Ranking Query Answers in Probabilistic Databases: Complexity and Efficient Algorithms

Dan Olteanu Hongkai Wen

Department of Computer Science, University of Oxford, Oxford, OX1 3QD, UK
{dan.olteanu,hongkai.wen}@cs.ox.ac.uk

***Abstract*—In many applications of probabilistic databases, the probabilities are mere degrees of uncertainty in the data and are not otherwise meaningful to the user. Often, users care only about the ranking of answers in decreasing order of their probabilities or about a few most likely answers.**

In this paper, we investigate the problem of ranking query answers in probabilistic databases. We give a dichotomy for ranking in case of conjunctive queries without repeating relation symbols: it is either in polynomial time or #P-hard. Surprisingly, our syntactic characterisation of tractable queries is not the same as for probability computation. The key observation is that there are queries for which probability computation is #P-hard, yet ranking can be computed in polynomial time. This is possible whenever probability computation for distinct answers has a common factor that is hard to compute but irrelevant for ranking.

We complement this tractability analysis with an effective ranking technique for conjunctive queries. Given a query, we construct a share plan, which exposes subqueries whose probability computation can be shared or ignored across query answers. Our technique combines share plans with incremental approximate probability computation of subqueries.

We implemented our technique in the SPROUT query engine and report on performance gains of orders of magnitude over Monte Carlo simulation using FPRAS and exact probability computation based on knowledge compilation.

I. Introduction

In this paper we study the problem of ranking the answers of conjunctive queries in probabilistic databases, where the ranking is based on the probabilities of the answers. A special instance of this problem is computing the top-k most probable answers [22]. This is a fundamental problem of high practical relevance. In many applications of probabilistic databases, the probabilities are mere degrees of uncertainty in the data, and do not have otherwise any semantics that is meaningful to the user. Often, users care only about ranking the answers and want to retrieve a few most likely answers.

Two simple observations are at the core of our ranking approach. Firstly, to compute the *exact ranking* of query answers, *approximate probabilities* of the individual answers may suffice. Secondly, the probability computation for distinct query answers may share a common factor which can be computed only once or even uniformly ignored for all answers.

These observations suggest that ranking is easier than exact query evaluation. A result of this paper is a dichotomy for ranking: We show that the syntactic notion of *head-hierarchical* queries partitions the language of non-repeating conjunctive queries into polynomial-time and #P-hard queries. This notion is strictly weaker than that of *hierarchical* queries, which separates tractable from hard queries for probability computation [5]. Queries that are tractable for ranking are thus not necessarily tractable for query evaluation.

Further inspection of these two observations leads to a practical ranking approach, for which we need two main ingredients: an approximation procedure that can incrementally refine lower and upper bounds on the probability of a query answer, and a static analysis procedure on the query structure to identify for which subqueries their computation can be shared across distinct query answers.

Our ranking approach does precisely this: (1) we introduce so-called *share plans*, which are statically inferred from the query and expose subqueries whose probability computation can be shared across several query answers, and (2) extend an existing deterministic approximation procedure [18] to incrementally compute bounds on the probability of shared subqueries and of query answers. Alternatively, share plans can be viewed as recipes for factorisations of the events of query answers that enable at runtime easy identification of factors common across query answers. Our approximation scheme is based on incremental decomposition of the events of query answers using knowledge compilation techniques such as Shannon expansion and independence partitioning [8].

Example 1.1: Consider the social network database in Fig. 1. Relation *Trends* stores popular keywords that appear in user tweets, relation *Follows* records users (*follower*) who follow tweets by other users (*user*), relation *Mentions* logs the events that $user_2$ has been mentioned by $user_1$'s tweets, and relation *Tweets* pairs users with their tweets.

We assume in this example that the relations are tuple-independent, i.e., each relation has an event column E storing pairwise independent Boolean random variables; we explain the case of databases with arbitrary correlations later in the paper. A tuple-independent database represents exponentially many possible instances, with one instance for each total valuation of the random variables in the database. For example, a valuation that maps x_1, y_1, z_1, and u_1 to true and all other variables to false defines the instance with the first tuple in each of the four relations. The probability of this instance is the product of the probabilities of x_1, y_1, z_1, and u_1 being true and of all remaining variables being false. The uncertainty in the data may originate from various sources, e.g., approximate matching between keywords or incomplete user information.

Fig. 1 shows the answers of two queries together with their events. The event of a join of two tuples t_1 and t_2 is the

Trends

keyword	user	E
#database	Dan	x_1
#database	Hongkai	x_2
#sensors	Niki	x_3
#sensors	Hongkai	x_4
#wimbledon	Hongkai	x_5
#skylab	Hongkai	x_6

Follows

user	follower	E
Dan	Niki	y_1
Dan	Hongkai	y_2
Niki	Hongkai	y_3
Hongkai	Dan	y_4
Hongkai	Niki	y_5

Mentions

tweetId	$user_1$	$user_2$	E
1	Hongkai	Dan	z_1
2	Dan	Niki	z_2
3	Hongkai	Dan	z_3
1	Hongkai	Niki	z_4
2	Dan	Hongkai	z_5
4	Niki	Hongkai	z_6

Tweets

tweetId	user	E
1	Hongkai	u_1
2	Dan	u_2
3	Hongkai	u_3
4	Niki	u_4

$Q_1(Z)$:- Trends(X,Y), Follows(Y,Z)

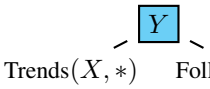
follower	E
Dan	$y_4(x_2 \vee x_4 \vee x_5 \vee x_6)$
Niki	$y_1x_1 \vee y_5(x_2 \vee x_4 \vee x_5 \vee x_6)$
Hongkai	$y_2x_1 \vee y_3x_3$

$Q_2(X)$:- Trends(X,Y), Follows(Y,Z), Mentions(U,Y,Z), Tweets(U,Y)

keyword	E
#database	$\Phi_1 = x_1(y_1z_2u_2 \vee y_2z_5u_2) \vee x_2(y_4z_1u_1 \vee y_4z_3u_3 \vee y_5z_4u_1)$
#sensors	$\Phi_2 = x_3y_3z_6u_4 \vee x_4(y_4z_1u_1 \vee y_4z_3u_3 \vee y_5z_4u_1)$
#wimbledon	$\Phi_3 = x_5(y_4z_1u_1 \vee y_4z_3u_3 \vee y_5z_4u_1)$
#skylab	$\Phi_4 = x_6(y_4z_1u_1 \vee y_4z_3u_3 \vee y_5z_4u_1)$

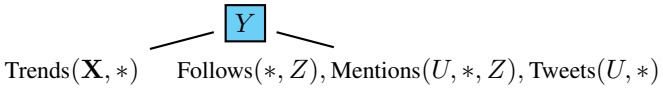
Fig. 1: Social network database where each tuple is associated with a probabilistic event. Query Q_1 asks for followers of users who contributed to trends. Query Q_2 asks for topics posted by users who have mentioned their followers.

conjunction of the events of t_1 and t_2. The event of a union of two equal tuples t_1 and t_2 is the disjunction of the events of t_1 and t_2. The event Φ of a query answer t is important for query evaluation and ranking since the probability of Φ is the probability of t [26]. To speed up ranking and even query evaluation, we identify (sub)events that are shared by several answers. In our example, the shared events are underlined. The events are factorised such that it is easy to see whether they share common factors. The structure of this factorisation can be statically inferred from the query in a share plan. A share plan for Q_1 is the following:



This plan encodes the following. The query answers are values of the *head* variable Z displayed in bold. The variable Y is *root*, i.e., it occurs in all relation symbols. By fixing a value (*) for Y, the subqueries Trends$(X,*)$ and Follows$(*,Z)$ become disconnected. Distinct answers can be paired with the same tuples in relation *Trends* and share their events. Since the relations are tuple-independent in our example, we can also infer that for two distinct Y-values, the events of any of the two subqueries are independent, and that the events of subqueries Trends$(X,*)$ and Follows$(*,Z)$ are independent.

