Top-k Query Processing in Probabilistic Databases with Non-Materialized Views

Maximilian Dylla
Iris Miliaraki
Martin Theobald

MPI–I–2012–5-002 June 2012

Authors' Addresses

Maximilian Dylla
Max-Planck-Institut für Informatik
Campus E1.4
D-66123 Saarbrücken

Iris Miliaraki
Max-Planck-Institut für Informatik
Campus E1.4
D-66123 Saarbrücken

Martin Theobald
Max-Planck-Institut für Informatik
Campus E1.4
D-66123 Saarbrücken

Abstract

In this paper, we investigate a novel approach of computing confidence bounds for top-k ranking queries in probabilistic databases with *non-materialized views*. Unlike prior approaches, we present an exact pruning algorithm for finding the top-ranked query answers according to their marginal probabilities *without* the need to first materialize all answer candidates via the views. Specifically, we consider conjunctive queries over multiple levels of select-project-join views, the latter of which are cast into Datalog rules, where also the rules themselves may be uncertain, i.e., be valid with some degree of confidence. To our knowledge, this work is the first to address *integrated data and confidence computations* in the context of probabilistic databases by considering confidence bounds over partially evaluated query answers with *first-order lineage formulas*. We further extend our query processing techniques by a tool-suite of scheduling strategies based on selectivity estimation and the expected impact of subgoals on the final confidence of answer candidates. Experiments with large datasets demonstrate drastic runtime improvements over both sampling and decomposition-based methods—even in the presence of recursive rules.

Keywords

Top-k Query Processing, Probabilistic Databases, Datalog,
First-Order Lineage

Contents

1 Introduction **3**
- 1.1 Contributions 6

2 Computational Model **7**
- 2.1 Probabilistic Database 7
- 2.2 Views 8
- 2.3 Queries 9
- 2.4 Lineage 10
- 2.5 Deductive Grounding with Lineage 10
- 2.6 Confidence Computations 13
- 2.7 Expressiveness 14

3 Confidence Bounds **15**
- 3.1 Bounds for Propositional Lineage 15
- 3.2 Bounds for First-Order Lineage 17

4 Subgoal Scheduling **20**
- 4.1 Selectivity Estimation 20
- 4.2 Impact of Subgoals 22
- 4.3 Benefit-oriented Subgoal Scheduling 23

5 Top-k Algorithm **24**
- 5.1 Top-k with Dynamic Subgoal Scheduling 24
- 5.2 SLD Resolution with Lineage Tracing 26
- 5.3 Final Result Ranking 27

6 Extensions **28**
- 6.1 Sorted Input Relations 28
- 6.2 Recursive Rules 29

7 Experiments **31**
7.1 Data Sets, Confidence Distributions, and Queries 31
7.2 Results 32

8 Related Work **37**

9 Conclusions **39**

A **40**
A.1 Data 40
A.1.1 MayBMS / Postgres 40
A.1.2 Trio 42
A.2 Queries 42
A.2.1 $Q1$ (non-repeating hierarchical, safe) 42
A.2.2 $Q2$ (repeating hierarchical, non-safe) 44
A.2.3 $Q3$ (head hierarchical, non-safe) 46
A.2.4 $Q4$ (general unsafe) 47
A.2.5 $Q5$ (one proof) 49
A.2.6 $Q6$ (three proofs) 49
A.2.7 $Q7$ (join of existential relations) 51
A.2.8 $Q8$ (non-read-once) 52
A.2.9 $Q9$ (nesting-depth = 1) 53
A.2.10 $Q10$ (nesting-depth = 2) 53
A.2.11 $Q11$ (nesting-depth = 3) 54
A.2.12 $Q12$ (with uncertain views) 54
A.2.13 $Q13$ (with uncertain views) 55
A.2.14 $Q14$ (recursion) 55
A.2.15 $Q15$ (recursion) 55
A.2.16 Confidence Distributions 55
A.2.17 Table Materialization 55

1 Introduction

Managing uncertain data via probabilistic databases (PDBs) has evolved as an established field of research, with a plethora of applications ranging from scientific data management, sensor networks, data integration, to knowledge management systems [36]. Despite the polynomial runtime complexity for the data computation step involved in finding probabilistic answer candidates, confidence computations for these answers are $\#\mathcal{P}$-hard already for fairly simple select-project-join (SPJ) queries [7, 28]. Thus, efficient strategies for confidence computation and early pruning of low-confidence query answers remain a key challenge for the scalable management of uncertain data.

Recent work on efficient confidence computations in PDBs has addressed this problem by either restricting the class of queries, i.e., by focusing on *safe query plans* [8], or by considering a specific class of tuple-dependencies, commonly referred to as *read-once functions* [32]. Intuitively, safe query plans denote a class of queries for which confidence computations can directly be coupled with the relational operators and thus be performed by an extensional query plan [36]. Read-once formulas, on the other hand, denote a class of propositional formulas which can be factorized (in polynomial time) into a form where every variable that represents a tuple in the database appears at most once, which again permits efficient confidence computations. If a query plan is safe, it permits for a decomposition into *independent-join* and *independent-project* operations (including restricted forms of *negation*) over the probabilistic input data, along which confidence computations can be performed in a bottom-up fashion [36]. [7] build upon this observation to derive a dichotomy of query plans for which confidence computations can be done either in polynomial time or are $\#\mathcal{P}$-hard.

While safe plans clearly focus on the characteristics of the query structure and read-once formulas focus on the logical dependencies among individual data objects derived from a query, top-k style pruning approaches, which are the subject of this work, have also been proposed as an alternative way of addressing confidence computations in PDBs [25, 24] . These approaches typically aim to efficiently identify the top-k most probable answers based on lower and upper bounds

Directed

	Director	*Movie*	p
t_1	Allen	MidnightInParis	0.7
t_2	Nolan	Inception	0.9
t_3	Tarantino	ReservoirDogs	0.8
t_4	Tarantino	PulpFiction	0.6

ActedIn

	Actor	*Movie*	p
t_5	Johansson	MatchPoint	0.9
t_6	DiCaprio	Inception	0.9
t_7	Travolta	PulpFiction	0.8
t_8	Roth	ReservoirDogs	0.6

Nominated

	Movie	*Award*	p
t_9	Inception	BestWriting	0.9
t_{10}	PulpFiction	BestDirector	0.6
t_{11}	PulpFiction	BestWriting	0.6

Category

	Movie	*Category*	p
t_{12}	Inception	Action	0.9
t_{13}	ReservoirDogs	Independent	0.5
t_{14}	PulpFiction	Crime	0.9

ν_1 : $\forall X, Y\ \mathit{KnownFor}(X,Y) \text{ :- } \exists Z\ \mathit{ActedIn}(X,Z), \mathit{Category}(Z,Y)$ 0.7

ν_2 : $\forall X, Y\ \mathit{KnownFor}(X,Y) \text{ :- } \exists Z\ \mathit{Directed}(X,Z), \mathit{Category}(Z,Y), \mathit{Won}(Z, \mathit{BestDirector})$ 0.8

ν_3 : $\forall X, Y\ \mathit{Written}(X,Y) \text{ :- } \mathit{Directed}(X,Y), \neg\mathit{Category}(Y, \mathit{Independent})$ 0.6

ν_4 : $\forall X\ \mathit{Won}(X, \mathit{BestWriting}) \text{ :- } \mathit{Nominated}(X, \mathit{BestWriting}), \neg\mathit{Category}(X, \mathit{Action})$ 0.6

ν_5 : $\forall X, Y\ \mathit{Won}(X,Y) \text{ :- } \mathit{Nominated}(X,Y)$ 0.4

Figure 1.1: Example movie database.

for their marginal probabilities without the need to compute the exact probabilities. Suciu et al. [25] addressed this by approximating the probabilities of the top-k answers using Monte-Carlo-style sampling techniques. Along the lines of these works, returning the correct set of top-k answers is guaranteed while the marginal probabilities of answer candidates are approximated only to the extent needed for computing this top-k set. Olteanu and Wen [24] have further developed the idea of decomposing propositional formulas for deriving confidence bounds based on partially expanded ordered binary decision diagrams (OBDDs), which can again be exploited by top-k algorithms for early candidate pruning. These top-k algorithms can effectively circumvent the need for exact confidence computations and can still—in many cases—return the set of top-ranked query answers in an exact way. However, as opposed to top-k approaches in traditional DBs [9], none of the former approaches saves on the data computation step that is required for finding potential answer candidates. All former approaches require extensive data materialization for queries with multiple nested subqueries or queries over multiple levels of potentially convoluted views.

In this paper, we specifically focus on the case when *views are not materialized*, and we are aiming to identify the top-ranked query answers, based on their marginal probabilities, even before the complete extensional input to these answers (and in particular that of the remaining answer candidates) has been seen by the query processor. Following the line of works on intensional query evaluation [11, 4, 30, 36], we employ lineage formulas to capture the logical dependencies between query answers, base tuples, and the views that were employed to

derive the answers. In contrast to all lineage models known to us, which consider lineage as propositional formulas [36], where each formula represents a single query answer, we introduce *first-order lineage formulas*, where each formula may represent an entire *set of query answers*. We describe methods for obtaining confidence bounds that hold for each answer candidate in such a set, thus allowing us to prune low-confidence answer candidates before they have been completely evaluated over the views. Our main observation is that each intermediate step of query processing can be unambiguously described and thus be captured by such a first-order lineage formula, which is our key for combining data and confidence computations in a probabilistic database setting. In particular, this allows us to adopt top-k-style pruning techniques known from traditional DBs [9, 15], which aim to avoid a complete scan of all input tuples that are potentially relevant for the top-k results based on the query plan. Moreover, we explicitly also allow our views to be uncertain. As a consequence, answers derived from an uncertain view are valid only with some degree of confidence even if the view's input tuples are not uncertain, which may greatly help in pruning answer candidates derived from low-confidence views. We illustrate our setting by the following example.

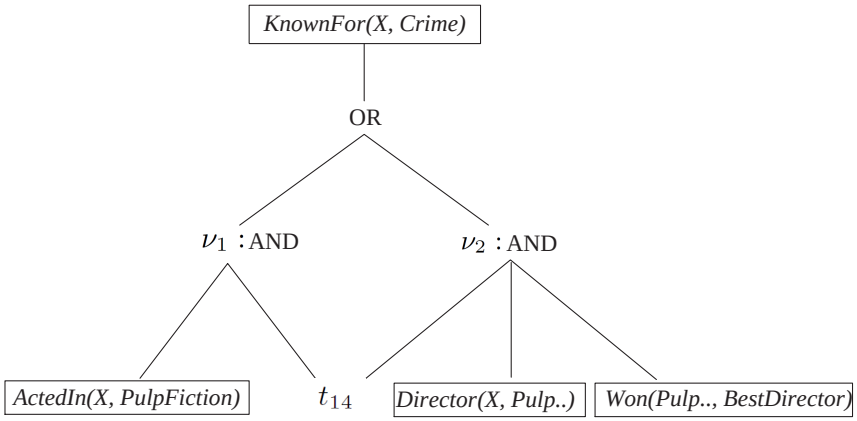


Figure 1.2: Lineage example.

Example 1 *Consider a probabilistic database that contains information about movie directors, actors, categories and award nominations as shown in Figure 1.1. View ν_1, for example, captures the case where actors are known for specific movie categories in which they acted. Likewise, from view ν_2 we may infer that directors who won an award for a movie of a specific category (e.g., $Action$) are likely, but not for sure, also known for this category. Consequently, Figure 1.2 depicts a partially evaluated lineage formula for the query $KnownFor(X, Crime)$ as a directed acyclic graph over the views and base tuples of Figure 1.1, thus asking for directors or actors X who are known for $Crime$ movies.*

This lineage DAG captures an intermediate step of processing the query via the views in a top-down fashion. We see that both ν_1 and ν_2 have only been partially resolved to the base tuple t_{14}, while all other literals in ν_1 and ν_2 (aka. "subgoals" in Datalog terminology) are still unexplored. At this point, we can already identify two intriguing questions which we may aim to answer even before further processing the query via the remaining views: (1) are there any high-confidence answers which we can obtain from views ν_1 and ν_2 at all; and (2) how can we develop bounds to efficiently identify the top-k most probable answers when processing the remaining literals? $\diamond$

1.1 Contributions

We summarize our contributions as follows:

- Current PDB engines perform data and confidence computations in two strictly separated phases. To our knowledge, our approach is the first to consider *integrated data and confidence computations*. Thus, early pruning of low-confidence answer candidates can greatly reduce data computations in PDBs with non-materialized views.
- We present a generic bounding approach for confidence computations over *first-order lineage formulas*. Our algorithm provides (1) *lower and upper bounds* for an individual query answer or for an entire set of answers if the query variables are not bound to constants yet. We show that (2) both our lower and upper bounds *converge monotonically* to the final confidences of the query answers as we gradually expand the formulas. Both (1) and (2) are key properties for building effective top-k-style pruning algorithms.
- Our approach allows for plugging in *different schedulers* which aim to select the subgoal which is most beneficial for top-k pruning at each query processing step. This benefit is estimated based on the expected selectivity, i.e., the expected number of answers, and the expected impact of a subgoal on the confidence of query answers.
- We present extensions for the case when *sorted input lists* for extensional relations are available and for adapting our pruning techniques to *recursive queries*.
- We present an extensive experimental evaluation and comparison to existing top-k pruning strategies in probabilistic databases. Moreover, to our knowledge, we are the first to report an improved runtime of our top-k algorithm in a probabilistic database setting compared to full query evaluations in a corresponding deterministic database setting.

2 Computational Model

In this section, we introduce our data model, which follows the common possible-worlds semantics [2] over tuple-independent probabilistic databases, along with the operations we allow over this kind of uncertain data. A summary of all symbols used in our notation, together with their description, is depicted in Table 2.1. Our computational model builds upon (and thus is consistent with) prior work on probabilistic databases [3, 5, 22, 33], and specifically upon the one considered in the context of *uncertain databases with lineage* [4, 30, 38], which is known to be *closed* and *complete* under the common semantics of relational operations.

2.1 Probabilistic Database

We define a *tuple-independent probabilistic database with uncertain views* $\mathcal{DB} = (\mathcal{T}, \mathcal{V}, p)$ as a triplet consisting of a set of *base tuples* $\mathcal{T}$, a set of *views* $\mathcal{V}$, and a *probability measure* $p : \mathcal{T} \cup \mathcal{V} \rightarrow [0, 1]$, which assigns a probability value $p(e)$ to each uncertain base tuple and view $e \in \mathcal{T} \cup \mathcal{V}$, respectively.[1] As in a regular database, we assume the set of base tuples $\mathcal{T}$ to be partitioned into a set of *extensional relations*. The probability value $p(e)$ denotes the confidence in the correctness of the tuple or view, i.e., a higher value $p(e)$ denotes a higher confidence in e being valid.

Uncertainty of base tuples is modeled by associating a Boolean random variable $\mathcal{X}_t$ with each base tuple $t \in \mathcal{T}$. The case when $\mathcal{X}_t = \mathit{true}$ denotes the probabilistic event that t is present in the probabilistic database. Analogously to base tuples, we model the uncertainty of views by introducing also a Boolean random variable $\mathcal{X}_\nu$ for each view $\nu \in \mathcal{V}$. Thus, $\mathcal{X}_\nu = \mathit{true}$ denotes the probabilistic event that view ν holds, which we assume to hold independently of other views or base tuples. Setting $p(\nu) < 1$ expresses that processing the view ν may deduce

[1] Usually a probabilistic database is defined as a probability distribution over possible instances of the database. In the case of a tuple-independent PDB (without any further constraints restricting the possible instances), the distribution corresponds to the one defined by Equation (2.1).

