org/10.1016/0022-0000(81)90012-X.

[31] Edgar A. Ramos. Linear programming queries revisited. In Siu-Wing Cheng, Otfried Cheong, Pankaj K. Agarwal, and Steven Fortune, editors, *Proceedings of the 16th Annual Symposium on Computational Geometry*, pages 176–181, 2000. URL https://doi.org/10.1145/336154.336198.

[32] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980. URL https://doi.org/10.1145/322217.322221.

[33] Dan Suciu. Probabilistic databases for all. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2020)*, pages 19–31, 2020. URL https://doi.org/10.1145/3375395.3389129.

[34] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. URL https://doi.org/10.2200/S00362ED1V01Y201105DTM016.

[35] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proc. VLDB Endow.*, 13(9):1582–1597, 2020. doi: 10.14778/3397230.3397250. URL http://www.vldb.org/pvldb/vol13/p1582-tziavelis.pdf.

[36] Guy van den Broeck and Dan Suciu. Query processing on probabilistic data: A survey. *Found. Trends Databases*, 7(3-4):197–341, 2017. URL https://doi.org/10.1561/1900000052.

Received December 2020