To compute the probabilities of query answers, we need to compute the probabilities of shared events only once. For ranking, we may even discard the latter entirely. We explain this for Q_2 in Fig. 1. This query is hard for both probability computation and ranking. A share plan for Q_2 is as follows:



The answers are X-values and several of them can be paired with the same value of the root variable Y and thus share its event. The following rewriting of Q_2 makes this explicit:

$Q_2(X)$:- Trends(X,Y), $Q_3(Y)$
$Q_3(Y)$:- Follows(Y,Z), Mentions(U,Y,Z), Tweets(U,Y)

For any two Y-values a and b, the events of the hard Boolean queries $Q_3[Y/a]$ and $Q_3[Y/b]$ are independent, since Y is a root variable in Q_3. Therefore, the events of the query answers can be expressed as disjunctions of sub-events that are conjunctions of events from Trends and independent events of hard Boolean queries $Q_3[Y/a]$. Fig. 1 shows the events of each query answer factorised in a form that follows the structure of the rewritten query and of the share plan. The (underlined) event of the hard query $Q_3[Y/\text{Hongkai}]$ is common to the events of all query answers and its probability computation can be shared across all answers. Also, to decide $P(\Phi_3) \leq P(\Phi_4)$, we only need to decide $P(x_5) \leq P(x_6)$.

In this paper, we show how to efficiently approximate the probabilities of events with shared factors and incrementally refine their lower and upper bounds as much as needed to compute top-k or ranked answers. □

The main contributions of this paper are as follows:

- We introduce a technique for ranking query answers and computing the top-k most probable answers for conjunctive queries in probabilistic databases. This technique uses statically-derived share plans and a deterministic approximation procedure that can incrementally refine lower and upper bounds on probabilities of query answers. Although it works for databases with arbitrary correlations, it is particularly effective for tuple-independent databases.
- We show that for non-repeating conjunctive queries on tuple-independent databases ranking is either in polynomial time or #P-hard and give a syntactic characterisation of the tractable non-repeating conjunctive queries.
- We have implemented our technique in the SPROUT query engine, which is part of the probabilistic database management system MayBMS.
- Extensive experimental evaluation with tractable and hard queries on probabilistic TPC-H and synthetic data sets shows performance gains of orders of magnitude over Monte Carlo simulation using FPRAS and exact probability computation based on knowledge compilation.

There is a solid body of work on ranking in probabilistic databases that defines ranking as a function of probabilities *and* additional score functions. The proposed techniques essentially exploit the interplay of various ranking functions and uncertainty in the data, e.g, [15]. A comprehensive survey of this strand has been recently compiled by Ilyas and Soliman [13].

The closest in spirit to our work is a paper by Ré, Dalvi, and Suciu, where ranking of query answers is based on probabilities. Their approach is to run in parallel Monte-Carlo simulations for the events of query answers and approximate their probabilities only to the extent needed to compute the top-k answers. The approximation of probabilities is done using an adaptation of the fully polynomial randomised approximation scheme (FPRAS) for model counting of DNF formulas [27]. The use of FPRAS for ranking has three important limitations: (1) the achieved ranking is only a probabilistic approximation of the correct one; (2) running one more Monte Carlo step does not necessarily lead to a refinement of probability bounds, and hence the approximation is not truly incremental [26]; (3) FPRAS sees events as black boxes and does not exploit their structure for faster evaluation [14] nor sharing of common factors. Our approach has none of these limitations: it uses truly incremental approximation and exploits sharing and structure of the events. We show experimentally that the third limitation severely hinders the scalability of FPRAS for ranking.

In a seminal paper [5], Dalvi and Suciu gave the dichotomy of probability computation for non-repeating conjunctive queries and introduced the notion of safe plans. Our share plans become safe plans in case of queries that are tractable for probability computation, yet they are designed to be particularly effective for hard queries, for which safe plans cannot exist. Our ranking dichotomy builds upon the dichotomy of probability computation, yet it is different.

Sharing computation for query evaluation and optimisation has been considered in the context of inference in graphical models [23] and answering queries using materialised views in probabilistic databases [7]. Factorisations of events of query answers are used for efficient probability computation in probabilistic databases, where the events can be rewritten as so-called read-once functions [16], [18], [24], and in provenance and relational databases for the purpose of compact representations and tractable query evaluation [19], [20].

II. Preliminaries

A. Probabilistic Events and Databases

Let $\mathbf{X}$ be a finite set of independent Boolean random variables with positive rational probabilities given by a function P. An *event* is a propositional formula over variables from $\mathbf{X}$. Two events are (syntactically) *independent* if their sets of variables are disjoint. A *valuation* over $\mathbf{X}$ is an assignment of each random variable in $\mathbf{X}$ to true or false.

A *probabilistic relation* is a relation with a distinct column E that stores an event for each tuple. A probabilistic database is a set of probabilistic relations. A database, where the events are pairwise independent, is called *tuple-independent*. Fig. 1 shows a tuple-independent database with four relations.

A probabilistic database defines a *finite set of possible worlds* by the one-to-one correspondence between worlds and valuations over $\mathbf{X}$: The world defined by a valuation ν consists of the tuples whose associated events are satisfied by ν.

B. Queries

We consider the language of conjunctive (or select-project-join) queries in datalog notation [1]. A query has the form

$$Q(\overline{X_0}) \text{ :- } R_1(\overline{X_1}), \ldots, R_n(\overline{X_n}),$$

where the query head is $Q(\overline{X_0})$ and the query body is a conjunction of relation symbols $R_1(\overline{X_1}), \ldots, R_n(\overline{X_n})$. A variable that appears in two sets $\overline{X_i}$ and $\overline{X_j}$ in the body defines a join between relations R_i and R_j. The query variables in $\overline{X_0}$ are *head* variables and they must also appear in the body. All other query variables are called *existential*.

A query Q can be represented as a graph, where relation symbols, including Q's head, are nodes and each query variable induces an edge between each pair of symbols that contains it. Each connected component in the graph corresponds to a disjoint set of relation symbols in Q. The subqueries corresponding to different connected components are *disconnected* in the sense that they do not have common query variables. The *head* component is the connected component that contains the query head. The *head-restriction* of Q is the query defined by the head component of Q, i.e., obtained from Q by dropping all relation symbols that represent edges in non-head components of Q's graph. Head restrictions of Boolean queries are Boolean queries with empty bodies.

A query Q is *non-repeating* if no relation symbol occurs more than once in Q. A query is *hierarchical* if for any two existential variables, either their sets of relation symbols are disjoint, or one set is contained in the other [5], [26]. A query variable that appears in all relation symbols of Q is called *root* of Q. The following weaker notion of hierarchical queries is particularly relevant to this work.