Symbol	Description
$\mathcal{T}$	Base tuples of all extensional relations
$\mathcal{V}$	Views defining intensional relations
$\mathcal{X}$	Boolean random variable
p	Probability measure
P	Marginal probability of a formula
ϕ, ψ	Propositional lineage formulas
Φ, Ψ	First-order lineage formulas
$\Phi_{[t \to true]}$	Restriction of Φ onto $t \to true$
$\bar{X}$	Tuple of variables and constants
$R^{\gamma}(\bar{X})$	Subgoal over relation R
$L(\bar{X})$	Literal (signed subgoal)

Table 2.1: Description of Symbols.

uncertain results and thus decreases the confidence of answers derived from ν.

We assume globally unique identifiers for base tuples and views. For convenience of notation, and if it is clear from the context, we will thus write t and ν to denote both the identifiers and the random variables $\mathcal{X}_t$ and $\mathcal{X}_\nu$ associated with t and ν, respectively.

Possible Worlds Semantics. A *possible world* $\mathcal{W} \subseteq (\mathcal{T} \cup \mathcal{V})$ is a subset of base tuples and views in $\mathcal{T}$ and $\mathcal{V}$, respectively. Since we assume independence among all Boolean random variables associated with tuples and views, the *probability* $P(\mathcal{W})$ of a possible world $\mathcal{W}$ is defined as follows.

$$P(\mathcal{W}) := \prod_{e \in \mathcal{W}} p(e) \prod_{e \notin \mathcal{W}} (1 - p(e)) \tag{2.1}$$

Intuitively, all tuples and views in $\mathcal{W}$ are valid (i.e., $true$) in the possible world $\mathcal{W}$, whereas all tuples and views in $(\mathcal{T} \cup \mathcal{V}) \setminus W$ are $false$ (i.e., they are not contained in the world $\mathcal{W}$). In the absence of further constraints, which would restrict the set of possible worlds (see, e.g., [7, 20]), each subset of base tuples and views $\mathcal{W} \in 2^{\mathcal{T} \cup \mathcal{V}}$ forms a possible world. Hence, there are exponentially many such possible worlds.

2.2 Views

We represent a view $\nu \in \mathcal{V}$ as a rule in Datalog notation. Hence $\mathcal{V}$ together with the set of base tuples (aka. "facts") $\mathcal{T}$ is also called a *Datalog program*. Consequently, we will denote the deductive query processing steps applied for processing these rules as *deductive grounding*. Syntactically, a Datalog rule is a disjunctive clause with a positive head literal and a conjunction of both positive

and negative literals in its body (see Figure 1.1 for examples). The views' head literals define a set of so-called *intensional relations*. An intensional relation may be defined via multiple rules; however, no extensional relation may occur also in the head of a rule.

Variables occurring in the head literal are universally quantified, while variables occurring only in the body literals are existentially quantified (see, e.g., ν_3 in Figure 1.1). Following common Datalog conventions, each variable that occurs in the head literal or in a negated body literal must also occur in at least one of the positive body literals. This form of *safe Datalog* programs ensures that grounding terminates, and the variables are properly bound to constants after grounding the rules. We remark that we do not focus on safe query plans [7, 8], and hence we do not pose any further restrictions on the views' shape. We further remark that this class of function-free, safe Datalog programs with negation (but without recursion) corresponds to the relational calculus for SPJ queries (including set operations such as union and difference, but without aggregations or other function calls) [1]. For any given instance of base tuples and non-recursive rules, data computations (but not confidence computations) are of polynomial runtime in the size of both the base tuples and rules [13]. Also, for the rest of this paper, we will use the terms *view* and *rule* interchangeably.

Uncertain Views. In contrast to classic probabilistic database approaches, we explicitly allow also our views to be uncertain. Considering, for example, ν_5 in Figure 1.1, we see that the probability value of 0.4 attached to ν_5 lets us express that not every movie nominated for an award indeed also wins this award. The uncertainty of this view (together with that of view ν_4) is thus propagated to every tuple in the intensional relation Won. Thus, the main difference of our data model to classic Datalog is the possible uncertainty of a rule (or view), which we model by introducing an additional random variable $\mathcal{X}_\nu$ for each view ν. In Subsection 2.4, we describe how the Boolean random variables associated with the views are encoded into the lineage formulas of tuples derived from these views.

2.3 Queries

We consider a *query* as a conjunction of first-order literals with arguments consisting of tuples of constants and free (aka. "distinguished") variables, which we will refer to as the *query variables*. Again, every variable occurring in a negated literal must also occur in at least one of the non-negated literals. Throughout this paper, we will explicitly refer to *query variables* as the distinguished variables that occur in the query literals. Tuples of constants which are bound to the tuples of query variables by the grounding procedure will yield the query answers.

2.4 Lineage

Propositional Lineage. In contrast to base tuples which are assumed to be independent, a derived tuple is completely defined via (and thus dependent of) the base tuples and views that were employed to derive that tuple. Thus, when completely grounded against the base tuples and views, we will refer to a derived tuple t directly via its *propositional lineage formula* ϕ_t. A propositional lineage formula thus captures the logical dependencies between base tuples, views, and the tuples derived from both these base tuples and views.

First-Order Lineage. As opposed to all probabilistic database approaches we are aware of (see, e.g., [4, 6, 14, 36]), which consider lineage only in propositional form, we more generally allow lineage to be a well-formed formula over a restricted class of first-order predicate logic. A *well-formed lineage formula* Φ may incorporate the constants $true$ and $false$, Boolean connectives ($\wedge$, $\vee$, $\neg$), Boolean (random) variables denoting tuples $t \in \mathcal{T}$ and views $\nu \in \mathcal{V}$, existential quantifiers ($\exists$), and first-order literals of the form $R^{\gamma}(\bar{X})$. Following common Datalog terminology, we refer to a first-order literal $R^{\gamma}(\bar{X})$ as a *subgoal*, where R denotes the relation name and $\bar{X}$ is a tuple consisting of both constants and variables. In what follows, these subgoals will represent yet unexplored (i.e., not yet grounded) parts of the lineage formula. We employ *adornments* in the form of a superscript γ of a subgoal to denote which variables of a subgoal are bound or free [1].

As opposed to propositional lineage, a first-order lineage formula is able to capture any intermediate step of the grounding procedure. If at least one query variable in a first-order lineage formula is not yet bound to a constant, the lineage formula represents a (possibly empty) set of query answers. In Figure 1.2, for example, the query variable X is not yet bound in two of the subgoals $ActedIn(X, PulpFiction)$ and $Directed(X, PulpFiction)$.

2.5 Deductive Grounding with Lineage

We next provide an inductive definition of *lineage* which is obtained from grounding a subgoal $R^{\gamma}(\bar{X})$ over uncertain views $\mathcal{V}$ and uncertain base tuples $\mathcal{T}$. The definition is based on two rewriting rules which follow the general course of a top-down grounding procedure. We are choosing top-down grounding over bottom-up grounding in order to be able to save data computations, i.e., to avoid touching base tuples of lower ranked answers whenever possible. Later, in Section 5, we provide a grounding algorithm, based on *SLD resolution* [1], which implements the two rewriting rules.

Rule (1) *(Disjunctive Lineage)* Let $R^{\gamma}(\bar{X})$ be a subgoal, and let $\bar{X}$ be a tuple of constants and variables not bound in γ. Then grounding $R^{\gamma}(\bar{X})$ over views $\mathcal{V}$

and base tuples $\mathcal{T}$ yields a disjunction over the lineages of base tuples or tuples derived from views that unify with $R^\gamma(\bar{X})$.

$$\Phi\left(R^\gamma(\bar{X})\right) := \begin{cases} \bigvee_\nu \left(\Phi(\nu)\right) & \bullet \text{ if } R \text{ is intensional and } head(\nu) \text{ of } \nu \in \mathcal{V} \text{ unifies with } R^\gamma(\bar{X}) \\ \bigvee_t \mathcal{X}_t & \bullet \text{ if } R \text{ is extensional and } t \in \mathcal{T} \text{ unifies with } R^\gamma(\bar{X}) \\ false & \bullet \text{ else} \end{cases}$$

Rule (2) *(Conjunctive Lineage)* Let $R^\gamma(\bar{X})$ be a subgoal, let $\bar{X}$ be a tuple of constants and variables not bound in γ, let

$$\nu:\ \forall \bar{X}'\ R(\bar{X}_0) \text{ :- } \exists \bar{X}''\ L_1(\bar{X}_1), \ldots, L_n(\bar{X}_n)$$

be a safe Datalog rule that unifies with $R^\gamma(\bar{X})$, and let $\bar{X}_0, \ldots, \bar{X}_n$ be tuples of constants and existentially quantified variables. Then grounding $R^\gamma(\bar{X})$ against ν yields a conjunction over the lineages of literals $L_1, \ldots, L_n$ in the body of ν, including an additional conjunction with the Boolean (random) variable $\mathcal{X}_\nu$ associated with ν.

$$\Phi\left(\nu\right) := \mathcal{X}_\nu \wedge \exists \bar{X}'' \left(\bigwedge_{i=1,\ldots,n} \begin{cases} \Phi\left(R_i^\gamma(\bar{X}_i)\right) & \bullet \text{ if } L_i = \ R_i \\ \neg\left(\Phi\left(R_i^\gamma(\bar{X}_i)\right)\right) & \bullet \text{ if } L_i = \neg R_i \end{cases} \right)$$

Query Processing. We always start the grounding procedure from a query literal. For a subgoal $R^\gamma(\bar{X})$ over an extensional relation R, only Rule (1) applies. It replaces $R^\gamma(\bar{X})$ by either a disjunction of Boolean variables representing base tuples or by the constant $false$, if no such tuples exist (which corresponds to the "negation-as-failure" semantics in Datalog[1]). If R is intensional, Rule (1) is utilized to create a disjunction over all rules with a head literal that unifies with the subgoal. Then, the additional application of Rule (2) results in a conjunction of literals in a rule's body, where existential quantifiers over the variables that occur in the rule's body are added. This process, known as SLD resolution, is repeated by using the body literals of the rule as new subqueries in the following grounding steps.

Creating Query Answers. If a tuple of arguments $\bar{X}$ in a subgoal $R^\gamma(\bar{X})$ becomes bound to one or more tuples of constants $\bar{C}_1, \ldots, \bar{C}_n$, we distinguish two cases. First, if $R^\gamma(\bar{X})$ relates to a top-level query literal, then each distinct tuple $\bar{C}_i$ corresponds to a new query answer and its lineage is copied correspondingly. Second, if $\bar{X}$ contains existentially quantified variables, then these can be eliminated through a standard quantifier elimination step [1]. In general, if the bindings

to a variable X in Φ are $C_1, \ldots, C_n$, then we transform Φ into a disjunction of formulas $\Phi_{[X \to C_i]}$ as follows.

$$\exists X \Phi \equiv \Phi_{[X \to C_1]} \vee \cdots \vee \Phi_{[X \to C_n]} \tag{2.2}$$

In this case, no new query answers are introduced, but the quantifier elimination results in a corresponding disjunction in the lineage formula.

Example 2 *Given the query $Won(X, BestWriting)$ over the base tuples and views depicted in Figure 1.1, we see that the head literals of both ν_4 and ν_5 unify with this query literal. Applying Rule (1) to the query literal, and subsequently Rule (2) results in:*

$$\begin{array}{c}(\nu_5 \wedge Nominated(X, BestWriting)) \vee \\ (\nu_4 \wedge Nominated(X, BestWriting) \wedge \neg Category(X, Action))\end{array}$$

By applying Rule (1) on the first instance of $Nominated(X, BestWriting)$, tuples t_9 and t_{11} unify with this subgoal and bind the query variable X to Inception and PulpFiction, respectively. Hence we obtain the two distinct answers $Won(Inception, BestWriting)$ and $Won(PulpFiction, BestWriting)$, where the former's lineage is

$$\left((\nu_5 \wedge t_9) \vee \begin{pmatrix} Nominated(Inception, BestWriting) \\ \wedge \neg Category(Inception, Action) \wedge \nu_4 \end{pmatrix}\right)$$

and the latter's lineage is:

$$\left((\nu_5 \wedge t_{11}) \vee \begin{pmatrix} Nominated(PulpFiction, BestWriting) \\ \wedge \neg Category(PulpFiction, Action) \wedge \nu_4 \end{pmatrix}\right)$$

Applying Rule (1) on the $Nominated$ literals of both lineage formulas yields $(\nu_5 \wedge t_9) \vee (\nu_4 \wedge t_9 \wedge \neg Category(Inception, Action))$ and $(\nu_5 \wedge t_{11}) \vee (\nu_4 \wedge t_{11} \wedge \neg Category(PulpFiction, Action))$. Finally, with respect to the $Category$ literals, Rule (1) delivers t_{12} and $false$ for the first and second answer, respectively. Thus, the final (propositional) lineage formulas of the two answers $Won(Inception, BestWriting)$ and $Won(PulpFiction, BestWriting)$ are $(\nu_5 \wedge t_9) \vee (\nu_4 \wedge t_9 \wedge \neg t_{12})$ and $(\nu_5 \wedge t_{11}) \vee (\nu_4 \wedge t_{11} \wedge \neg false)$, respectively. $\diamond$

Notice that all our operations are by default duplicate-eliminating. That is, in the case when a same literal can be answered by multiple rules (or a combination of rules and base tuples), we create a corresponding disjunction in the Boolean lineage formula of the derived tuple.

Complete Lineage. In a non-probabilistic Datalog setting, it is sufficient to find a single proof for an answer in order to show that the answer exists. In contrast, for

Datalog rules over probabilistic data, *all* such proofs over the given rules and base tuples are required in order to correctly capture all the possible worlds (and only those) for which a query answer exists, i.e., for which the propositional lineage formulas of the answer evaluates to $true$. Omitting a single proof might thus entail incorrect marginal probabilities. SLD resolution yields this "all-proofs" semantics [19].

2.6 Confidence Computations

A propositional lineage formula can be evaluated over a possible world $\mathcal{W} \subseteq (\mathcal{T} \cup \mathcal{V})$ by setting all variables in $\mathcal{W}$ to $true$ and all variables in $(\mathcal{T} \cup \mathcal{V}) \backslash \mathcal{W}$ to $false$, respectively.

Computing Marginals. For a propositional lineage formula ϕ, let $\mathcal{M}(\phi)$ be the set of possible worlds (aka. "models") satisfying ϕ. Then, the marginal probability $P(\phi)$ of a derived tuple (represented by its propositional lineage formula ϕ) is defined as the sum of the probabilities of all worlds for which ϕ evaluates to $true$.

$$P(\phi) := \sum_{\mathcal{W} \in \mathcal{M}(\phi)} P(\mathcal{W}) \tag{2.3}$$

We note that the above sum may range over exponentially many terms because there are exponentially many possible worlds. In fact, computing $P(\phi)$ is known to be $\#\mathcal{P}$-hard for general propositional formulas [7, 28].