Definition 2.1: A conjunctive query Q is *head-hierarchical* if its head-restriction is hierarchical. □

Example 2.2: The query Q_1 from the introduction is hierarchical and hence head-hierarchical. The query Q_2 is not hierarchical, since its subquery Q_3 is not hierarchical as witnessed by query variables U and Z. The variable Y is root in Q_1 and both root and head in Q_3. A modified query $Q_2'(X)$:- Trends$(X, Y), Q_3(V)$ is head-hierarchical since its head restriction $Q_2'(X)$:- Trends(X, Y) is hierarchical.

The following three queries are not hierarchical:

$$\begin{aligned}
Q_a(X) &\text{ :- } R_1(X,Y), R_2(Y,Z), R_3(Y,Z,U), R_4(Y,W),\\
&\quad R_5(Y,W,V), R_6(Y,V)\\
Q_b(X,Z) &\text{ :- } R_1(X,Y), R_2(Y,Z), R_3(Y,U), R_4(Y,U,W),\\
&\quad R_5(Y,W), R_6(H), R_7(H,I), R_8(I)\\
Q_c(X,Z) &\text{ :- } R_1(X,U), R_2(U,Y,Z), R_3(Y,V,W), R_4(Y,V),\\
&\quad R_5(Y,V,H)
\end{aligned}$$

as witnessed by variables W and V for Q_a, U and W for Q_b, and U and Y for Q_c. The head-restriction Q_b' of Q_b is Q_b without the last three relation symbols. Since Q_b' is not hierarchical, Q_b is not head-hierarchical. □

C. Query Evaluation

Semantically, conjunctive queries are evaluated in each world. Given a query Q and a probabilistic database D, the probability of a distinct answer t is the probability of t being in the result of Q in the worlds of D, or equivalently,

$$\Pr[t \in Q(D)] = \sum_{\mathcal{A} \text{ is a world of D: } t \in Q(\mathcal{A})} \Pr[\mathcal{A}].$$

The semantics does not suggest a practical approach to query evaluation, since the number of worlds is exponential in the number of random variables. The evaluation of queries in probabilistic databases has two conceptually distinct steps [26]: (1) the computation of answers, and (2) the computation of probabilities of distinct answers. One widely adopted approach to (1), which is also considered in this paper, is to evaluate queries following the standard semantics, where additionally the events of input tuples are copied along in answer tuples, see Fig. 1 for an example. For step (2), we use the well-known correspondence between the probabilities of a tuple t in the query answer $Q(D)$and its event $t.E$: For any query Q, probabilistic database D, and tuple $t \in Q(D)$, it holds that $\Pr[t \in Q(D)] = P(t.E)$.

The result of a top-k query (Q, k) is a set of k most probable answers to the conjunctive query Q. The result of a rank-k query (Q, k) is the same as top-k where in addition the answers in the top-k set are ordered by their probabilities.

III. Share Plans for Ranking and Probability Computation

Our ranking algorithm has two steps. Given a conjunctive query Q, we first derive a share plan for Q, as explained in this section. These plans are useful for (exact and approximate) probability computation since they expose subqueries whose probability computation can be shared across several query answers. For ranking, the plan of the head-restriction of Q is already sufficient, as discussed later in Section V. Share plans can also be viewed as recipes for factorisations of the events of query answers that enable at runtime easy identification of factors that are common to such events.

The second step of our algorithm, given in Section IV, takes as input the query answers together with their factorised events and incrementally approximates the probabilities of the answers and of their shared events until the desired ranking is obtained. Both steps work for databases with arbitrary correlations but we also discuss simplifications due to the independence assumption for input database tuples.

A share plan for a query Q is a forest with one tree for each connected component in the graph of Q. It has two types of inner nodes, namely *root variables* and *split variables*, and leaves that represent (possibly singleton) conjunctions of relation symbols from the query body. We depict in a plan a variable Y as $\boxed{Y}$ if it is root and as (Y) if it is split.

The variables $\bar{Y}$ are root in Q if they occur in all relation symbols of Q. In this case, the events of the query answers can be factorised as $\bigvee_{\bar{a}} Q[\bar{Y}/\bar{a}]$, where the event of $Q[\bar{Y}/\bar{a}]$ is obtained by evaluating Q under the assignment $\bar{Y}/\bar{a}$ and

plan (Query Q)

if Q is one relation symbol **then return** Q;
if $Q = Q_1, \ldots, Q_n$ s.t. $\forall i, j : Q_i, Q_j$ are disconnected **then**
  return plan$(Q_1), \ldots,$ **plan**(Q_n);
if Q has a maximal set $\bar{Y}$of root variables **then**
  let $Q' = Q$ where all occurrences of $\bar{Y}$ are replaced by $*$
  in return $\boxed{\bar{Y}}$**(plan**$(Q'))$;
if $\exists$ variables $\bar{Y}$ such that $Q = Q_+(\bar{Y}), Q_-(\bar{Y})$ **and**
  Q_+ contains all relation symbols with Q's head variables **and**
  Q_- has root variables $\bar{Y}$ and several relation symbols **then**
  return $(\bar{Y})$**(plan**(Q_+), **plan**$(Q_-))$;
return Q;

Fig. 2: Construction of share plans for conjunctive queries.

can be expressed as a conjunction of events of the subqueries representing connected components of the graph of $Q[\bar{Y}/\bar{a}]$, where all edges induced by $\bar{Y}$ are removed. Fig. 1 shows event factorisations induced by root variables for two queries.

The variables $\bar{Y}$ are split if Q can be rewritten as $Q(\bar{X}) \text{ :- } Q_+(\bar{X}, \bar{Y}), Q_-(\bar{Y})$, where $\bar{X}$ are the head variables of Q, Q_+ contains all relation symbols with $\bar{X}$, and Q_- has root variables $\bar{Y}$. Split variables generalise root variables: In case the split variables $\bar{Y}$ are also root in Q_+, then $\bar{Y}$ would be root in the entire query Q. Under an assignment $\bar{Y}/\bar{a}$ for the split variables $\bar{Y}$, the event of $Q_-[\bar{Y}/\bar{a}]$ is paired with events of Q's answers that have the same $\bar{Y}$-value. Split variables are useful for factorising events of non-hierarchical queries, for which root variables are not always possible.

Example 3.1: We give two share plans in the introduction. Both plans have a root variable Y. The plan in Fig. 1 has a relation symbol as the left leaf and a conjunction of relation symbols as the right leaf. Both leaves represent the subqueries Trends(X, Y) and $Q_3(Y)$ that are joined on Y. This plan can be interpreted as follows: For a given Y-value a, the events of several X-values (i.e., query answers) can be paired with the same event of the subquery $Q_3[Y/a]$. This factorisation of events is shown in Fig. 1 for a particular input database.

Consider now the bottom-most share plan in Fig. 3; this corresponds to the query Q_c in Example 2.2. The topmost node is a split variable. It separates the left subquery Q_+, which contains the head variables X and Z of Q_c, and the right subquery Q_-, which has two variables Y and V that are root and head. Thus, this plan sees Q_c rewritten as follows: $Q_c(X, Z) \text{ :- } Q_+(X, Y, Z), Q_-(Y)$. It states that several query answers, i.e., (X, Z)-values, can be paired with the same Y-value a and thus share the event of $Q_-[Y/a]$. Both subqueries Q_+ and Q_- have additional root variables that are made explicit in their plans. □

The construction of a share plan for a conjunctive query Q is described in Fig. 2 by a recursive function **plan**. A relation symbol is a plan with one node. If Q can be partitioned into disconnected subqueries, then we return the forest of plans

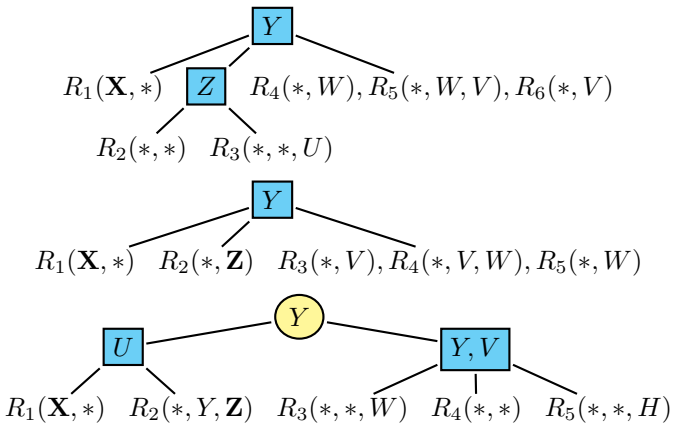


Fig. 3: Share plans for queries Q_a, Q'_b, and Q_c from Example 2.2 (top to bottom). The plan for Q_b (not shown) is the one for Q'_b, where in addition there is one plan node for the non-head component with relations symbols R_6, R_7, R_8.

with one plan for each subquery. If Q has a (non-empty and maximal) set $\bar{Y}$ of root variables, then we create a variable node for $\bar{Y}$ and recurse on a modified query, where $\bar{Y}$ are considered fixed to constants (the constant * is used here). This modification can allow to partition the query into disconnected subqueries. In case none of the above rules apply, we choose a split of the query into Q_+ and Q_-, as discussed above. There may be several splits, since Q_+ is not constrained to consist of the relation symbols with head variables only. The smaller Q_+ is, the larger Q_- and thus the events produced by Q_- and shared by Q_+'s answers. This can lead in practice to better performance for probability computation and ranking, since the shared events of Q_- tend to get larger and more complex with the increase in the size of Q_-. We rule out the case of Q_- consisting of one relation symbol; In our experiments, sharing is severely limited and not rewarding in this case.

Example 3.2: We exemplify the construction of share plans for the queries Q_a and Q_b from Example 2.2. Their plans are shown in Fig. 3. For Q_a, we first discover the root variable Y. Under a fixed value a for Y, $Q_a[Y/a]$ can be partitioned into three disconnected subqueries: $R_1(X,*)$, then $R_2(*,Z), R_3(*,Z,U)$, and $R_4(*,W), R_5(*,W,V), R_6(*,V)$. The first and third subqueries cannot be refined further; the second one has the root variable Z.

For Q_b, we first observe that it can be partitioned into two disconnected subqueries: the first five relation symbols denoted by Q'_b in Example 2.2, and the last three relation symbols. The second subquery cannot be decomposed further. For Q'_b, we find the root variable Y and under fixed values for Y, we obtain three disconnected subqueries. □

Our algorithm can construct share plans efficiently: At each recursion level, it needs polynomial time to check for disconnected components, root variables, and split variables, and it recurses further onto partitions of the query.

Proposition 3.3: For any conjunctive query, the algorithm in Fig. 2 constructs a share plan in polynomial time. □

A special case is that of hierarchical queries.

Proposition 3.4: For any hierarchical query, the algorithm in Fig. 2 constructs a share plan, where the inner nodes are root variable nodes and the leaves are relation symbols. □

In the case of non-repeating hierarchical queries, share plans become the safe plans proposed by Dalvi and Suciu [5].

We next discuss aspects concerning the use of share plans.

Event factorisation by share plans. Share plans can be seen as layering the original query into strata, where each layer corresponds to either root or split variables. The factorisation of the events of the query answers precisely follows this layering. If we execute the share plan bottom-up, we compute both the query answers and the factorisation of their events. Recall, for instance, the share plans of the queries Q_1 and Q_2 from the introduction. We first evaluate its subqueries at the leaves of the plan. Then for each value of the root variable Y, we create events that are conjunctions of events of the two subqueries. Fig. 1 shows the events of the query answers factorised according to these plans.

A caveat of the above approach is that a share plan fixes, to some extent, the join ordering for query evaluation, although it does this solely for the purpose of event factorisation. Earlier work [17] has shown that the safe plans for non-repeating hierarchical queries can dramatically limit the performance of query evaluation by fixing the join orders so as to obtain optimal factorisation of query events. It has been argued that the structure of safe plans should only be used for the purpose of event factorisation and be decoupled from the computation of query answers, for which standard query plans can do a much better job. This approach is also applicable here, but we do not detail it due to space limitation.

The case of tuple-independent databases. The factorisations described by share plans apply to databases with arbitrarily correlated tuples. Under the strong assumption of tuple-independent databases, share plans for non-repeating conjunctive queries become even more effective. In particular, the events of two disconnected queries Q_1 and Q_2 become independent and $P(Q_1 \wedge Q_2) = P(Q_1) \cdot P(Q_2)$. Also, under different assignments $\bar{Y}/\bar{a}_1$ and $\bar{Y}/\bar{a}_2$ of the root variables $\bar{Y}$ of a query Q, the queries $Q_1 = Q[\bar{Y}/\bar{a}_1]$ and $Q_2 = Q[\bar{Y}/\bar{a}_2]$ have independent events and $P(Q_1 \vee Q_2) = 1-(1-P(Q_1)\cdot(1-P(Q_2))$. These simplifications dramatically improve probability computation and ranking. Earlier work discusses the connection between read-once factorisations, where each variable occurs at most once, and tractability of query processing in probabilistic databases [16], [24], [26].

IV. Incremental Approximate Probability Computation for Events with Shared Factors

In this section, we present the second ingredient of our ranking technique: approximate probability computation for events of query answers. We first discuss a method to compute probabilities of events by decomposing them, and then show how to use this method to rank events. Our approach is based on earlier work on probability computation by event decomposition in the SPROUT query engine [18], [10], and on top-k probability computation [22].

A. Probability Computation by Event Decomposition

In earlier work [18], [10] we have introduced an incremental algorithm for exact and approximate probability computation of events. This approach is based on incremental decomposition of an event Φ using three types of decomposition:

1) Independent-or: Partition Φ into independent events Φ_1, Φ_2 such that Φ is equivalent to $\Phi_1 \vee \Phi_2$.
2) Independent-and: Partition Φ into independent events Φ_1, Φ_2 such that Φ is equivalent to $\Phi_1 \wedge \Phi_2$.
3) Exclusive-or: Choose a variable x in Φ. Then, Φ is equivalent to $x \wedge \Phi|_x \vee \neg x \wedge \Phi|_{\neg x}$, where the event $\Phi|_\alpha$ is obtained from Φ by setting α to true. This is called Shannon expansion.