Shannon Expansions. Alternatively, to avoid computing the sum of Equation (2.3), in a tuple-independent probabilistic database setting we can compute marginals by incrementally decomposing the propositional lineage formulas into variable-disjoint subformulas [10, 23]. Generally, for two propositional formulas ϕ, ψ over disjoint sets of independent random variables, the following relationships hold:

$$P(\phi \wedge \psi) := P(\phi) \cdot P(\psi) \tag{2.4}$$

$$P(\phi \vee \psi) := 1 - (1 - P(\phi)) \cdot (1 - P(\psi)) \tag{2.5}$$

$$P(\neg\phi) := 1 - P(\phi) \tag{2.6}$$

If the above principles are not directly applicable to two propositional formulas ϕ and ψ due to a shared variable t, this variable can be eliminated by a *Shannon expansion*. This is based on the equivalence

$$\phi \equiv (t \wedge \phi_{[t \to true]}) \vee (\neg t \wedge \phi_{[t \to false]})$$

where $\phi_{[t \to true]}$ denotes the restriction of ϕ to the case when t is $true$, i.e., all occurrences of t in $\phi_{[t \to true]}$ are substituted by the constant $true$. Then, it holds

that:

$$P(\phi) = P(t)\ P(\phi_{[t \to true]}) + (1 - P(t))\ P(\phi_{[t \to false]}) \tag{2.7}$$

Repeated Shannon expansions can increase the size of a formula exponentially. This issue can be addressed to some extent by incremental decompositions as shown in [23].

2.7 Expressiveness

With the above form of grounding, we encode the Boolean random variables that represent the uncertain views used for grounding a query answer directly into the lineage formulas of that query answer. Deriving a tuple via a view thus denotes the probabilistic event that "the input tuples to the view hold *and* the view itself holds", which is captured by a conjunction of the random variables for the tuples that ground the rule with the random variable associated with the view.

We remark that annotating views with probabilities does not exceed the expressiveness of a tuple-independent probabilistic database model, i.e., our model still falls within the class of uncertain databases with lineage [4]. The reason is that we can equivalently introduce a new relation $Rules$ which stores one tuple r per view ν that holds the respective probability $p(\nu)$. For example, we can rewrite ν_4 in Figure 1.1 to

$$\forall X, Y\ \ Won(X, Y) \text{ :- } Nominated(X, Y), Rules(r_4)$$

where we add the tuple r_4 to the relation ***Rules*** with confidence 0.4. In the following, we argue that, while uncertain views yield an intuitive semantics, they may also greatly contribute to an early pruning of answers derived from low-confidence views (which we evaluate empirically in Section 7).

3 Confidence Bounds

In this section, we develop lower and upper bounds for the marginal probability of any query answer that can be obtained from grounding a first-order lineage formula. We will proceed by constructing two propositional lineage formulas ϕ_{low} and ϕ_{up} from a given first-order lineage formula Φ. Then, the confidences of ϕ_{low} and ϕ_{up} will serve as lower and upper bounds on the confidences of all query answers captured by Φ. More formally, if $\phi_1, \ldots, \phi_n$ represent all query answers we would obtain by fully grounding Φ, then it holds that:

$$\forall i \in \{1, \ldots, n\} : \; P(\phi_{low}) \leq P(\phi_i) \leq P(\phi_{up})$$

Building upon results of [10, 23, 29], we develop two theorems, which (1) provide a mechanism for obtaining lower and upper bounds for formulas with first-order literals, and which (2) guarantee that these bounds converge monotonically to the marginal probabilities $P(\phi_i)$ of each query answer ϕ_i as we continue to ground Φ in a top-down manner.

3.1 Bounds for Propositional Lineage

As a first step, we relate the confidence of two propositional lineage formulas ϕ and ψ via their sets of models $\mathcal{M}(\phi)$ and $\mathcal{M}(\psi)$, i.e., the sets of possible worlds over which ϕ and ψ evaluate to *true*, respectively.

Proposition 1 *For two propositional lineage formulas ϕ and ψ, it holds that:*

$$\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi) \;\; \Rightarrow \;\; P(\phi) \leq P(\psi) \; [23]$$

That is, $\mathcal{M}(\phi)$ includes all possible worlds for which ϕ evaluates to *true*. Since we assume $\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi)$, the same worlds satisfy ψ as well. However, there might be more worlds fulfilling ψ but not ϕ. This might yield more terms over which the sum of Equation (2.3) ranges, and thus we obtain $P(\phi) \leq P(\psi)$.

Example 3 *Consider the two formulas $\phi \equiv t_1$ and $\psi \equiv t_1 \vee t_2$. From $\mathcal{M}(t_1) \subset \mathcal{M}(t_1 \vee t_2)$ it follows that $P(t_1) < P(t_1 \vee t_2)$, which we can easily verify using Equation* (2.3). $\diamond$

Conjunctive Clauses. To turn Proposition 1 into upper and lower bounds, we proceed by considering *clauses* in the form of conjunctions of propositional literals, where $Literals(\phi)$ denotes the set of literals contained in a clause.

Proposition 2 *Let ϕ, ψ be two propositional, conjunctive clauses. It holds, that $\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi)$ if and only if $Literals(\phi) \supseteq Literals(\psi)$* [23].

The above statement expresses that adding literals to a conjunction ϕ removes satisfying worlds from $\mathcal{M}(\phi)$.

Example 4 *For the clauses $t_1 \wedge t_2$ and t_1, it holds that $Literals(t_1 \wedge t_2) \supset Literals(t_1)$ and $\mathcal{M}(t_1 \wedge t_2) \subseteq \mathcal{M}(t_1)$.* $\diamond$

Disjunctive Normal Form. Moreover, we say that a propositional formula ϕ is in *disjunctive normal form* (DNF), if it is a disjunction of conjunctive clauses.

Lemma 1 *For two propositional DNF formulas ϕ and ψ, it holds that*

$$\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi) \;\; \Leftrightarrow \;\; \forall \phi' \in \phi \; \exists \psi' \in \psi : \mathcal{M}(\phi') \subseteq \mathcal{M}(\psi')$$

where ϕ' and ψ' are conjunctive clauses [23, 29].

The lemma establishes a relationship between two formulas in DNF. If we can map all clauses ϕ' of a formula ϕ to a clause ψ' of ψ with more satisfying worlds, i.e., $\mathcal{M}(\phi') \subseteq \mathcal{M}(\psi')$, then ψ has more satisfying possible worlds than ϕ. The mapping of clauses is established via Proposition 2.

Example 5 *For the DNF formula $\phi \equiv (t_1 \wedge t_2) \vee (t_1 \wedge t_3) \vee t_4$, we can map all clauses in ϕ to a clause in $\psi \equiv t_1 \vee t_4$. Hence, ψ has more models than ϕ, i.e., $\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi)$.* $\diamond$

Thus, Lemma 1 together with Proposition 1 enables us to compare the marginal probabilities of propositional formulas in DNF based on their clause structure.

Converting Formulas to DNF. Any propositional formula can equivalently be transformed into DNF by iteratively applying De Morgan's law, and thereafter the distributive law.

Observation 1 *If a variable t occurs exactly once in a propositional formula ϕ, then all occurrences of t in the DNF of ϕ have the same sign.*

The reason is that the sign of a variable t changes only when using De Morgan's law. However, when applying De Morgan's law, no variables are duplicated. When utilizing the distributive law, on the other hand, variables are duplicated but preserve their signs.

3.2 Bounds for First-Order Lineage

Analogously to the DNF for propositional formulas, any first-order formula can equivalently be transformed into prenex normal form by pulling all quantifiers in front of the formula. The remaining formula can again be transformed into DNF, which is then called prenex-DNF (PDNF). For our following constructions on first-order formulas, we will assume the first-order formulas to be given in PDNF. In general, such a normalization may lead to an exponential increase of the size of the formula. However, this construction serves as a theoretical tool, only, and never actually needs to be performed by the algorithms described in Section 5.

Let us assume we are given a first-order lineage formula Φ_R which is in propositional form except for one subgoal $R^\gamma(\bar{X})$. Moreover, we require the grounding of $R^\gamma(\bar{X})$ (see Section 2.4) to yield only propositional terms, i.e., Boolean variables referring to base tuples. Hence, we refer to $\phi_1, \ldots, \phi_n$ as the propositional formulas, which we obtain by grounding $R^\gamma(\bar{X})$ in Φ_R. The following theorem provides bounds on each $P(\phi_i)$ by means of Φ_R.

Theorem 1 *Given a first-order lineage formula Φ_R, which is in propositional form except for one subgoal $R^\gamma(\bar{X})$, and propositional lineage formulas $\phi_1, \ldots, \phi_n$, which are obtained from Φ_R by grounding $R^\gamma(\bar{X})$. We construct ϕ_{up} by*

- *substituting $R^\gamma(\bar{X})$ with $true$ if $R^\gamma(\bar{X})$ occurs positive in the PDNF of Φ_R, or*
- *substituting $R^\gamma(\bar{X})$ with $false$ if $R^\gamma(\bar{X})$ occurs negated in the PDNF of Φ_R.*

Analogously, we construct ϕ_{low} by

- *substituting $R^\gamma(\bar{X})$ with $false$ if $R^\gamma(\bar{X})$ occurs positive in the PDNF of Φ_R, or*
- *substituting $R^\gamma(\bar{X})$ with $true$ if $R^\gamma(\bar{X})$ occurs negated in the PDNF of Φ_R.*

Then it holds that:

$$\forall i \in \{1, \ldots, n\} : \; P(\phi_{low}) \leq P(\phi_i) \leq P(\phi_{up})$$

Proof 1 *Choose an arbitrary but fixed $i \in \{1, \ldots, n\}$. W.l.o.g., we assume Φ_R and ϕ_i to be in PDNF. The PDNF of Φ_R may consist of one or more clauses that contain $R^\gamma(\bar{X})$, which are either of the form $(\psi \wedge R^\gamma(\bar{X}))$ or $(\psi \wedge \neg R^\gamma(\bar{X}))$. For each of these clauses, ϕ_i may contain a number (due to Equation* (2.2)*) of clauses of the form $(\psi \wedge \varphi)$. Here, the literals φ correspond to groundings of $R^\gamma(\bar{X})$.*

When substituting $R^\gamma(\bar{X})$ by $true$ or $false$ as stated in Theorem 1, Φ_R's clauses turn into $(\psi \wedge true)$ and $(\psi \wedge false)$ for the upper and lower bounds, respectively. Subsequently, considering the upper bound, we employ Proposition 2 which yields $\mathcal{M}(\psi \wedge \varphi) \subseteq \mathcal{M}(\psi \wedge true)$, since $Literals(\psi \wedge \varphi) \supseteq Literals(\psi \wedge true)$. Next,

from Lemma 1 it follows that $\mathcal{M}(\phi_i) \subseteq \mathcal{M}(\phi_{up})$. This matches exactly the precondition of Proposition 1 from which we obtain $P(\phi_i) \leq P(\phi_{up})$. The case for the lower bound $P(\phi_{low})$ follows analogously.

Example 6 *For the first-order lineage formula*

$$\Phi_R \equiv t_{14} \wedge \exists X \, actedIn(Travolta, X)$$

the upper bound is given by $P(t_{14} \wedge true) = P(t_{14}) := p(t_{14})$, and the lower bound is $P(t_{14} \wedge false) = P(false) = 0$. For any set of tuples $t_1, \ldots, t_n$ matching $\exists X \; actedIn(Travolta, X)$, we have $\phi \equiv (t_{14} \wedge (t_1 \vee \cdots \vee t_n))$ with $0 \leq P(t_{14} \wedge (t_1 \vee \cdots \vee t_n)) \leq P(t_{14})$. $\diamond$

Since $R^\gamma(\bar{X})$ has exactly one occurrence in Φ_R, all occurrences of $R^\gamma(\bar{X})$ in the PDNF of Φ_R have the same sign (see Observation 1). Therefore, we replace all occurrences of $R^\gamma(\bar{X})$ in the PDNF of Φ_R by either $true$ or $false$.

Moreover, for a general first-order lineage formula Φ, which contains multiple distinct subgoals, we can apply the substitution provided by Theorem 1 multiple times to obtain bounds. That is, we replace every occurrence of a subgoal $R^\gamma(\bar{X})$ in Φ by one application of Theorem 1's substitution to obtain ϕ_{low} and ϕ_{up} of Φ.

Our last step is to show that, for a fixed query answer ϕ (see Section 2.5), the confidence bounds converge monotonically to the marginal probability of the propositional lineage formula $P(\phi)$ with each SLD step. The argument follows from the execution of the grounding procedure, but backwards. By Φ_1 we denote the original query, and Φ_n corresponds to the lineage formula before the last grounding step from which we obtain the final propositional formula ϕ.

Theorem 2 *Let $\Phi_1, \ldots, \Phi_n$ denote a series of first-order formulas obtained from iteratively grounding a conjunctive query via SLD resolution. Then, rewriting each Φ_i to $\phi_{i,low}$ and $\phi_{i,up}$ according to Theorem 1 creates a monotonic series of lower and upper bounds $P(\phi_{i,low})$, $P(\phi_{i,up})$ for the marginal probability $P(\phi)$. That is:*

$$\begin{aligned} 0 \leq P(\phi_{1,low}) \leq \cdots \leq P(\phi_{n,low}) \leq P(\phi) \\ \leq P(\phi_{n,up}) \leq \cdots \leq P(\phi_{1,up}) \leq 1 \end{aligned}$$

Proof 2 *We prove the theorem by induction, where i denotes the number of grounding steps taken.*
Basis $i = n$: $P(\phi_{n,low}) \leq P(\phi) \leq P(\phi_{n,up})$ is covered by one application of Theorem 1. That is, we have exactly one occurrence of a subgoal $R^\gamma(\bar{X})$ in Φ_n, which we replace with either $true$ or $false$ to obtain $\phi_{n,low}$ and $\phi_{n,up}$, respectively, such that $P(\phi_{n,low}) \leq P(\phi)$ and $P(\phi_{n,up}) \geq P(\phi)$.
Step $i \to i-1$: By the hypothesis, we are given a formula Φ_i with bounds characterized by $P(\phi_{i,up})$ and $P(\phi_{i,low})$.

Let us consider the grounding step which led to Φ_i. In Φ_{i-1} a subgoal $R^\gamma(\bar{X})$ must have been processed from which we obtained Φ_i. Let $\Phi_{i-1} \equiv \Psi\ R^\gamma(\bar{X})\ \Psi'$, where Ψ, Ψ' are a prefix and suffix of Φ_{i-1}. One step of grounding $R^\gamma(\bar{X})$ via SLD resolution thus leads to the formula $\Phi_i \equiv \Psi\ \Phi'_i\ \Psi'$, where Φ'_i is a first-order formula that consists of one or more subgoals or ground terms (including the constants $true$ and $false$). If we replace every occurrence of $R^\gamma(\bar{X})$ in the conjunctive clauses of the PDNF of Φ_{i-1} by Φ'_i, we obtain a formula that is equivalent to Φ_i (but which is not necessarily in PDNF), and whose clauses contain more terms than the clauses in Φ_{i-1} and thus are more specific. From this, it follows that applying Theorem 1 on all subgoals in the actual PDNF's of Φ_i and Φ_{i-1} yields propositional formulas $\phi_{i,up}$ and $\phi_{i-1,up}$, such that $\mathcal{M}(\phi_{i,up}) \subseteq \mathcal{M}(\phi_{i-1,up})$. That is, $P(\phi_{i,up}) \leq P(\phi_{i-1,up})$. The case for the lower bounds $P(\phi_{i,low})$ follows analogously.

4 Subgoal Scheduling

In this section, we present our scheduling techniques for determining the benefit of exploring a particular subgoal. A major difference of Datalog (with recursion) to a traditional database setting is the lack of a static query plan. Instead, we aim to dynamically and adaptively determine the best join order among the literals in a rule's body by prioritizing subgoals with a *low selectivity*. Additionally, given our probabilistic setting, we also aim to ground those subgoals first which have a *high impact* on the confidence bounds of the answers.

4.1 Selectivity Estimation

Generally, selectivity estimation aims at computing how many answers are expected when a subgoal is expanded. For our setting, we employ a simple probabilistic model for intensional relations defined over both the view structure and the extensional data, with independence assumptions for joins and unions. We will express this by a function $S : \Phi \rightarrow [0, 1]$ which, when given a first-order lineage formula Φ, calculates a likelihood in the range of $[0, 1]$ for this lineage formula to return an answer over the set of base tuples and views. For ease of readability, we omit the arguments of subgoals with formulas for the remainder of this section.