These decompositions preserve equivalence, can be performed in polynomial time in case the events Φ_1 and Φ_2 can be obtained by efficient manipulations of Φ, and can be used to efficiently compute probabilities of known tractable queries.

A *decomposition tree*, or d-tree, of an event Φ is obtained by recursively decomposing Φ in the order given above. The inner nodes are decomposition types, and the leaves are events. A complete d-tree is obtained by applying the decomposition rules until we reach clauses or simple literals at leaves.

The probability of an event can be computed in one traversal of any of its d-trees, provided the probabilities at leaves are known. This is because the probabilities at the inner nodes can be computed efficiently. Let P be a function that maps events to their probabilities. We then have:

$$P(\text{independent-or}(\Phi_1, \Phi_2)) = 1 - [1 - P(\Phi_1)] \cdot [1 - P(\Phi_2)]$$
$$P(\text{independent-and}(\Phi_1, \Phi_2)) = P(\Phi_1) \cdot P(\Phi_2)$$
$$P(\text{exclusive-or}(\Phi_1, \Phi_2)) = P(\Phi_1) + P(\Phi_2).$$

Example 4.1: The event $\Phi = x_1 \wedge y_1 \vee x_1 \wedge y_2 \vee \neg x_1 \wedge y_2$ can be decomposed using Shannon expansion into $x_1 \wedge (y_1 \vee y_2)$ and $\neg x_1 \wedge y_2$. The first term can be decomposed into independent parts x_1 and $y_1 \vee y_2$, with the former decomposed into independent parts y_1 and y_2. Then, we have: $P(\Phi) = P(x_1) \cdot [1 - (1 - P(y_1)) \cdot (1 - P(y_2))] + (1 - P(x_1)) \cdot P(y_2)$.□

D-trees can also be used to incrementally compute approximate probabilities. Assume we are constructing a d-tree for an event Φ, and that after every single decomposition step, we are given lower and upper bounds on the probabilities of the leaves. Then we can compute lower and upper bounds on the probability of the event Φ as follows: for the lower (upper) bound, assume that the probabilities at the leaves are their lower (upper) bounds and apply the recursive computation given above by the function P. The probability at the root of the d-tree represents the lower (upper) bound on the probability of Φ. By further decomposing the event, the bounds become tighter and eventually converge to the exact probability of Φ.

A crucial aspect of approximation based on d-trees is that at any time the current bounds are included in the bounds computed at the previous step. Moreover, this property holds regardless of the algorithm used to compute bounds at the leaves. To see this, assume that the lower and upper bounds on the probability of Φ, as obtained by the algorithm sketched above, are $[L_0, U_0]$ before a decomposition step and $[L_1, U_1]$ after that step. If $[L_1, U_1] \not\subseteq [L_0, U_0]$, we can then obtain a new pair $[L, U]$ of correct bounds, where $L = \max(L_0, L_1) \leq U = \min(U_0, U_1)$. Thus $[L, U] \subseteq [L_0, U_0]$ and $[L, U] \subseteq [L_1, U_1]$. This bounds-monotonicity property makes our approach truly incremental, yet it does not hold for the state-of-the-art ranking approach based on Monte Carlo simulations [26].

A question remains: how to efficiently compute lower and upper bounds at the leaves? There are several existing solutions to this [18], [9], [10], [11], [26]. Most of them need at most quadratic time in the size of the event and compute model-based bounds: Given an event Φ, lower and upper bound events Φ_L and Φ_U respectively are such that the satisfying assignments of Φ_L are also satisfying assignments of Φ, which in turn are satisfying assignments of Φ_U. In addition, these bounds are optimal with respect to an event language $\mathcal{L}$, such as the language of monotone DNF events, if there are no events Φ'_L and Φ'_U in $\mathcal{L}$ that are lower and respectively upper bounds of Φ and that Φ_L is a strict lower bound of Φ'_L and Φ_U is a strict upper bound of Φ'_U. Some bounds apply to monotone DNF events only [18], [26], whereas others apply to arbitrary events [9]. The decomposition-based approximation scheme we consider here works with any of these solutions. In our prototype, we used a combination of the ones in [18], [10]. In the sequel, we assume that we are given an algorithm called **bounds** to compute probability bounds at the leaves.

B. Decomposition of Events with Shared Factors

Decomposing events that have shared factors is naturally supported by d-trees (and in fact by any knowledge compilation technique such as BDDs or d-DNNFs).

Assume we are given the query answers and their events factorised according to a share plan. We can identify at runtime syntactically equal factors by scanning these events. The main idea of handling events with shared factors is as follows. We first enable sharing by naming common factors and replacing all of their occurrences by their names. We can now decompose the events as if the common factors would be single variables until we reach leaves holding these variables only. Those leaves become now pointers to the roots of the d-trees of the common factors, hence the common factors are only decomposed once and used by all events that contain them. We then continue the decomposition in the d-trees of those factors. In case of approximate probability computation, the only additional difficulty is computing lower and upper bounds at the leaves after each decomposition step. This has now to consider common factors and, if possible, only compute their bounds once for all sharing events. Although this is the general picture, there are additional technical aspects and restrictions that we will discuss after looking at an example.

Example 4.2: Let Ψ be the underlined factor that is common to all Q_1's events in Fig. 1. By naming it, we obtain:

$$\Psi = y_4 z_1 u_1 \vee y_4 z_3 u_3 \vee y_5 z_4 u_1$$
$$\Phi_1 = x_1(y_1 z_2 u_2 \vee y_2 z_5 u_2) \vee x_2 \underline{\Psi}$$
$$\Phi_2 = x_3 y_3 z_6 u_4 \vee x_4 \underline{\Psi}$$

Since Ψ is independent of the other parts of the events Φ_1 to Φ_4, we can decompose it separately from the rest. The probabilities of these events are as follows:

$$P(\Phi_1) = 1 - [1 - P(x_1) \cdot P(u_2) \cdot (1 - (1 - P(y_1 z_2)) \cdot (1 - P(y_2 z_5)))] \cdot [1 - P(x_2) \cdot P(\Psi)]$$
$$P(\Phi_2) = 1 - (1 - P(x_3 y_3 z_6 u_4))(1 - P(x_4) \cdot P(\Psi))$$

The above expressions only use independent-or and independent-and decompositions. To decompose Ψ, however, we need Shannon expansion. The probability $P(\Psi)$ is then:

$$P(y_4) \cdot P((z_1 \vee y_5 z_4) u_1 \vee z_3 u_3) + (1 - P(y_4)) \cdot P(y_5 z_4 u_1).$$

The remaining folded sub-events can now be decomposed using independent-and and independent-or. □

The above example shows a simplified case, where the common factor is independent of the rest. This always holds for events of non-repeating conjunctive queries on tuple-independent databases, since such factors represent events of subqueries over relations that do not occur in other subqueries. In case of repeating queries on databases with arbitrary correlations, this may not hold in general since any two input events may share variables. In particular, the root query variables cannot help us infer independence of sub-events and disconnected queries do not necessarily have independent events. In this case, the treatment of common factors during decomposition becomes slightly more involved.