We recursively define the *selectivity* $S(R^\gamma)$ of a subgoal R^γ with binding pattern γ and Boolean connectives $\wedge$, $\vee$ and $\neg$ as follows

$$
\begin{aligned}
S(R^\gamma, d) &:= s_R^\gamma \text{ if } R \text{ is extensional} \\
S(\neg R^\gamma, d) &:= 1 - S(R^\gamma, d) \\
S(\bigwedge_{i=1}^{n} R_i^\gamma, d) &:= \prod_{i=1}^{n} S(R_i^\gamma, d) \\
S(\bigvee_{i=1}^{n} R_i^\gamma, d) &:= \begin{cases} 1 - \prod_{i=1}^{n} (1 - S(R_i^\gamma, d+1)) & \bullet\ d < D \\ c & \bullet\ \text{else} \end{cases}
\end{aligned}
$$

where s_R^γ denotes the selectivity of an extensional relation R given the binding pattern γ, and c is a constant selectivity factor which serves as a cut-off point for the selectivity estimation at a given (top-down) grounding depth D.

This cut-off depth D serves to break the selectivity estimation for highly convoluted views and thus also allows us to handle recursive views (see Subsection 6.2). Our rationale for employing this heuristic is that we usually obtain enough information for *ordering subgoals* by analyzing the structure of the views and the selectivities of the extensional relations by expanding the views up to a reasonable nesting depth. Hence, we mostly focus on finding good estimates for the relative selectivity (i.e., the best order) of subgoals to be scheduled, while exact selectivity values are not necessarily required. The advantage of this approach is that these selectivity estimates can be fully precomputed over views $\mathcal{V}$ and extensional tuples $\mathcal{T}$, and for any relation R with binding pattern γ. At query time, we simply pick the selectivity estimates for all subgoals to be scheduled from a precomputed table and use $S(R_i^\gamma, 0)$, i.e., always the estimate starting at level 0 (regardless of the current state of the grounding procedure), to determine the scheduling order of subgoals. For an extensional relation R, we use the average amount of tuples matching a given relation and binding pattern over the size of the extensional database as an estimate of s_R^γ. However, our approach would also allow for precomputing more precise statistics for predicates with individual constants as binding patterns, or to incorporate explicit correlations among predicates in the form of joint selectivities in case this information is available. Our experiments show that $c = 0.001$ and a maximum nesting depth of $D = 5$ yield good estimates for this form of subgoal scheduling.

Example 7 *Given the query* $Won(X, BestWriting)$ *and views* ν_4 *and* ν_5*, selectivity estimation proceeds as follows*

$$
\begin{aligned}
& S(\mathit{Won}^\gamma, 0) \\
&= S\left((\mathit{Nominated}^{\gamma_1} \wedge \neg \mathit{Category}^{\gamma_2}) \vee \mathit{Nominated}^{\gamma_3}, 0\right) \\
&= 1 - \begin{pmatrix} (1 - S(\mathit{Nominated}^{\gamma_1} \wedge \neg \mathit{Category}^{\gamma_2}, 1)) \\ \cdot\, (1 - S(\mathit{Nominated}^{\gamma_3}, 1)) \end{pmatrix} \\
&= 1 - \begin{pmatrix} (1 - S(\mathit{Nominated}^{\gamma_1}, 1) \cdot\ S(\neg \mathit{Category}^{\gamma_2}, 1)) \\ \cdot\, (1 - S(\mathit{Nominated}^{\gamma_3}, 1)) \end{pmatrix} \\
&= 1 - (1 - \tfrac{2}{14} \cdot (1 - \tfrac{1}{14})) \cdot (1 - \tfrac{2}{14})
\end{aligned}
$$

where $\frac{2}{14}$ *results from the fact that* $Nominated$ *contains* 2 *tuples over a total of* 14 *in the database when the second argument is bound by* $BestWriting$*. Instead,* $Category$ *returns just* 1 *tuple (again over a total of* 14 *in the database) when binding the second argument to* $Action$*.* $\diamond$

4.2 Impact of Subgoals

In the next step, we aim to quantify the impact of the confidence $p(t)$ of a Boolean variable t on the marginal probability $P(\phi)$ of the propositional formula ϕ it occurs in. Later, the scheduler will exploit this to choose the subgoal with the highest impact on the confidence bounds of Φ. Following results from [17, 25], this impact measure can be captured by the following derivative.

$$\frac{\partial P(\phi)}{\partial p(t)} = \frac{P(\phi_{[t \to true]}) - P(\phi_{[t \to false]})}{1 - 0}$$

Lemma 2 *For a propositional formula ϕ, if we fix the confidences of all variables except t, it holds that*

$$P(\Phi) = c\ p(t) + c'$$

where c and c' are two constants independent of $p(t)$.

Proof 3 *One step of Shannon Expansion on t results in:*

$$\begin{aligned} \phi &\equiv (t \wedge \phi_{[t \to true]}) \vee (\neg t \wedge \phi_{[t \to false]}) \\ &\Rightarrow \\ P(\phi) &= p(t)\ P(\phi_{[t \to true]}) + (1 - p(t))\ P(\phi_{[t \to false]}) \\ &= p(t) \underbrace{(P(\phi_{[t \to true]}) - P(\phi_{[t \to false]}))}_{c} + \underbrace{P(\phi_{[t \to false]})}_{c'} \end{aligned}$$

Thus, to compute the above derivative, it suffices to compute c. A general first-order lineage formula Φ, however, may contain subgoals which makes the above sensitivity analysis not directly applicable to Φ. Again, by substituting all subgoals in Φ according to Theorem 1, we can quantify the impact of a subgoal $R^\gamma(\bar{X})$ on both the upper and lower bounds $P(\phi_{low})$, $P(\phi_{up})$ in the corresponding propositional formulas ϕ_{low}, ϕ_{up}. That is, to quantify the impact of $R^\gamma(\bar{X})$ on the upper bound, we substitute all other subgoals to obtain ϕ_{up} and then compute c by substituting $R^\gamma(\bar{X})$ once by the constant $true$ and once by $false$.

Example 8 *Consider the lineage formula*

$$\begin{aligned} &(t_4 \wedge \exists X\, Written(X, PulpFiction)) \\ \vee\ &(t_7 \wedge \exists X\, Category(PulpFiction, X)) \end{aligned}$$

containing the subgoals $Written(X, PulpFiction)$ *and* $Category(PulpFiction,X)$*. The impact on the formula's upper bound by the latter subgoal is then calculated as:*

$$\begin{aligned}
&(1-(1-p(t_4)\cdot P(true))(1-p(t_7)\cdot P(true)))\\
&\quad -(1-(1-p(t_4)\cdot P(true))(1-p(t_7)\cdot P(false)))\\
&=1-(1-0.6\cdot 1)(1-0.8\cdot 1)-(1-(1-0.6\cdot 1)(1-0.2\cdot 0))\\
&=0.92-0.6
\end{aligned}$$

4.3 Benefit-oriented Subgoal Scheduling

We now define the *benefit* of scheduling a subgoal for the next grounding step as

$$ben(\Phi, R^\gamma) := \frac{|im_{up}(\Phi, R^\gamma)| + |im_{low}(\Phi, R^\gamma)|}{1 + S(R^\gamma, d)} \tag{4.1}$$

where $im_{up}(\Phi, R^\gamma)$ describes the impact of R^γ on the upper bound of Φ. Hence, we favor subgoals with high impact and low selectivity.

5 Top-k Algorithm

Our top-k algorithm for computing query answers primarily operates on the lineage formulas of (sets of) answer candidates. Specifically, subgoals from all answer candidates, are kept in a priority queue Q that is ordered according to the benefit-oriented subgoal scheduler (Section 4). Additionally, we maintain two disjoint sets of answer candidates A_{top} and A_{cand}. Following the seminal line of threshold algorithms [9], the former comprises the current top-k answers with respect to the lower bounds we compute for their marginal probabilities. The latter consists of all remaining answer candidates whose upper confidence bounds are still higher than the worst lower bound of any of the top-k answers. As an additional constraint, the top-k set A_{top} consists only of query answers whose lower bound is greater than 0. This coincides with answers for which all query variables have already been bound to constants by the grounding procedure, i.e., those for which we have at least one proof but not yet necessarily all proofs. The candidate set, on the other hand, may hold answers candidates with a lower confidence bound of 0, i.e., also those for which the query variables are not yet bound to constants, hence representing sets of answers.

Each grounding step is performed by Algorithm 2, which is based on SLD resolution [1] that has been extended by an incremental form of lineage tracing.

5.1 Top-k with Dynamic Subgoal Scheduling

At each processing step, the scheduler chooses the currently best subgoal $R^{\gamma}_{best}(\bar{X})$ from the subgoal queue Q (Line 9), and we expand the lineage formula of this subgoal by performing a single SLD step over both $\mathcal{V}$ and $\mathcal{T}$ (Line 10) as described in Section 2.5. Then, (Line 11) we update A_{top} and A_{cand} due to following three reasons. First, expanding $R^{\gamma}_{best}(\bar{X})$ can change the confidence bounds of the answer. Second, if there are no matches to $R^{\gamma}_{best}(\bar{X})$, neither in $\mathcal{T}$ nor in $\mathcal{V}$, the answer candidates corresponding to $R^{\gamma}_{best}(\bar{X})$ may be deleted (thus its lineage evaluates to $false$). And third, if a query variable was bound to more than one constant, one

or more new top-k answer candidates are created.

Just like the original line of top-k algorithms [9], we keep track of answer candidates residing in A_{top} and A_{cand} until the following threshold-based stopping condition holds:

$$\forall\ \Phi_{top} \in A_{top}, \Phi_{cand} \in A_{cand} : P(\phi_{cand,up}) \leq P(\phi_{top,low}) \tag{5.1}$$

Furthermore, an answer candidate $\Phi_{cand} \in A_{cand}$ may be dropped from the candidate set A_{cand} if:

$$\forall\ \Phi_{top} \in A_{top} : P(\phi_{cand,up}) \leq P(\phi_{top,low}) \tag{5.2}$$

To keep Q consistent with A_{top} and A_{cand}, we update Q (Line 11). First, all subgoals occurring in deleted answers are dropped from Q. Then, all newly created subgoals (due to new answers or the application of rules) are added to Q. Finally, the impact (see Section 4.2) of all subgoals appearing in the same lineage formula as $R^{\gamma}_{best}(\bar{X})$ might have changed. Hence their priority in Q is updated.

Finally, Algorithm 1 terminates when the threshold-based breaking condition in Line 7 of the algorithm holds, or when the candidate set A_{cand} runs out of valid answer candidates.

Algorithm 1 Top-$k(\mathcal{V}, \mathcal{T}, \Phi_q, k)$

Input: Views $\mathcal{V}$, uncertain tuples $\mathcal{T}$, an intensional query Φ_q, and an integer value k
Output: Top-k answers A_{top} for Φ_q according to their lower confidence bounds

1: Initialize global priority queue Q with subgoals from Φ_q
2: $A_{top} := \emptyset$ ▷ Current top-k answers
3: $A_{cand} := \{\Phi_q\}$ ▷ Answer candidates
4: **while** $A_{cand} \neq \emptyset$ **do**
5: min-$k := \min_{\Phi_i \in A_{top}}\{P(\phi_{i,low}), 0\}$ ▷ Thm. 1
6: max-cand $:= \max_{\Phi_i \in A_{cand}} P(\phi_{i,up})$ ▷ Thm. 1
7: **if** min-$k >$ max-cand **then**
8: **break**
9: $\left(\Phi_{best}, R^{\gamma}_{best}(\bar{X})\right) :=$ $\arg\max_{\left(\Phi_i \in A_{top} \cup A_{cand}, R_i(\bar{X}) \in Q\right)} ben(\Phi_i, R_i(\bar{X}))$ ▷ Eqn. 4.1
10: $\Phi :=$ SLD($\mathcal{V}, \mathcal{T}, \Phi_{best}, R^{\gamma}_{best}(\bar{X})$) ▷ Alg. 2
11: Update A_{top}, A_{cand}, Q using Φ ▷ Eqns. 5.1,5.2
12: **return** A_{top}

5.2 SLD Resolution with Lineage Tracing

Algorithm 2 covers a single SLD step and is called as a subroutine of Algorithm 1 presented in the previous subsection. During each SLD step, a subgoal $R^{\gamma}(\bar{X})$ is replaced by new subgoals obtained from grounding the rules that define $R^{\gamma}(\bar{X})$, such that an updated version of all answers' lineages that share $R^{\gamma}(\bar{X})$ is returned. The algorithms corresponds to Rule (1) and (2) introduced in Section 2.5.

Algorithm 2 SLD($\mathcal{V}, \mathcal{T}, \Phi, R^{\gamma}(\bar{X})$)

Input: Uncertain views $\mathcal{V}$, uncertain tuples $\mathcal{T}$, a first-order lineage formula Φ, a subgoal $R^{\gamma}(\bar{X})$ contained in Φ

Output: Updated lineage formula Φ

1: **if** R extensional **then**
2: M := $\{(t, \gamma_u) \mid t$ and γ unify to $\gamma_u\}$
3: **else**
4: $M := \left\{ (\nu, \gamma_u) \;\middle|\; \begin{array}{c} \nu = R'^{\gamma'}(\bar{X}) \text{:- } body \in \mathcal{V}, R = R', \\ \gamma \text{ and } \gamma' \text{ unify to } \gamma_u \end{array} \right\}$
5: **if** $M = \emptyset$ **then**
6: Replace $R^{\gamma}(\bar{X})$ in Φ by $false$ ▷ Rule (1)
7: **return** Φ
8: **for** $\gamma_u^* \in bindings(M)$ **do**
9: **if** γ_u^* binds new variables in γ **then**
10: $\Phi :=$ expand Φ utilizing Eqn. (2.2)
11: $L := R^{\gamma_u^*}(\bar{X})$ ▷ Created in previous step
12: **else**
13: $L := R^{\gamma}(\bar{X})$
14: **if** R is extensional **then**
15: Replace L by $\bigvee_{(t,\gamma_u) \in M, \gamma_u = \gamma_u^*} \mathcal{X}_t$ ▷ Rule (1)
16: **else**
17: $B := \{body \mid (R'^{\gamma'}(\bar{X}) \text{:- } body, \gamma_u) \in M, \gamma_u = \gamma_u^*\}$
18: **for** $body \in B$ **do**
19: Propagate γ_u^* to bindings in *body*'s literals
20: Replace L in Φ by $\bigvee_{body \in B} body$ ▷ Rules (1),(2)
21: **return** Φ

If R is extensional, we collect all matching tuples in M (Line 2). Otherwise, in Line 4 we gather all rules whose relation R' coincides with R and whose bindings γ' unify with γ. Thus, the set M holds a pair consisting of both the rule and the unified bindings γ_u. Now, if there are no matching rules or tuples, that is $M = \emptyset$, we replace the subgoal by $false$ and return the altered Φ in Line 7. The loop in Line 8 ranges over all different bindings γ_u^* obtained from unifying the subgoal's bindings γ with the bindings γ' occurring in the head of a matching rule. If γ_u^*

binds more variables than γ, then in Line 10 we instantiate the quantifiers that hold the newly bound variables according to Equation 2.2. Afterwards, in Line 11, the copy $R^{\gamma^*_u}(\bar{X})$ of $R^{\gamma}(\bar{X})$ is saved in L. If $R^{\gamma^*_u}$ matches tuples from $\mathcal{T}$, we replace L in Φ by a disjunction of the variables $\mathcal{X}_t$ in Line 15. Otherwise L is substituted by a disjunction over the bodies of all rules with head $R^{\gamma^*_u}$ (Line 20). For an illustration of the algorithm, we refer the reader to Example 2.