Assume again that several events share a factor Ψ, yet Ψ is not independent of them. The decompositions of the events happen as before, yet when a leaf with Ψ is reached, the actual truth value at that leaf is not that of Ψ alone, but is instead that of Ψ *conditioned* by all decisions on the truth of variables made by Shannon expansion steps on the path from that leaf to the root of the d-tree. That is, although Ψ can be decomposed once for all events, its d-tree d does not contribute in equal amounts to all these events and their leaves. To compute the contribution of d to a particular leaf, we need one pass over d and only consider the probabilities of those branches that do not contradict the decisions at that leaf.

Example 4.3: Consider the events Φ_1 and Φ_2 that share Ψ:

$$\Psi = x_1 x_2 \vee x_1 x_3 \vee \neg x_2 x_3$$
$$\Phi_1 = x_2 \underline{\Psi}$$
$$\Phi_2 = (x_1 \vee x_3) \underline{\Psi}$$

A d-tree d for Ψ can be obtained by Shannon expansion on x_2, after which we are left with two leaves: $\Psi|_{\neg x_2} = x_3$ and $\Psi|_{x_2} = x_1$. Then $P(\Psi) = P(\neg x_2) \cdot P(x_3) + P(x_2) \cdot P(x_1)$.

A d-tree d_1 for Φ_1 has one Shannon expansion node for x_2 and two children: Ψ in case x_2 = true, and false otherwise. In the first case, only one branch ($x_2 x_1$) in d satisfies x_2 = true and hence the probability of the leaf Ψ in d_1 is $P(x_1)$. The probability of d_1 is then $P(x_2 x_1)$.

For Φ_2, there are two leaves Ψ in a d-tree d_2 reachable via the paths $p_1 = x_1$ and $p_2 = \neg x_1 x_3$. Under p_1, both branches in d are satisfied and thus the probability at the leaf Ψ in d_2 is $P(\neg x_2) \cdot P(x_3) + P(x_2)$. Under p_2, no branch in d is satisfied, thus the probability at the leaf Ψ in d_2 is 0. As for d, the probability of d_2 is $P(x_1) \cdot (P(\neg x_2) \cdot P(x_3) + P(x_2))$. Indeed, Ψ and Φ_2 are equivalent events. □

We thus compute a d-tree representing the composition of d-trees for events Φ_i and Ψ. A similar technique is known for OBDDs [3]: Computing an OBDD for the composition of Boolean functions f_1 and f_2, where a variable x from f_1 is replaced by f_2, can be done in polynomial time in the size of the OBDDs for f_1 and f_2.

top (Integer k, Events $e_1, \ldots, e_n$)
let $D_i =$ partial d-tree for event $e_i, \forall 1 \leq i \leq n$;
let critical region $[L, U] = [0, 1]$;
while $L \leq U$ **do**
  let $[L_i, U_i] =$ **bounds** $(D_i), \forall 1 \leq i \leq n$;
  compute order π_l s.t. $L_{\pi_l(1)} \geq \cdots \geq L_{\pi_l(n)}$;
  compute order π_u s.t. $U_{\pi_u(1)} \geq \cdots \geq U_{\pi_u(n)}$;
  let critical region $[L, U] = [L_{\pi_l(k)}, U_{\pi_u(k+1)}]$;
  foreach $1 \leq i \leq n$ **do**
    if $[L, U] \cap [L_i, U_i] \neq \emptyset$ **then**
      decompose_until_progress_made (D_i);
end-while
return $\{e_{\pi_l(1)}, \ldots, e_{\pi_l(k)}\}$

Fig. 4: Computing the set of top-k most probable events. To compute rank-k, we run top-k, then top-$(k-1)$ on the top-k set and so on.

C. Ranking Algorithm

Our ranking algorithm is described in Fig. 4. The idea is to approximate the probabilities of the events of the query answers to the extent needed to separate the top-k most probable events from the others. It achieves this by incrementally refining the lower and upper bounds of the events until there are k lower bounds that are greater than the upper bounds of the other events. To compute approximate probabilities of the events, we can use any of the two decomposition-based approaches from Sections IV-A and IV-B. In the former case, the events are decomposed separately from each other, whereas in the latter case, we use share plans to exploit common factors. In the ranking algorithm, event decomposition is done by a call to the algorithm **decompose_until_progress_made**. The algorithm **bounds** computes probability bounds of d-trees for events efficiently.

For each event e_i, we keep a partially constructed d-tree D_i together with its probability bounds $[L_i, U_i]$. In case there are no k lower bounds greater than the upper bounds of the other events, we further decompose the events in order to tighten their bounds. Although decomposing all events is certainly correct, it is not efficient in general, especially when we already know that (a) some events have already made it in the top-$k' < k$ (their lower bounds are larger than the upper bounds of the other $n - k'$ events), and (b) some other events will never make it in the top-k (their upper bounds are lower than the lower bounds of at least k other tuples). It is therefore desirable to only spend time on those events for which it is

not yet clear to which category they belong. This defines the *critical region* [22]: this is the interval $[L, U]$ where L is the k-th highest lower bound, and U is the $k+1$-st highest upper bound. All events, whose bounds intersect the critical region, are further decomposed until their bounds get tightened. As argued in Section IV-A, each decomposition step does not widen the bounds interval.

The process of ordering of events by their probability bounds and of checking these bounds against the critical region denotes a *separation* step. We found experimentally that separation steps are more expensive than decomposition steps in d-trees, especially when k is large or the events are rather small, and preferred to invest more time in decompositions than in separation steps.

Non-uniqueness of top-k. There are cases when the top-k set is not unique: this can happen when several events have equal probabilities. In such cases, the outcome of our algorithm depends on the order of events in the input.

From top-k to rank-k. The algorithm given in Fig. 4 only computes a set of top-k most probable events. To rank them, we can then call top-$(k-1)$ on this set, then top-$(k-2)$, and so on. To improve performance, the d-trees of the top-k events are kept between the runs so that event decomposition resumes from the previous round.

Shared events for probability computation and ranking. There is a subtle, yet significant difference in the way sharing can be used for (exact or approximate) probability computation versus ranking. For probability computation, factors common to several events, as exposed by share plans, can be potentially decomposed once for all these events. For ranking, this decomposition is not exhaustive but performed as much as needed to separate the probability bounds of answer events, as discussed above. In addition, only the tree of the share plan representing the head component is relevant for ranking, the remaining trees can be safely discarded. This argument is followed in more detail in Section V.

V. Dichotomy for Ranking Query Answers

We study the following ranking problem: Given a conjunctive query Q, a tuple-independent probabilistic database D, and any two answers $t_1, t_2 \in Q(D)$, does $P(t_1) \leq P(t_2)$ hold? This problem is trivial for Boolean conjunctive queries, since their answer is top-1 regardless of its probability (which is non-zero since the variables have non-zero probabilities). In this section, we show that the head-hierarchical queries are precisely those non-repeating non-Boolean conjunctive queries for which ranking can be decided in polynomial time. This result settles an open problem raised by Dalvi and Suciu [6].

A. The ranking problem is in PP

The complexity class PP is the set of decision problems that can be solved by a nondeterministic polynomial-time Turing machine where the acceptance condition is that at least half of computation paths accept. MAJ SAT is a natural PP-complete problem [25], [12]: Given a formula Φ and a positive integer i, does Φ have at least i satisfying assignments? A simpler, yet still PP-complete, version of this problem asks whether at least half of the assignments of Φ are satisfying. We will use both versions in our reductions.