5.3 Final Result Ranking

Some applications may require a complete ranking of the top-k answers. When Algorithm 1 finishes, the marginal probabilities of the top-k answers might not be exactly known (but only their bounds). We can tackle this by either iteratively running top-1, ..., top-k queries, where an inspection of the answer sets yields the final ranking, or by continuing grounding and applying decompositions until the confidence bounds of the top-k answers do not overlap anymore.

6 Extensions

6.1 Sorted Input Relations

A powerful technique in top-k query processing is to store each relation in decreasing order of local ranks and to use the rank at the current scan position as an upper bound for the ranks of all remaining tuples. Top-k algorithms for extensional data [9, 15, 21] specifically focus on the former case with sorted input relations. In our setting, we rank by confidence values and an extensional relation R may contain a large amount of tuples that unify with a subgoal $R^\gamma(\bar{X})$. For example, when querying for the top-2 answers of $Directed(X, Y)$ in Figure 1.1, we only need to read t_2 and t_3 if the relation is sorted by $p(t_i)$. This strategy assumes that R contains no duplicate tuples, which we can however overcome by a preprocessing step applying an independent-project operation [36] to the relation, if necessary.

In Theorem 1, we replaced a positive subgoal $R^\gamma(\bar{X})$ by the constant $true$ to obtain this upper bound. This corresponds to using 1 as a conservative upper bound for $R^\gamma(\bar{X})$, since $P(true) = 1$. The following observation allows us to lower this upper bound for a subgoal over an extensional input relation that is sorted in decreasing order of ranks.

Observation 2 *Let R be an extensional relation with tuples sorted in decreasing order of $p(t_i)$, let $R^\gamma(\bar{X})$ be a subgoal and let $t_1, \ldots, t_m$ be the tuples matching $R^\gamma(\bar{X})$. Then, for subgoal $R^\gamma(\bar{X})$, we can set the upper confidence bound of each tuple $t_j \in R$, with $i < j \leq m$, to $\min\{p(t_1), \ldots, p(t_i)\}$, if and only if all unbound variables of $R^\gamma(\bar{X})$ are query variables and there are no duplicates in R.*

The key for this observation is that binding a query variable yields a new query answer, while binding existentially quantified variables introduced by a rule results in a disjunction in the lineage formula due to a quantifier elimination. This disjunction may result in a higher confidence than that of the individual input tuples due to Equation (2.5). For example, if two independent tuples t_1, t_2 with a

confidence of 0.5 each match a single subgoal with non-query variables, then we obtain $1-(1-0.5)\cdot(1-0.5)=0.75>0.5$. Thus, using an upper bound of 0.5 would be incorrect.

6.2 Recursive Rules

We develop an algorithmic extension for handling rules with recursively defined intensional relations. To ensure a safe semantics for the deductive grounding steps, we require the set of recursive rules $\mathcal{V}$ to be *stratifiable* [1], i.e., it is not allowed to deduce a tuple from its own negation. Stratifiability is a pure syntactic check on the rule structure and can be done prior to query processing. The combined complexity (in terms of data and rules) for Datalog programs with a single, recursive, non-linear rule is known to be EXPTIME-complete [13]. Although we cannot improve upon this worst-case bound, we argue that top-k pruning may also help to improve the runtime for recursive queries in practice.

Recursion poses a challenging problem for grounding. In principle, the lineage formula of an answer could grow infinitely large due to a cycle in the rules. Therefore, we next formulate a theorem which ensures the finiteness of a lineage formula Φ, without altering the possible worlds that satisfy Φ. We define a *cycle* to be a subgoal $R^{\gamma}(\bar{X})$ whose SLD expansion results in a formula Φ containing subgoals $R^{\gamma_1}(\bar{X}_1),\ldots,R^{\gamma_n}(\bar{X}_n)$, such that $(\gamma_1,\bar{X}_1),\ldots,(\gamma_n,\bar{X}_n)$ bind or contain the same constants as $(\gamma,\bar{X})$, but the names of the unbound variables in the $\bar{X}_i$'s may differ.

Theorem 3 *Let $\Phi=\Psi R^{\gamma}(\bar{X})\Psi'$ be a lineage formula and let the expansion $\Phi_{ex}(R^{\gamma}(\bar{X}))$ of $R^{\gamma}(\bar{X})$ yield the cycle $\Psi\Phi_{ex}(R^{\gamma}(\bar{X}))\Psi'$. Then it holds that:*

$$\Psi\Phi_{ex}(R^{\gamma}(\bar{X}))\Psi' \equiv \Psi\Phi_{ex}(\Phi_{ex}(R^{\gamma}(\bar{X})))\Psi'$$

In other words, expanding a cycle more than once does not change the validity of a lineage formula, which agrees with results from [27].

Proof 4 *W.l.o.g., we assume all formulas to be in prenex form. Furthermore, let $\Phi'\vee\bigvee_i(\Phi''_i\wedge R^{\gamma}(\bar{X}))$ be the DNF of $\Phi_{ex}(R^{\gamma}(\bar{X}))$. That is Φ' is a DNF formula, Φ_i are conjunctions of literals and both do not contain $R^{\gamma}(\bar{X})$. Due to stratification, $R^{\gamma}(\bar{X})$ must occur positively in the above formula. Now, we can rewrite $\Psi\Phi_{ex}(\Phi_{ex}(R^{\gamma}(\bar{X})))\Psi'$ through the following series of algebraic transformations:*

$$\begin{aligned}
&\Psi\Phi_{ex}(\Phi_{ex}(R^{\gamma}(\bar{X})))\Psi'\\
\equiv\ &\Psi\Phi'\vee\textstyle\bigvee_i(\Phi''_i\wedge(\Phi'\vee\bigvee_j(\Phi''_j\wedge R^{\gamma}(\bar{X}))))\Psi'\\
\equiv\ &\Psi\Phi'\vee\textstyle\bigvee_i(\Phi''_i\wedge\Phi)\vee\bigvee_{i,j}(\Phi''_i\wedge\Phi''_j\wedge R^{\gamma}(\bar{X}))\Psi'\\
\equiv\ &\Psi\Phi'\vee\textstyle\bigvee_{i,j}(\Phi''_i\wedge\Phi''_j\wedge R^{\gamma}(\bar{X}))\Psi'\\
\equiv\ &\Psi\Phi'\vee\textstyle\bigvee_i(\Phi''_i\wedge\Phi''_i\wedge R^{\gamma}(\bar{X}))\vee\bigvee_{i\neq j}(\Phi''_i\wedge\Phi''_j\wedge R^{\gamma}(\bar{X}))\Psi'
\end{aligned}$$

$$\equiv \Psi\Phi' \vee \bigvee_i(\Phi_i'' \wedge R^\gamma(\bar{X})) \vee \bigvee_{i\neq j}(\Phi_i'' \wedge \Phi_j'' \wedge R^\gamma(\bar{X}))\Psi'$$
$$\equiv \Psi\Phi' \vee \bigvee_i(\Psi_i'' \wedge R^\gamma(\bar{X}))$$
$$\equiv \Psi\Phi_{ex}(R^\gamma(\bar{X}))\Psi'$$

This yields again the form of the first expansion of $R^\gamma(\bar{X})$.

In our implementation of SLD resolution with dynamic subgoal scheduling, we block those subgoals in our priority queue that form the leaf of a cycle. If all subgoals in the lineage formula of an answer are blocked, no new results can be obtained, so we substitute these subgoals by $false$.

7 Experiments

In this section, we present our experimental evaluation. All experiments are performed on an 8-core Intel Xeon with 2.4 GHz and 48 GB of RAM. Our top-k approach is implemented in about 8k lines of Java code and utilizes a PostgreSQL database in the backend. We use two well-known PDB systems for comparison purposes, MayBMS[1] and Trio[2], both computing all answers and their respective probabilities. We also include measurements using a deterministic database, denoted as PostgreSQL, by storing confidence values in all relations but ignoring the actual confidence computations (which are combinatorially much more complex than a simple score aggregation). This serves as a lower bound on all top-k approaches (including [24, 25]), which require materialization and lineage tracing of all answer candidates. Additionally, we implemented [25], which we refer to as MultiSim.

7.1 Data Sets, Confidence Distributions, and Queries

We use two different datasets based on IMDB and YAGO. The IMDB movie dataset consists of 6 relations, namely *directed*, *acted*, *edited*, *produced*, *written*, and *hasCategory*, altogether summing up to $26 \cdot 10^6$ tuples. Since this data is deterministic, we sample confidence values from three distributions, namely *uniform*, *gaussian*, and *exponential*. Our second dataset is derived from the YAGO [35] knowledge base with $132 \cdot 10^6$ tuples, where we also sample confidences using a uniform distribution (YAGO's tuples come with a constant confidence value per relation). All rules and queries used in the experiments are available in Appendix A.

[1] MayBMS: `http://maybms.sourceforge.net/`
[2] Trio: `http://infolab.stanford.edu/trio/`

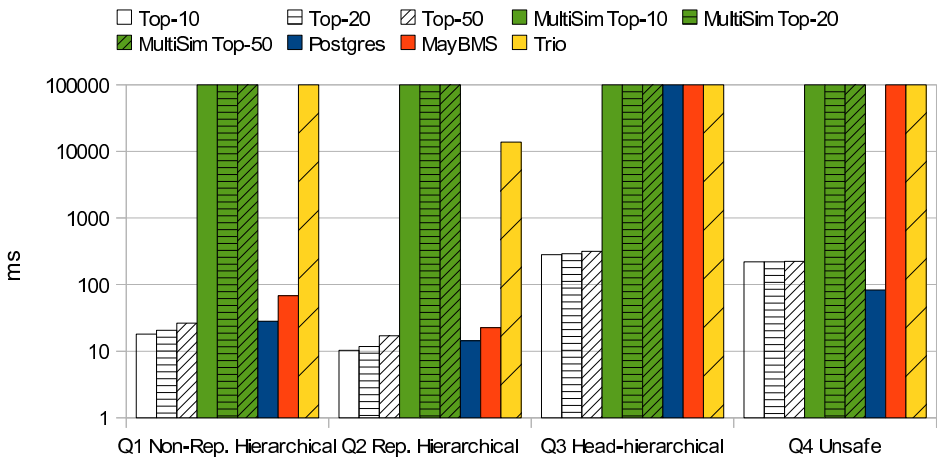


Figure 7.1: Query Classes

7.2 Results

Query Classes. We start by focusing on four established query classes [7, 24, 36] in the PDB field, namely *non-repeating hierarchical* queries, *repeating hierarchical* queries, *non-repeating head-hierarchical* queries, and *general unsafe queries* which are represented by query patterns $Q1$, $Q2$, $Q3$ and $Q4$, respectively. Out of the former three query classes, only the first class is guaranteed to yield safe query plans, while the second and third (as well as the fourth) are unsafe. Each of the four query patterns is instantiated to 1,000 different queries by inserting randomly chosen constants into the query literals, each ensuring at least 50 answers. In the following, we report average runtimes over the 1,000 individual queries per query class by running each query 4 times and taking the average runtime of the latter 3 runs to calculate these averages. That is, the following figures depict results from 16,000 individual queries for these first four query patterns $Q1$–$Q4$. For presentation purposes, we depict only up to an average runtime of 100 seconds for all systems.

The results for the IMDB dataset with uniform confidences are depicted in Figure 7.1. For the non-repeating hierarchical queries ($Q1$) our top-k approach outperforms all systems including the deterministic setting. We gain this advantage because not all answers' lineage needs to be materialized for determining the top-k answers. At the same time, the confidence computations required for computing the bounds are processed in polynomial time, not slowing down our top-k approach. $Q2$ contains repeated relations and the gains in data computations are partially diminished by the Shannon expansions performed for computing the bounds. Moreover, $Q3$ includes expensive data computations, which are caused by a subquery common to all answers and hence not required to rank the

results. So, our top-k algorithm successfully terminates and even outperforms the deterministic PostgreSQL baseline. Finally, $Q4$ is dominated by a subquery with $\#\mathcal{P}$-hard confidence computations. However, our top-k approach is able to prune answers before the expensive subquery is fully evaluated.

In all queries, Trio and MultiSim exhibit at least 100 times slower performance. For MultiSim, the runtime is spent in sampling (except for $Q3$). The runtime does not significantly increase with k, but rather depends on the distance of the k-th and $k+1$-th answers' confidences. The smaller the distance, the longer it takes to run MultiSim.

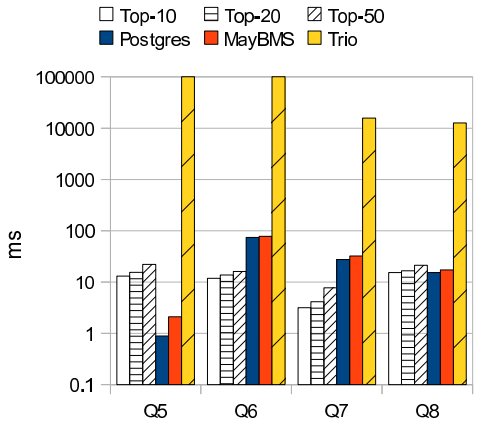


Figure 7.2: Performance Factors

Performance Factors. In this experiment, we specifically choose query patterns to highlight the features that affect the performance of our top-k approach against the competing systems. Figure 7.2 depicts runtimes on the IMDB dataset with uniform confidences for 4 additional query patterns $Q5$, $Q6$, $Q7$, and $Q8$, which are again instantiated to $1,000$ queries each. $Q5$ gives exactly one proof for each answer candidate. In this case, no pruning in terms of omitting grounding a proof is possible for our top-k approach. Also, the proof involves an existentially quantified variable, thus prohibiting the use of sorted input lists. As a result, MayBMS's bottom-up lineage computation of all answers is much more efficient. Continuing with $Q6$, the possibility of three proofs per answer allows for pruning. Also, the lack of existential quantifiers puts our approach in favor of the competitors. $Q7$ contains a join of two existential relations and can be considered as the main case where sorted input lists can yield a significant gain for our approach which outperforms all others. Finally, in $Q8$ each answer has up to three proofs, however their relations overlap requiring Shannon expansions for confidence computations. Since top-k repeatedly invokes these expensive confidence computations to determine the bounds, the advantages even out with the competitors.

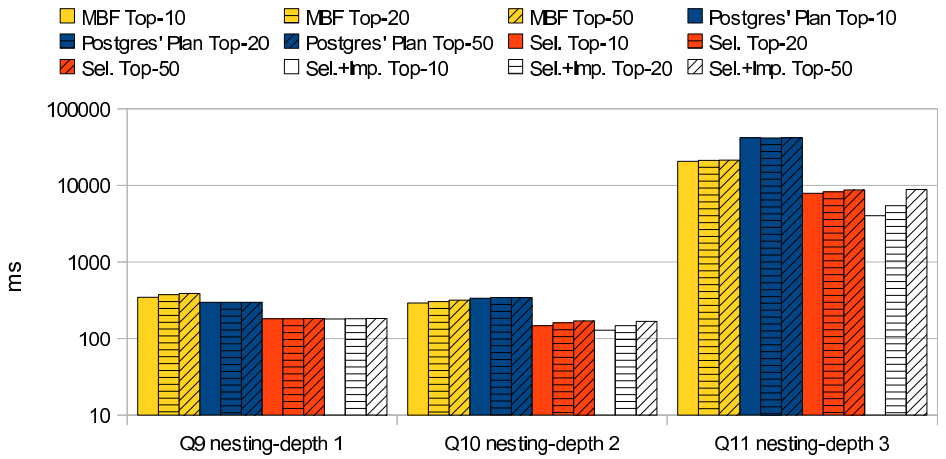
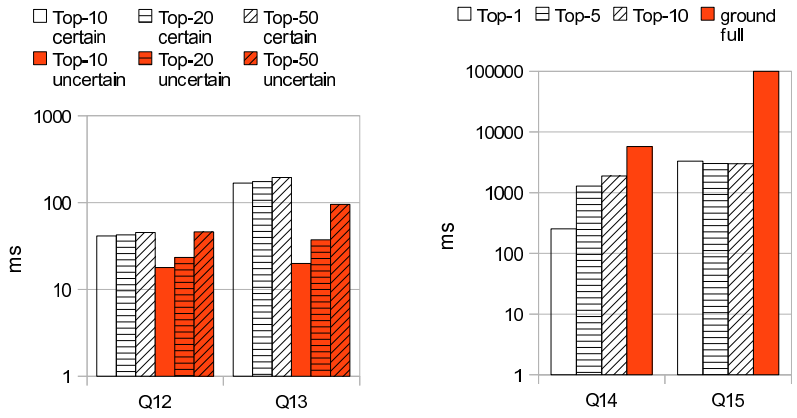


Figure 7.3: Scheduling

Scheduling. In the experiment of Figure 7.3, we empirically evaluate our scheduling techniques based on selectivity estimation (denoted as *Sel.*), presented in Section 4.1, and impact (denoted as *Imp.*), presented in Section 4.2, against two baselines. First, we implemented a dynamic subgoal scheduling strategy called "most-bound-first" (aka. "bound-is-easier" in [37]), which dynamically chooses the subgoal with the highest amount of arguments bound to constants at each SLD grounding step (denoted as *MBF*). Second, we obtained PostgreSQL's static query plan (see Appendix A) and forced our system to adhere to this plan (*Postgres' Plan*) for our combined top-k grounding and confidence computation strategy. Using the YAGO dataset, the three query patterns $Q9$, $Q10$, and $Q11$ were instantiated by 100 constants each. We arranged the query patterns by increasing the nesting depth of subqueries, such that $Q9$, $Q10$, and $Q11$ come with nesting depths of 1, 2, and 3, respectively.

For $Q9$, *MBF* is outperformed by both the Postgres plan and the scheduler that is guided by selectivity estimation. However, adding the impact calculations to the selectivity estimation does not yield any performance gains, but even results in slight losses. When moving to the higher nesting depths of $Q10$ and $Q11$, the impact calculations start improving the performance of the selectivity-based scheduler. This is due to the fact that without subqueries (in the case of $Q9$) the number of matching tuples to a subgoal dominates the runtime. In contrast, with higher nesting depth the subgoals that are closer to the query are more influential to determine the answers' confidence bounds, which is correctly discovered by the impact calculations. At the same time, higher nesting depths yield worse runtimes for the Postgres plan. The reason is that PostgreSQL's query optimizer is purely driven by selectivity estimation, thus neglecting the impact of a subgoal on an answer's confidence value.



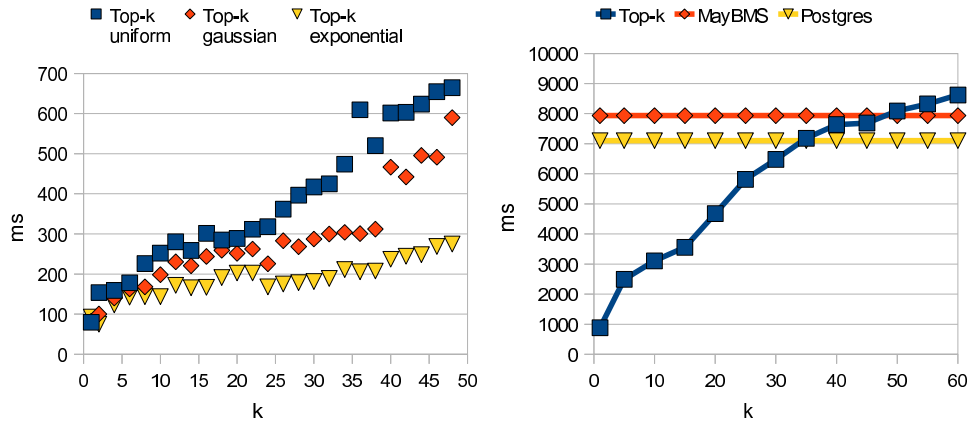
(a) Uncertain Views (b) Recursion

Figure 7.4: Experiments

Uncertain Views. In this experiment, we study how our approach can benefit from uncertain views. Figure 7.4(a) depicts the results of query patterns $Q12$ and $Q13$ over the IMDB dataset with uniform confidences. First, $Q12$ contains a union of three existential relations. In this case, uncertain views prioritize the extensional relations and downweight their answers such that less tuples have to be read from the database, which explains the performance gains. Second, $Q13$ allows for several proofs per answer, where the views' confidences decide on their impact. Therefore, proofs with lower confidences can be omitted speeding up the grounding process.

Recursion. Figure 7.4(b) depicts the performance of our top-k approach over the YAGO data set using two recursive query patterns, $Q14$ and $Q15$, instantiated to 50 queries each. We also include the runtime for a *full grounding* approach corresponding to an SLD grounding algorithm with lineage tracing, but without any confidence computations. $Q14$ computes ancestors of persons utilizing the *hasChild* relation, whereas $Q15$ asks for politicians of nations by recursively following the *hasSuccessor* relation. For $Q14$, the runtime increases with k, since more ancestors being generations away from the queried person have to be computed. However, for $Q15$ our top-k algorithm takes the same amount of time for different values of k, which is because more than 10 politicians are known per country and are ranked in the top-10 results. Also, when full grounding is used for $Q15$, the lineage computation of all answers becomes very expensive.

Confidence Distributions. In the previous experiments, we focused on a uniform distribution of the confidences. We now proceed by exploring how different distributions can affect our performance by comparing between uniformly, gaussian, and exponentially distributed confidences. Posing a join query on two existential relations over the IMBD dataset, the results for each confidence distribution



(a) Confidence Distributions (b) Table Materialization

Figure 7.5: Experiments

are depicted in Figure 7.5(a). As k grows, the runtimes increase for all distributions. Uniform seems to yield a linear increase. The exponential one exhibits the slowest grow, since there are very few tuples with high confidences. The gaussian shows a jump at $k = 40$, since more tuples with similar probabilities exist in this mode.

Table Materialization. Last, we compare our top-k approach against a full materialization of all answers performed both by MayBMS and PostgreSQL. We focus on how the value of k affects runtime. The query asks for directors of Comedies using the IMDB data set with uniform confidences (a join between *Directed* and *hasCategory* relations). Our top-k system computes the top-ranked answers for different values of k. In contrast, MayBMS and PostgreSQL fully materialize a table containing all results, where PostgreSQL ignores the confidence computation. When k is below 50, our top-k approach performs faster. For larger values, the bookkeeping overhead needed for the top-k algorithm starts dominating, such that it becomes more efficient to compute all answers by a full join.

8 Related Work

The increasing amount of uncertain data that has meanwhile become available practically at Web-scale has driven the development of various PDB engines in recent years, including systems like MystiQ [8], Trio [38], MayBMS [3], Orion [33], PrDB [31] and SPROUT [22]. Works on intensional query evaluation such as [11, 4, 30, 6, 14, 36] capture the lineage of derived tuples as propositional formulas and have been shown to be closed and complete under the relational model. To cope with the challenge of confidence computations, recent work has concentrated on exploiting *safe query plans* [8] and *read-once formulas* [32]. Dalvi and Suciu [7] define a dichotomy of query plans for which confidence computations can be done either in polynomial time or are $\#\mathcal{P}$-hard. As an alternative way of addressing confidence computations in PDBs, top-k style pruning approaches [25, 34, 12, 21, 24] have also been proposed. In relational database systems the most influential line of works for extensional data still is given by the family of threshold algorithms [9], often also simply referred to as "Fagin's algorithm". A comprehensive survey of top-k queries for relational DBs is presented by Ilyas et al. [15].

Most top-k approaches in the context of PDBs consider separate numerical attributes for capturing the confidence and the score of input tuples, where usually only the latter is used for ranking purposes. Soliman et al. [34] were the first to discuss the different semantics under which one can interpret uncertain top-k queries in such a setting and defined two types of queries, namely *U-topK* and *U-kRanks*. Recently, Ge et al. [12] studied the tradeoffs between reporting tuples of a high score and tuples of a high probability, while Li et al. [21] proposed a unified ranking approach by considering both the scores and confidences of tuples.

Very few works however consider top-k ranking by the marginal probabilities of query answers. Ré et al. [25] compute the top-k answers using MCMC-style sampling techniques. In this case, the correct ranking of the top-k answers is guaranteed while marginal probabilities are approximated only to the extent needed for computing the top-k answer set. To achieve this, the authors introduce the *multisimulation* algorithm which runs several Monte-Carlo simulations [18] in parallel for approximating the probability of each candidate answer. Very recently,

Olteanu and Wen [24] have further developed the idea of decomposing propositional formulas for deriving bounds based on a combination of partially expanded OBDDs and shared query plans, which can be exploited by top-k algorithms for early candidate pruning. While our bounding approach for propositional formulas is related to the one presented there, we more generally consider bounds for first-order lineage formulas, thus having a focus on the case when views are not materialized. With regard to computing bounds on confidences of lineage formulas, there are three major pieces of work [10, 23, 29], which we build upon for the propositional lineage case. However, we found the consideration of first-order lineage formulas to be a key to also incorporate pruning techniques known from managing extensional data [9] into a PDB setting.

With respect to uncertain views, in [16] MarkoViews were introduced recently, which build on [26] and allow views to express arbitrary correlations among its input tuples. Being more expressive than our uncertain views, they require potentially very large lineage formulas to be expressed, whereas our views merely add a conjunction with one additional random variable per grounding step to the lineage formulas.

9 Conclusions

We presented processing strategies for efficient top-k queries which lie at the intersection of probabilistic databases and probabilistic Datalog. Our approach does not assume safe query plans nor read-once lineage formulas, and it is able to return the exact top-k answers according to their marginal probabilities in many cases when exact confidence computations for these answers are intractable. Moreover, by focusing on non-materialized views, our pruning strategies can effectively help to avoid extensive data materialization and thus can contribute to reduced data computations. Extensions of our framework allow us to adopt top-k pruning strategies and sequential access patterns known from managing extensional data, and they even help to improve the runtime in the presence of recursive rules. In future work, our methods could be combined with techniques from [24] which we expect to enable even better performance gains.

Appendix A

A.1 Data

A.1.1 MayBMS / Postgres

We first create 6 tables, named `actedIn`, `directed`, `edited`, `hasCategory`, `produced` and `written`, using the following SQL commands. We query these tables for the Postgres measurements where no confidence computations take place.

```
CREATE TABLE actedIn (Id integer, Arg1 varchar(1023), Arg2
varchar(1023), Conf float8);

CREATE TABLE directed (Id integer, Arg1 varchar(1023), Arg2
varchar(1023), Conf float8);

CREATE TABLE edited (Id integer, Arg1 varchar(1023), Arg2
varchar(1023), Conf float8);

CREATE TABLE hasCategory (Id integer, Arg1 varchar(1023), Arg2
varchar(1023), Conf float8);

CREATE TABLE produced (Id integer, Arg1 varchar(1023), Arg2
varchar(1023), Conf float8);

CREATE TABLE written (Id integer, Arg1 varchar(1023), Arg2
varchar(1023), Conf float8);
```

Then we load the IMDB data using the copy command and create the corresponding uncertain tables using the "pick tuples" operation from MayBMS. A compressed file (637 MB) containing the data is available online [1].

[1] `http://www.mpi-inf.mpg.de/~miliaraki/probtopk/imdb-uni-maybms.zip`

```
COPY edited FROM 'EDITEDuniform-mb.txt' with delimiter '\t';

COPY actedIn FROM 'ACTEDINuniform-mb.txt' with delimiter '\t';

COPY directed FROM 'DIRECTEDuniform-mb.txt' with delimiter '\t';

COPY hasCategory FROM 'HASCATEGORYuniform-mb.txt' with delimiter
 '\t';

COPY produced FROM 'PRODUCEDuniform-mb.txt' with delimiter '\t';

COPY written FROM 'WRITTENuniform-mb.txt' with delimiter '\t';
```

```
CREATE TABLE editedprob AS (PICK TUPLES FROM edited
independently with probability conf);

CREATE TABLE actedInprob AS (PICK TUPLES FROM actedIn
independently with probability conf);

CREATE TABLE directedprob AS (PICK TUPLES FROM directed
independently with probability conf);

CREATE TABLE hasCategoryprob AS (PICK TUPLES FROM hasCategory
independently with probability conf);

CREATE TABLE producedprob AS (PICK TUPLES FROM produced
independently with probability conf);

CREATE TABLE writtenprob AS (PICK TUPLES FROM written
independently with probability conf);
```

For each relation REL, we create indices on each attribute except confidence.

```
CREATE INDEX REL_id_idx ON REL (id);

CREATE INDEX REL_arg1_idx ON REL (arg1);

CREATE INDEX REL_arg2_idx ON REL (arg2);
```

An additional relation called `dummy` is created to allow the union between uncertain tables where subqueries return a different number of columns since confidence fields are also considered. The following commands are used:

```
CREATE TABLE dummyCERT (Id integer, Dummyconf float8);

INSERT INTO dummyCERT VALUES (1, 1.0);

CREATE TABLE dummy AS (PICK TUPLES FROM dummyCERT independently
with probability dummyconf);
```

A.1.2 Trio

In the case of Trio, we create the Trio tables using the following TriQL commands:

```
CREATE TRIO TABLE EDITED (Id int, Arg1 varchar(1023), Arg2
varchar(1023), uncertain(Arg1, Arg2)) with confidences;
CREATE TRIO TABLE ACTEDIN (Id int, Arg1 varchar(1023), Arg2
varchar(1023), uncertain(Arg1, Arg2)) with confidences;
CREATE TRIO TABLE DIRECTED (Id int, Arg1 varchar(1023), Arg2
varchar(1023), uncertain(Arg1, Arg2)) with confidences;
CREATE TRIO TABLE HASCATEGORY (Id int, Arg1 varchar(1023), Arg2
varchar(1023), uncertain(Arg1, Arg2)) with confidences;
CREATE TRIO TABLE PRODUCED (Id int, Arg1 varchar(1023), Arg2
varchar(1023), uncertain(Arg1, Arg2)) with confidences;
CREATE TRIO TABLE WRITTEN (Id int, Arg1 varchar(1023), Arg2
varchar(1023), uncertain(Arg1, Arg2)) with confidences;
```

Then, we load the IMDB data using insert statements. The TriQL scripts used for this purpose can be found online [2]. Similar to MayBMS, we also create indices for all attributes.