PP is tightly connected to the class #P of functions that count the number of accepting paths of nondeterministic polynomial-time Turing machines [2]: $\mathrm{P}^{\mathrm{PP}} = \mathrm{P}^{\#\mathrm{P}}$. This means that the class of decision problems computable in polynomial time using #P oracles coincides with the class of decision problems computable in polynomial time using PP oracles. Thus, PP-complete problems are decision problems that capture the inherent computational complexity of #P-complete problems.

Membership in PP. The ranking problem is in PP by the following argument[1]. Let F and G be the events of two answers t_1 and t_2 respectively. We thus have that $P(t_1) = P(F)$ and $P(t_2) = P(G)$. In case of conjunctive queries, the events F and G are positive DNF formulas; the following argument holds however for arbitrary events.

The problem is to check whether $P(F) \leq P(G)$. Define the disjoint events $A = F \wedge \neg G$ and $B = G \wedge \neg F$. Then,

$$P(F) \leq P(G) \Leftrightarrow P(A) \leq P(B), \text{ since}$$
$$P(A) = P(F) + P(\neg G) - P(F \vee \neg G)$$
$$= P(F) - P(G) + P(G \wedge \neg F) = P(F) - P(G) + P(B).$$

Now denote $A' = A \vee \neg A \neg B$ and $B' = B \vee \neg A \neg B$. Clearly, $A' \vee B' = \text{true}$. Then,

$$P(A') = P(A) + P(\neg A \neg B). \quad P(B') = P(B) + P(\neg A \neg B)$$
$$\text{Hence, } P(A) \leq P(B) \Leftrightarrow P(A') \leq P(B') \Leftrightarrow P(B') \geq 1/2.$$

In case of uniform probability distributions, the latter inequality is the PP-complete problem MAJ SAT for arbitrary formulas B'. In case of non-uniform probability distributions, the same argument used for membership in #P of the probability computation problem can be made here [26](page 47).

B. The Dichotomy

Our syntactic characterisation of queries for which ranking is tractable with respect to data complexity is given next.

Theorem 5.1: Fix a non-repeating non-Boolean conjunctive query Q.

- If Q is head-hierarchical, then, for any tuple-independent database, the ranking problem is in polynomial time.
- If Q is not head-hierarchical, then the ranking problem is #P-hard. □

A corollary of Theorem 5.1 is that either ranking all answers can be done in polynomial time, or finding the top-1 (most probable) answer is #P-hard.

For the tractability result, we note that if Q is head-hierarchical, then its head-restriction is hierarchical by Definition 2.1. The query Q has then the following pattern: $Q(\bar{X}) \text{ :- } Q_1(\bar{Y}), Q_2(\bar{Z})$, where $\bar{X} \subseteq \bar{Y}$, $\bar{Y} \cap \bar{Z} = \emptyset$, the head-component Q_1 is hierarchical, and Q_2 can be hierarchical or non-hierarchical. We next show that to rank Q's answers, it suffices to compute the exact probabilities of Q_1's answers. Since Q_1 is hierarchical, this can be done in polynomial time by known algorithms using safe plans [5] and OBDDs [16].

[1]Personal communication with Dan Suciu, July 2011.

Consider two answers a_1 and a_2. Then,

$$P(Q[\bar{X}/\bar{a}_i]) = P(Q_1[\bar{X}/\bar{a}_i], Q_2(\bar{Z})) = P(Q_1[\bar{X}/\bar{a}_i]) \cdot P(Q_2(\bar{Z})).$$
$$P(Q[\bar{X}/\bar{a}_1]) \leq P(Q[\bar{X}/\bar{a}_2]) \Leftrightarrow$$
$$P(Q_1[\bar{X}/\bar{a}_1]) \cdot P(Q_2(\bar{Z})) \leq P(Q_1[\bar{X}/\bar{a}_2]) \cdot P(Q_2(\bar{Z})) \Leftrightarrow$$
$$P(Q_1[\bar{X}/\bar{a}_1]) \leq P(Q_1[\bar{X}/\bar{a}_2]), \text{ since } P(Q_2(\bar{Z})) \geq 0.$$

The exact probabilities of Q_1's restrictions $P(Q_1[\bar{X}/\bar{a}_1])$ and $P(Q_1[\bar{X}/\bar{a}_2])$ can be computed in polynomial time.

For the hardness part, we first show that MAJ SAT for positive bipartite DNF formulas (MAJ-PP2DNF for short) is #P-hard under a polynomial-time Turing reduction from the #P-complete problem #SAT for positive bipartite DNF formulas (#PP2DNF for short). We then give a polynomial-time many-one reduction from MAJ-PP2DNF to ranking for any (non-Boolean) non-head-hierarchical query.

Reduction from #PP2DNF to MAJ-PP2DNF. We know that #PP2DNF is #P-complete [21], [26]. Assume we are given any PP2DNF Φ with n variables. Then, we can compute $\#\Phi$ using at most n calls to a MAJ-PP2DNF oracle for Φ. This means that MAJ-PP2DNF is at least as hard as the #PP2DNF under polynomial-time Turing reduction. We first ask MAJ-PP2DNF with $i = 2^{n-1}$. If true, then we ask with $i = 2^{n-1} + 2^{n-2}$; otherwise, $i = 2^{n-2}$. We thus do binary search to determine the right value $\#\Phi$. The decision tree of this binary search is exponential in n, yet has depth at most n. To find $\#\Phi$, we only need to construct exactly one of its paths.

Reduction from MAJ-PP2DNF to ranking for non-head-hierarchical queries. Let Φ be a PP2DNF, We would like to decide the #P-hard problem MAJ-PP2DNF for Φ where at least half of the assignments of Φ are satisfying. Equivalently, we would like to decide $P(\Phi) \geq 1/2$, where Φ is over n Boolean random variables with uniform probability distribution.

Consider any non-head-hierarchical query $Q(\bar{X})$ and two (distinct) answers $\bar{a}_1$ and $\bar{a}_2$. Then, the Boolean queries $Q[\bar{X}/\bar{a}_1]$ and $Q[\bar{X}/\bar{a}_2]$ are not head-hierarchical, and in particular not hierarchical. Given Φ, we create a tuple-independent database D with random variables being the n variables of Φ. Most importantly, D is such that (1) the event of $Q[\bar{X}/\bar{a}_1]$ is precisely Φ and (2) the event of $Q[\bar{X}/\bar{a}_2]$ is one clause which is independent of Φ and for which the probability is 1/2. The ranking problem becomes then $P(\Phi) \geq 1/2$. The database D is $(R_1^1 \cup R_1^2, \ldots, R_m^1 \cup R_m^2)$, which consists of one database $D^i = (R_1^i, \ldots, R_m^i)$ for each of the two steps such that $\forall 1 \leq j \leq m : R_j^1 \cap R_j^2 = \emptyset$.

The construction of the database D^1 is exactly as in the hardness proof of query evaluation for non-hierarchical queries [5], [26] and is skipped. The second construction step is such that, for any query Q, we construct each of the relations $R_1^2, \ldots, R_m^2$ referred in the query to consist of a single tuple associated with fresh random variables x_1^2 to x_m^2 such that exactly one of them has probability 1/2 and the others have probability 1. The database D^2 is only used to generate the answer a_2, whose event $x_1^2 \wedge \ldots \wedge x_m^2$ has probability 1/2.