A.2 Queries

A.2.1 $Q1$ (non-repeating hierarchical, safe)

Using Datalog notation we express the query as follows:
$level2(X,Y) :- edited(X,Y), directed(X,Z).$
$level2(X,Y) :- produced(X,Y), written(X,Z).$
$level1(X,Y) :- actedIn(X,Y).$
$level1(X,Y) :- level2(X,Y).$
$? - level1(X, CONSTANT).$

We express $Q1$ using the following commands in the query language of MayBMS which extends SQL:

```
CREATE TABLE level2 AS SELECT DISTINCT person, movie FROM (
SELECT DISTINCT edit.arg1 AS person, edit.arg2 AS movie FROM
editedprob edit, directedprob dir WHERE edit.arg1 = dir.arg1) AS
 level2a WHERE movie = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM producedprob prod,
writtenprob writ WHERE prod.arg1 = writ.arg1) AS level2b WHERE
movie = CONSTANT;
```

[2] http://www.mpi-inf.mpg.de/~miliaraki/probtopk/imdb-uni-trio.zip

```
CREATE TABLE level1 AS SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedInprob act, dummy d WHERE act.arg2 = CONSTANT) AS level1a
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT person
,movie FROM level2) AS level1b;

SELECT DISTINCT person, conf() FROM level1 GROUP BY person;
```

In Postgres queries are posed at the certain tables and no confidence computation takes place.

```
CREATE TABLE level2 AS SELECT DISTINCT person, movie FROM (
SELECT DISTINCT edit.arg1 AS person, edit.arg2 AS movie FROM
edited edit, directed dir WHERE edit.arg1 = dir.arg1) AS level2a
 WHERE movie = CONSTANT
UNION  SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM produced prod, written
writ WHERE prod.arg1 = writ.arg1) AS level2b WHERE movie =
CONSTANT;

CREATE TABLE level1 AS SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedIn act WHERE act.arg2 = CONSTANT) AS level1a
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT person
,movie FROM level2) AS level1b;

SELECT DISTINCT person FROM level1 GROUP BY person;
```

In Trio we use the TriQL language to express $Q1$. We need to create multiple tables. In all cases constants are pushed into the selections. Note that there are constants for which Trio returns the error "ERROR: MAXTUPS or MAXALTS too small".

```
CREATE TABLE level2a AS (SELECT edit.arg1 AS person, edit.arg2
AS movie FROM edited edit, directed dir WHERE edit.arg1 = dir.
arg1 AND edit.arg2 = CONSTANT
compute confidences);

CREATE TABLE level2b AS (SELECT prod.arg1 AS person, prod.arg2
AS movie FROM produced prod, written writ WHERE prod.arg1 = writ
.arg1 AND prod.arg2 = CONSTANT
compute confidences);

CREATE TABLE level2 AS (SELECT DISTINCT * FROM level2a compute
confidences)
UNION (SELECT DISTINCT person, movie FROM level2b compute
confidences);
```

```
CREATE TABLE level1a AS (SELECT person, movie FROM level2, rules
 WHERE rules.rule = 1);

CREATE TABLE level1b AS (SELECT arg1 AS person, arg2 AS movie
FROM actedin act WHERE act.arg2=CONSTANT compute confidences);

CREATE TABLE level1 AS (SELECT DISTINCT person, movie FROM
level1a compute confidences)
UNION (SELECT DISTINCT person, movie FROM level1b compute
confidences);

SELECT DISTINCT person FROM level1 compute confidences;
```

A.2.2 $Q2$ (repeating hierarchical, non-safe)

Using Datalog notation we express the query as follows:
$query(X,Y) : -actedIn(X,Y), hasCategory(Y, Action).$
$query(X,Y) : -produced(X,Y), hasCategory(Y, Action), hasCategory(Y, Drama).$
$query(X,Y) : -written(X,Y), hasCategory(Y, Drama).$
$? - query(X, CONSTANT).$
In MayBMS we express $Q2$ using the following commands in MayBMS query language. First, we create a table to compute the union of 3 subqueries, one for each rule, and then query this table. Relation dummy is included so that each subquery returns the same number of columns (including confidence values) allowing union operation.

```
CREATE TABLE query AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedInProb act, hasCategoryProb cat, dummy d WHERE act.arg2 =
cat.arg1 AND cat.arg2 = ''Action'') AS s1  WHERE s1.movie =
CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM producedProb prod,
hasCategoryProb cat1, hascategoryProb cat2 WHERE prod.arg2 =
cat1.arg1 AND prod.arg2 = cat2.arg1 AND cat2.arg2 = ''Action''
AND cat1.arg2=''Drama'') AS s2 WHERE s2.movie = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT writ.
arg1 AS person, writ.arg2 AS movie FROM writtenProb writ,
hasCategoryProb cat, dummy d WHERE writ.arg2 = cat.arg1 AND cat.
arg2 = ''Drama'')  AS s3 WHERE s3.movie = CONSTANT;

SELECT DISTINCT person, conf() FROM query GROUP BY person;
```

In Postgres queries are posed at the certain tables and no confidence computation takes place.

```
CREATE TABLE query AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedIn act, hasCategory cat WHERE act.arg2 = cat.arg1 AND cat.
arg2 = ''Action'') AS one WHERE one.movie = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM produced prod,
hasCategory cat1, hascategory cat2 WHERE prod.arg2 = cat1.arg1
AND prod.arg2 = cat2.arg1 AND cat2.arg2 = ''Action'' AND cat1.
arg2=''Drama'') AS two WHERE two.movie = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT writ.
arg1 AS person, writ.arg2 AS movie FROM written writ,
hasCategory cat WHERE writ.arg2 = cat.arg1 AND cat.arg2 = ''
Drama'') AS three WHERE three.movie = CONSTANT;

SELECT DISTINCT person FROM query;
```

In Trio the following TriQL commands are used:

```
CREATE TABLE query1 AS (SELECT act.arg1 AS person, act.arg2 AS
movie FROM actedin act, hascategory cat WHERE act.arg2 = cat.
arg1 AND cat.arg2 = ''Action'' AND cat.arg1 = CONSTANT);

CREATE TABLE query2 AS (SELECT prod.arg1 AS person, prod.arg2 AS
 movie FROM produced prod, hascategory cat1, hascategory cat2
WHERE prod.arg2 = cat1.arg1 AND prod.arg2 = cat2.arg1 AND cat1.
arg2= 'Action' AND cat2.arg2=''Drama'' AND prod.arg2 = CONSTANT)
;

CREATE TABLE query3 AS (SELECT writ.arg1 AS person, writ.arg2 AS
 movie FROM written writ, hascategory cat WHERE writ.arg2 = cat.
arg1 AND cat.arg2 = ''Drama'' AND cat.arg1 = CONSTANT);

CREATE TABLE query12 AS (SELECT DISTINCT person FROM query1
compute confidences)
UNION (SELECT DISTINCT person FROM query2 compute confidences);

CREATE TABLE query AS (SELECT DISTINCT person FROM query12
compute confidences)
UNION (SELECT DISTINCT person FROM query3 compute confidences);

SELECT DISTINCT person FROM query compute confidences;
```

A.2.3 $Q3$ (head hierarchical, non-safe)

Using Datalog notation we express the query as follows:
$level(X,Y) :- directed(X,Y), expensiveSubquery(A,B).$
$level(X,Y) :- edited(X,Y).$
$expensiveSubquery(A,B) :- actedIn(X,A), written(X,Y), hasCategory(Y,B).$
$level(X,Y) :- produced(X,Y), hasCategory(Y,Z).$
$? - level(CONSTANT,Y).$

In MayBMS we express $Q3$ using the following commands in MayBMS query language:

```
CREATE TABLE level2 AS SELECT DISTINCT person, movie FROM (
SELECT DISTINCT edit.arg1 AS person, edit.arg2 AS movie FROM
editedprob edit, directedprob dir WHERE edit.arg1 = dir.arg1) AS
 s1level2 WHERE movie = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM producedprob prod,
writtenprob writ WHERE prod.arg1 = writ.arg1) AS s3level2 WHERE
movie = CONSTANT;

CREATE TABLE toplevel AS SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedInprob act, dummy d WHERE act.arg2 = CONSTANT ) AS
toplevel1
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT person
,movie FROM level2) AS toplevel2;

SELECT DISTINCT person, conf() FROM toplevel GROUP BY person;
```

In Postgres queries are posed at the certain tables and no confidence computation takes place.

```
CREATE TABLE level2 AS SELECT person, movie FROM (SELECT edit.
arg1 AS person, edit.arg2 AS movie FROM edited edit, directed
dir WHERE edit.arg1 = dir.arg1) AS s1level2 WHERE movie =
CONSTANT
UNION SELECT person, movie FROM (SELECT prod.arg1 AS person,
prod.arg2 AS movie FROM produced prod, written writ WHERE prod.
arg1 = writ.arg1) AS s3level2 WHERE movie = CONSTANT;

CREATE TABLE toplevel AS SELECT person, movie FROM (SELECT act.
arg1 AS person, act.arg2 AS movie FROM actedIn act WHERE act.
arg2 = CONSTANT ) AS toplevel1
UNION SELECT person, movie FROM (SELECT person,movie FROM level2
) AS toplevel2;
```

```
SELECT DISTINCT person FROM toplevel GROUP BY person;
```

In Trio the following TriQL commands are used:

```
CREATE TABLE expensive AS (SELECT act.arg2 AS movie, cat.arg2 AS
 genre FROM actedin act, written writ, hascategory cat WHERE act
.arg1 = writ.arg1 AND writ.arg2 = cat.arg1 compute confidences);

CREATE TABLE level1 AS (SELECT dir.arg1 AS person, dir.arg2 AS
movie FROM directed dir, expensive exp WHERE dir.arg1 = CONSTANT
 compute confidences);

CREATE TABLE level2 AS (SELECT edit.arg1 AS person, edit.arg2 AS
 movie FROM edited edit WHERE edit.arg1 = CONSTANT compute
confidences);

CREATE TABLE level3 AS (SELECT prod.arg1 AS person, prod.arg2 AS
 movie FROM produced prod, hascategory cat WHERE prod.arg2 = cat
.arg1 AND prod.arg1 = CONSTANT compute confidences);

CREATE TABLE level12 AS (SELECT DISTINCT person, movie FROM
level1 compute confidences)
UNION (SELECT DISTINCT person, movie FROM level2 compute
confidences);

CREATE TABLE level123 AS (SELECT DISTINCT person, movie FROM
level12 compute confidences)
UNION (SELECT DISTINCT person, movie FROM level3 compute
confidences);

SELECT DISTINCT movie FROM level123 compute confidences;
```

A.2.4 $Q4$ (general unsafe)

Using Datalog notation we express the query as follows:
$unsafe(A, B) : -edited(A, Y), produced(X, Y), written(X, B), hasCategory(U, action).$
$unsafe(A, B) : -actedIn(A, B).$
$unsafe(A, B) : -directed(A, B).$
$? - unsafe(CONSTANT, Y).$
In MayBMS we express $Q4$ using the following commands in MayBMS query language:

```
CREATE TABLE unsafe AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT edit.arg1 AS person, prod.arg2 AS movie FROM
editedProb edit, producedProb prod, writtenProb writ,
hasCategoryProb cat WHERE edit.arg2 = prod.arg2 AND prod.arg1=
```

```
writ.arg1 AND cat.arg2 = ''Action'') AS unsafeone WHERE
unsafeone.person = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT act.
arg1 AS person, act.arg2 AS movie FROM actedInProb act, dummy d1
, dummy d2, dummy d3) AS unsafetwo WHERE unsafetwo.person =
CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT dir.
arg1 AS person, dir.arg2 AS movie FROM directedProb dir, dummy
d1, dummy d2, dummy d3) AS unsafethree WHERE unsafethree.person
= CONSTANT);

SELECT DISTINCT movie, conf() FROM unsafe GROUP BY movie;
```

In Postgres queries are posed at the certain tables and no confidence computation takes place.

```
CREATE TABLE unsafe AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT edit.arg1 AS person, prod.arg2 AS movie FROM
editedProb edit, producedProb prod, writtenProb writ,
hasCategoryProb cat WHERE edit.arg2 = prod.arg2 AND prod.arg1=
writ.arg1 AND cat.arg2 = ''Action'') AS unsafeone WHERE
unsafeone.person = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT act.
arg1 AS person, act.arg2 AS movie FROM actedInProb act) AS
unsafetwo WHERE unsafetwo.person = CONSTANT
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT dir.
arg1 AS person, dir.arg2 AS movie FROM directedProb dir) AS
unsafethree WHERE unsafethree.person = CONSTANT);

SELECT DISTINCT movie FROM unsafe;
```

In Trio the following TriQL commands are used:

```
CREATE TABLE unsafe1 AS (SELECT edit.arg1 AS person, writ.arg2
AS movie FROM edited edit, produced prod, written writ,
hascategory cat WHERE edit.arg2 = prod.arg2 AND prod.arg1 = writ
.arg1 AND cat.arg2 = 'Action' AND edit.arg1 = CONSTANT compute
confidences);

CREATE TABLE unsafe2 AS (SELECT act.arg1 AS person, act.arg2 AS
movie FROM actedin act WHERE act.arg1 = CONSTANT compute
confidences);

CREATE TABLE unsafe3 AS (SELECT dir.arg1 AS person, dir.arg2 AS
movie FROM directed dir WHERE dir.arg1 = CONSTANT compute
confidences);

CREATE TABLE unsafe12 AS (SELECT DISTINCT person, movie FROM
unsafe1 compute confidences)
```

```
UNION (SELECT DISTINCT person, movie FROM unsafe2 compute
confidences);

CREATE TABLE unsafe AS (SELECT DISTINCT person, movie FROM
unsafe12 compute confidences)
UNION (SELECT DISTINCT person, movie FROM unsafe3 compute
confidences);

SELECT DISTINCT movie FROM unsafe compute confidences;
```

A.2.5 $Q5$ (one proof)

Using Datalog notation we express the query as follows:
$query(A, B) : -actedIn(X, A), directed(X, B).$
$? - query(A, CONSTANT).$

In MayBMS we express $Q5$ using the following query:

```
SELECT movie1, conf() FROM SELECT DISTINCT act.arg2 AS movie1,
dir.arg2 AS movie2 FROM actedinprob act, directedprob dir WHERE
act.arg1 = dir.arg1 AS oneproof WHERE movie2 = CONSTANT;
```

In Postgres query is posed at the certain tables and no confidence computation takes place.

```
SELECT movie FROM SELECT DISTINCT act.arg2 AS movie1, dir.arg2
AS movie2 FROM actedin act, directed dir WHERE act.arg1 = dir.
arg1 AS oneproof WHERE movie2 = CONSTANT;
```

In the case of Trio, we first create a table and then run the query, as follows.

```
CREATE TABLE oneproof AS (SELECT act.arg2 AS movie1, dir.arg2 AS
 movie2 FROM actedin act, directed dir WHERE act.arg1=dir.arg1
AND dir.arg2=CONSTANT compute confidences);

SELECT DISTINCT movie1 FROM oneproof compute confidences;
```

A.2.6 $Q6$ (three proofs)

Using Datalog notation we express the query as follows:
$query(X, Y) : -written(X, Y), directed(X, Y).$
$query(X, Y) : -level2(X, Y).$
$level2(X, Y) : -actedIn(X, Y), hasCategory(Y, action).$
$level2(X, Y) : -produced(X, Y).$
$? - query(X, CONSTANT)$

In MayBMS we express $Q6$ using the following commands in MayBMS query language:

```
CREATE TABLE level2 AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedInprob act, hasCategoryProb cat WHERE act.arg2 = cat.arg1
AND cat.arg2 = ''Action'' AND cat.arg1 = CONSTANT) AS level12
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM producedProb prod, dummy
 d WHERE prod.arg2 = CONSTANT) AS level22);

CREATE TABLE query AS SELECT DISTINCT person, movie FROM (SELECT
 DISTINCT writ.arg1 AS person, writ.arg2 AS movie FROM
writtenprob writ, directedprob dir WHERE writ.arg1 = dir.arg1
AND writ.arg2 = dir.arg2 AND writ.arg2= CONSTANT) AS query
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT person
, movie FROM level2) AS level2;

SELECT DISTINCT person, conf() FROM query GROUP BY person;
```

In Postgres queries are posed at the certain tables and no confidence computation takes place.

```
CREATE TABLE level2 AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedIn act, hasCategory cat WHERE act.arg2 = cat.arg1 AND cat.
arg2 = 'Action' AND cat.arg1 = CONSTANT) AS level12
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT prod.
arg1 AS person, prod.arg2 AS movie FROM produced prod WHERE prod
.arg2 = CONSTANT) AS level22) ;

CREATE TABLE query AS SELECT DISTINCT person, movie FROM (SELECT
 DISTINCT writ.arg1 AS person, writ.arg2 AS movie FROM written
writ, directed dir WHERE writ.arg1 = dir.arg1 AND writ.arg2 =
dir.arg2 AND writ.arg2= CONSTANT) AS query
UNION SELECT DISTINCT person, movie FROM (SELECT DISTINCT person
, movie FROM level2) AS level2;

SELECT DISTINCT person FROM query GROUP BY person;
```

In Trio the following TriQL commands are used:

```
CREATE TABLE level2produced2 AS (SELECT prod.arg1 AS person,
prod.arg2 AS movie FROM produced prod WHERE prod.arg2=CONSTANT
compute confidences);

CREATE TABLE level2actcat2 AS (SELECT act.arg1 AS person, act.
arg2 AS movie FROM actedin act, hascategory cat WHERE act.arg2 =
 cat.arg1 AND cat.arg2 = ''Action'' AND cat.arg1 = CONSTANT
compute confidences);
```

```
CREATE TABLE level2all2 AS (SELECT DISTINCT * FROM
level2produced2 compute confidences)
UNION (SELECT DISTINCT * FROM level2actcat2 compute confidences)
;

CREATE TABLE querylevel2 AS (SELECT person, movie FROM
level2all2 compute confidences);

CREATE TABLE querywritdir AS (SELECT writ.arg1 AS person, writ.
arg2 AS movie FROM written writ, directed dir WHERE writ.arg1 =
dir.arg1 AND writ.arg2 = dir.arg2 AND writ.arg2 = CONSTANT
compute confidences);

CREATE TABLE queryall AS (SELECT DISTINCT person FROM
querylevel2 compute confidences)
UNION (SELECT DISTINCT person FROM querywritdir compute
confidences);

SELECT DISTINCT person FROM queryall compute confidences;
```

A.2.7 $Q7$ (join of existential relations)

Using Datalog notation we express the query as follows:
$? - produced(X, CONSTANT), written(X, Y)$.
In MayBMS we express $Q7$ by creating first a table joining the two relations and then pose the query.

```
CREATE TABLE joinonly AS (SELECT DISTINCT prod.arg1 AS person,
writ.arg2 AS movie FROM producedProb prod, writtenProb writ
WHERE prod.arg1 = writ.arg1 AND prod.arg2 = CONSTANT);

SELECT DISTINCT person, movie, conf() FROM joinonly GROUP BY
person, movie;
```

In Postgres we pose the following query:

```
SELECT DISTINCT prod.arg1 AS person, writ.arg2 AS movie FROM
produced prod, written writ WHERE prod.arg1 = writ.arg1 AND prod
.arg2 = CONSTANT
```

In Trio the following TriQL commands are used:

```
CREATE TABLE joinonly AS (SELECT prod.arg1 AS person, writ.arg2
AS movie FROM produced prod, written writ WHERE prod.arg1=writ.
arg1 AND prod.arg2=CONSTANT compute confidences);

SELECT DISTINCT person, movie FROM joinonly compute confidences;
```

A.2.8 $Q8$ (non-read-once)

Using Datalog notation we express the query as follows:
$query(X,Y) : -actedIn(X,Y), hasCategory(Y, Action).$
$query(X,Y) : -produced(X,Y), hasCategory(Y, Action).$
$query(X,Y) : -produced(X,Y), hasCategory(Y,Z), notEquals(Z, Action).$
$? - query(X, CONSTANT)$
In MayBMS we express $Q8$ by creating first a table joining the two relations and then pose the query.

```
CREATE TABLE query AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedInprob act, hascategoryprob cat WHERE act.arg2=cat.arg1
AND cat.arg2=''Action'') AS one WHERE one.movie = CONSTANT
UNION SELECT DISTINCT two.person, two.movie FROM(SELECT DISTINCT
 prod.arg1 AS person, prod.arg2 AS movie FROM producedprob prod,
 hascategoryprob cat WHERE prod.arg2=cat.arg1 AND cat.arg2=''
Action'') AS two WHERE two.movie = CONSTANT
UNION SELECT DISTINCT three.person, three.movie FROM (SELECT
DISTINCT prod.arg1 AS person, prod.arg2 AS movie FROM
producedprob prod, hascategoryprob cat WHERE prod.arg2=cat.arg1
AND cat.arg2!=''Action'') AS three WHERE three.movie = CONSTANT)
;

SELECT DISTINCT person, conf() FROM query GROUP BY person;
```

In Postgres we pose the following query:

```
CREATE TABLE query AS (SELECT DISTINCT person, movie FROM (
SELECT DISTINCT act.arg1 AS person, act.arg2 AS movie FROM
actedIn act, hascategory cat WHERE act.arg2=cat.arg1 AND cat.
arg2=''Action'') AS one WHERE one.movie = CONSTANT
UNION SELECT DISTINCT two.person, two.movie FROM(SELECT DISTINCT
 prod.arg1 AS person, prod.arg2 AS movie FROM produced prod,
hascategory cat WHERE prod.arg2=cat.arg1 AND cat.arg2='Action'
) AS two WHERE two.movie = CONSTANT
UNION SELECT DISTINCT three.person, three.movie FROM (SELECT
DISTINCT prod.arg1 AS person, prod.arg2 AS movie FROM produced
prod, hascategory cat WHERE prod.arg2=cat.arg1 AND cat.arg2!=''
Action'') AS three WHERE three.movie = CONSTANT);

SELECT DISTINCT person FROM query;
```

In Trio the following TriQL commands are used:

```
CREATE TABLE query1 AS (SELECT act.arg1 AS person, act.arg2 AS
movie FROM actedin act, hascategory cat WHERE act.arg2 = cat.
arg1 AND cat.arg2='Action' AND cat.arg1=CONSTANT compute
confidences);
```

```
CREATE TABLE query2 AS (SELECT prod.arg1 AS person, prod.arg2 AS
 movie FROM produced prod, hascategory cat WHERE prod.arg2 = cat
.arg1 AND cat.arg2 = 'Action' AND cat.arg1 = CONSTANT compute
confidences);

CREATE TABLE query3 AS (SELECT prod.arg1 AS person, prod.arg2 AS
 movie FROM produced prod, hascategory cat WHERE prod.arg2 = cat
.arg1 AND cat.arg2 <> 'Action' AND cat.arg1 = CONSTANT compute
confidences);

CREATE TABLE queryall1 AS (SELECT DISTINCT * FROM query1 compute
 confidences)
UNION (SELECT DISTINCT * FROM query2 compute confidences);

CREATE TABLE queryall AS (SELECT DISTINCT * FROM queryall1
compute confidences)
UNION (SELECT DISTINCT * FROM query3 compute confidences);

SELECT DISTINCT person FROM queryall compute confidences;
```

A.2.9 $Q9$ (nesting-depth = 1)

Using Datalog notation we express the query as follows:
$query(X,Y) : -diedIn(X,Y), bornIn(X,Y).$
$query(X,Y) : -livesIn(X,Y), isMarriedTo(Z,Y).$
$query(X,Y) : -isCitizenOf(X,Y).$
$query(X,Y) : -politicianOf(X,Y).$
$? - query(X, CONSTANT)$
PostgreSQL's query plan is obtained by:

```
EXPLAIN
(SELECT DISTINCT arg1 FROM
((SELECT t1.arg1, t1.arg2 FROM livesIn AS t1 JOIN isMarriedTo AS
 t2 ON t1.arg2=t2.arg2)
UNION (SELECT t1.arg1, t1.arg2 FROM diedIn AS t1 JOIN bornIn AS
t2 ON t1.arg2=t2.arg2 AND t1.arg1=t2.arg1)
UNION (SELECT arg1, arg2 FROM politicianOf)
UNION (SELECT arg1, arg2 FROM isCitizenOf))
AS x WHERE arg2='CONSTANT')
```

A.2.10 $Q10$ (nesting-depth = 2)

Using Datalog notation we express the query as follows:
$query(X,Y) : -isCitizenOf(X,Y), diedOnDate(X,Z).$

$query(X,Y) : -bornIn(X,Y).$
$query(X,Y) : -politicianOf(X,Y), hasChild(X,Z).$
$query(X,Y) : -level2(X,Y).$
$level2(X,Y) : -diedIn(X,Y).$
$level2(X,Y) : -livesIn(X,Y).$
$? - query(X, CONSTANT)$
PostgreSQL's query plan is obtained by:

```
EXPLAIN
(SELECT DISTINCT arg1 FROM
((SELECT arg1, arg2 FROM bornIn)
UNION (SELECT t1.arg1, t1.arg2 FROM isCitizenOf AS t1 JOIN
diedIn AS t2 ON t1.arg1=t2.arg1)
UNION ( SELECT arg1, arg2 FROM
((SELECT arg1, arg2 FROM diedIn)
UNION SELECT arg1, arg2 FROM livesIn) AS z)
UNION (SELECT t1.arg1, t1.arg2 FROM politicianOf AS t1 JOIN
hasChild AS t2 ON t1.arg1=t2.arg1))
AS x WHERE arg2='CONSTANT')
```

A.2.11 $Q11$ (nesting-depth = 3)

Using Datalog notation we express the query as follows:
$query(X,Y) : -livesIn(X,Z), diedOnDate(X,Y).$
$query(X,Y) : -level2(X,Y).$
$level2(X,Y) : -bornOnDate(X,Y), bornIn(X,Z).$
$level2(X,Y) : -level3(X,Y)$
$level3(X,Y) : -createdOnDate(X,Y)$
$level3(X,Y) : -writtenInYear(X,Y)$
$? - query(X, CONSTANT)$
PostgreSQL's query plan is obtained by:

```
EXPLAIN
(SELECT DISTINCT arg1 FROM
((SELECT t2.arg1, t2.arg2 FROM livesIn AS t1 JOIN diedInOnDate
AS t2 ON t1.arg1=t2.arg2)
UNION ((SELECT t1.arg1, t1.arg2 FROM bornOnDate AS t1 JOIN
bornIn AS t2 ON t1.arg1=t2.arg2)
UNION (SELECT arg1, arg2 FROM createdOnDate
UNION SELECT arg1, arg2 FROM writtenInYear)))
AS x WHERE arg2='CONSTANT')
```

A.2.12 $Q12$ (with uncertain views)

Using Datalog notation we express the query as follows:

$query(X, Y) : -actedIn(X, Y).$
$query(X, Y) : -produced(X, Y).$
$query(X, Y) : -edited(X, Y).$
$? - query(CONSTANT, Y)$
For the uncertain view setting, we attached the confidences 0.1, 0.2, and 1.0 to the rules (from top to bottom).

A.2.13 $Q13$ (with uncertain views)

Using Datalog notation we express the query as follows:
$query(X, Y) : -level(X, Y).$
$query(X, Y) : -edited(X, Y).$
$level(X, Y) : -actedIn(X, Y), hasCategory(Y, Action).$
$level(X, Y) : -directed(X, Y), written(X, Z).$
$? - query(CONSTANT, Y)$
For the uncertain view setting, we attached the confidences 0.2, 1.0, 1.0, and 1.0 to the rules (from top to bottom).

A.2.14 $Q14$ (recursion)

Using Datalog notation we express the query as follows:
$ancestor(X, Y) : -hasChild(X, Y).$
$ancestor(X, Y) : -hasChild(X, Y), ancestor(Y, Z).$
$? - ancestor(CONSTANT, Y)$

A.2.15 $Q15$ (recursion)

Using Datalog notation we express the query as follows:
$politician(A, B) : -hasPredecessor(A, C), politician(C, B).$
$politician(A, B) : -politicianOf(A, B).$
$? - politician(A, CONSTANT)$

A.2.16 Confidence Distributions

Using Datalog notation we express the query as follows:
$? - hasCategory(Y, Talk - Show), produced(X, Y)$

A.2.17 Table Materialization

Using Datalog notation we express the query as follows:
$? - directed(X, Z), hasCategory(Z, Comedy)$

In MayBMS we first create a table joining the two relations and then pose the query to compute the confidences.

```
CREATE TABLE comedydirectors AS SELECT DISTINCT dir.arg1 AS
director, cat.arg1 AS movie FROM directedprob dir,
hascategoryprob cat WHERE dir.arg2 = cat.arg1 AND cat.arg2 = ''
Comedy'';

SELECT DISTINCT director, movie, conf() FROM comedydirectors
GROUP BY director, movie;
```

In Postgres we use the following SQL commands:

```
CREATE TABLE comedydirectors AS SELECT DISTINCT dir.arg1 AS
director, cat.arg1 AS movie FROM directed dir, hascategory cat
WHERE dir.arg2 = cat.arg1 AND cat.arg2 = ''Comedy'';

SELECT DISTINCT director, movie FROM comedydirectors;
```

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):159–187, 1991.

[3] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, pages 983–992, 2008.

[4] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

[5] J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *SIGMOD*, pages 891–893, 2005.

[6] P. Buneman and W. C. Tan. Provenance in databases. In *SIGMOD*, pages 1171–1173, 2007.

[7] N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *PODS*, pages 293–302, 2007.

[8] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16:523–544, 2007.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[10] R. Fink and D. Olteanu. On the optimal approximation of queries using tractable propositional languages. In *ICDT*, pages 174–185, 2011.

[11] N. Fuhr. Probabilistic Datalog - a logic for powerful retrieval methods. In *SIGIR*, pages 282–290, 1995.

[12] T. Ge, S. B. Zdonik, and S. Madden. Top-*k* queries on uncertain data: on score distribution and typical answers. In *SIGMOD*, pages 375–388, 2009.

[13] G. Gottlob and C. H. Papadimitriou. On the complexity of single-rule datalog queries. *Inf. Comput.*, 183(1):104–122, 2003.

[14] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[15] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[16] A. Jha and D. Suciu. Probabilistic databases with MarkoViews. To appear in PVLDB 2012.

[17] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, pages 841–852, 2011.

[18] R. M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64, 1983.

[19] A. Kimmig, G. V. den Broeck, and L. D. Raedt. An algebraic prolog for reasoning about possible worlds. In *AAAI*, 2011.

[20] C. Koch and D. Olteanu. Conditioning probabilistic databases. *PVLDB*, 1(1):313–325, 2008.

[21] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1):502–513, 2009.

[22] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, pages 640–651, 2009.

[23] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, pages 145–156, 2010.

[24] D. Olteanu and H. Wen. Ranking in probabilistic databases: Complexity and efficient algorithms. To appear in ICDE 2012.

[25] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.

[26] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[27] T. Rölleke and N. Fuhr. Probabilistic reasoning for large scale databases. In *BTW*, pages 118–132, 1997.

[28] D. Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82:273–302, April 1996.

[29] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27:633–655, 1980.

[30] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, pages 1023–1032, 2008.

[31] P. Sen, A. Deshpande, and L. Getoor. PrDB: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.

[32] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 3(1):1068–1079, 2010.

[33] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *SIGMOD*, pages 1239–1242, 2008.

[34] M. Soliman, I. Ilyas, and K. Chen-Chuan Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896 –905, 2007.

[35] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[36] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[37] J. D. Ullman and M. Y. Vardi. The complexity of ordering subgoals. In *PODS*, pages 74–81, 1988.

[38] J. Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. Springer, 2008.