Ranking vs. Query Evaluation. The hardness of ranking is not the same as hardness of query evaluation, which asks for exact probabilities of query answers. It is known that the *hierarchical* queries are those non-repeating conjunctive queries that can be evaluated in polynomial time [5]. All hierarchical queries are head-hierarchical, but not all head-hierarchical queries are hierarchical. In terms of complexity, there are (head-hierarchical) queries that are hard for query evaluation (non-hierarchical ones), but still tractable for ranking. Consider in the argument following Theorem 5.1 that Q_2 is non-hierarchical. Then, Q is not hierarchical and thus hard for query evaluation, but remains head-hierarchical and thus tractable for ranking.

Query	scale 0.1	scale 0.5	scale 1
Q_6	(11618, 11618, 7)	(57301, 57301, 7)	(114159, 114159, 7)
Q_{15}	(2460, 1651, 7)	(11935, 11935, 7)	(24047, 16001, 7)
Q_{16}	(3095, 2476, 50)	(16015, 12812, 50)	(32280, 25824, 50)
Q_{17}	(14893, 14420, 50)	(77817, 75302, 50)	(156305, 151262, 50)
Q_2	(2958, 1601, 5)	(14557, 7807, 5)	(29522, 15859, 5)
Q_9	(72419, 34694, 25)	(365780, 175219, 25)	(728040, 348759, 25)
Q_{20}	(2729, 2925, 5)	(32473, 16627, 5)	(63568, 32548, 5)

Fig. 5: Characteristics of TPC-H query answers. Each table entry represents (#variables,#clauses,#tuples) for the answer of the corresponding query and data scale factor.

Query/Data	#answers	#vars	#clauses	$N \cdot f/s$	avg(f)
$Q_a/1$	99	19876	34200	143	2.879
$Q_a/6$	99	1296	32880	7	2.768
$Q_a/12$	100	737	35160	3	2.930
$Q_b/1$	297	8358	98154	144	2.899
$Q_b/6$	297	677	115092	7	2.808
$Q_b/12$	300	463	79110	3	2.930

Fig. 6: Characteristics of query answers on synthetic data sets 1, 6, and 12. While the number of answers and clauses are kept in the same range, the number of variables decreases and the sharing factor increases when going from data set 1 to 12.

VI. Experimental Evaluation

Setup: The experiments were performed on an Intel Quad Processor Q8300 64bit/3.7GB/Linux 2.6.38/gcc 4.5.2. We report wall-clock execution times of queries run with a warm cache obtained by running a query once and then reporting the average runtime over five subsequent, identical executions with query answers and their events materialised to disk.

Algorithms: We experimentally evaluated ranking techniques on two specific problems: (1) compute the top-k most probable query answers (**top**), and (2) in addition to (1), order the top-k answers by probabilities (**rank**). We considered two flavours of the approach described in this paper:

shared: This is our technique based on share plans and approximate probability computation.

plain: This is a vanilla version of **shared** without share plans, and hence the incremental approximate computation is done independently for each query answer.

Both flavours are implemented within the SPROUT query engine, which extends the PostgreSQL 8.3.3 backend.

We compare our technique with several existing techniques.

KL: This algorithm is based on the state-of-the-art multi-simulation top-k algorithm [22], where the core Monte Carlo algorithm is replaced by an improved variant which is a

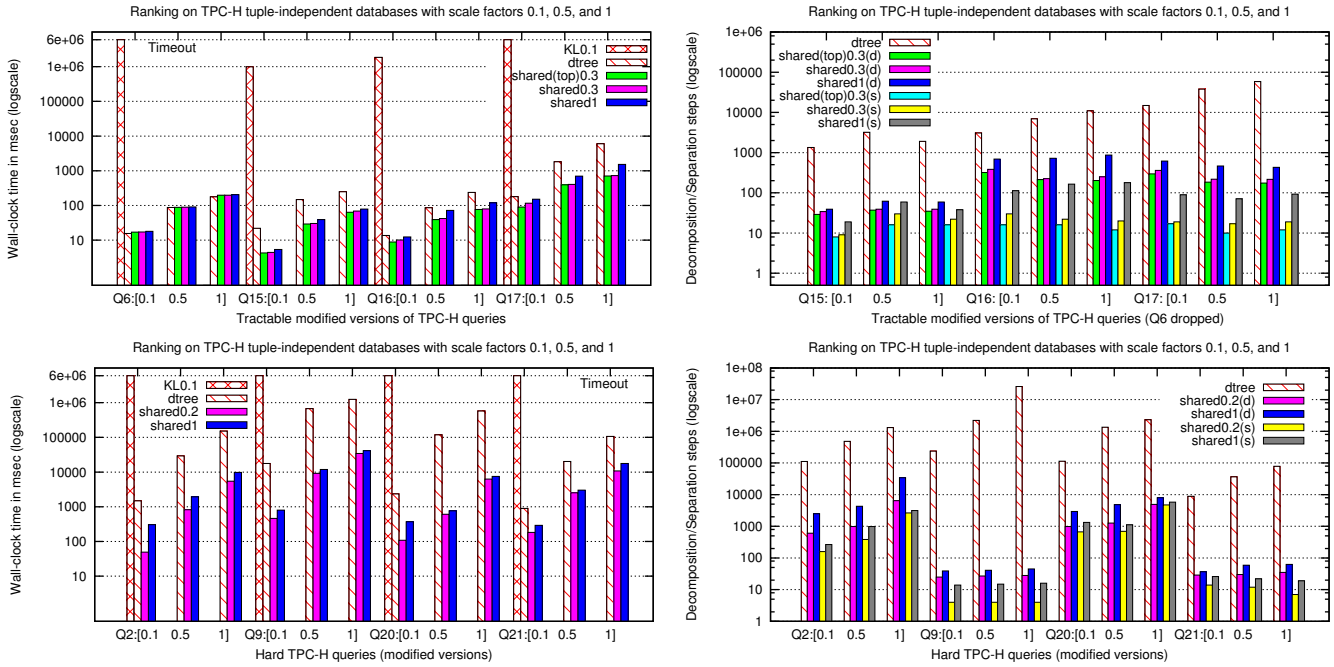


Fig. 7: Ranking in the TPC-H scenario: Comparison of performance and number of separation/decomposition steps. In the legend: 0.2, 0.3, and 1 mean that 20%, 30%, and respectively 100% of the top most probable answers are identified and ranked amongst themselves; (d) and (s) stand for decomposition steps and separation steps respectively.

combination of the Karp-Luby FPRAS unbiased estimator for DNF counting [27] adapted to probability computation and the Dagum-Karp-Luby-Ross optimal algorithm for Monte Carlo estimation [4]. This core algorithm is available in MayBMS 2.1. The multisimulation component of this approach is also implemented in MystiQ [22].

dtree: This is a state-of-the-art probability computation technique for relational algebra queries on probabilistic databases implemented in SPROUT [18], [10]. It incrementally decomposes the event of each answer using a knowledge compilation approach based on independence partitioning and Shannon expansion. Before each decomposition step, it checks whether the event is a DNF of pairwise independent clauses, in which case it computes the exact probability of the event and stops. We use **dtree** to compute the exact probabilities of *all* query answers and then sort them by their probabilities.

Experiment Design: The key observations underlying this work are that (1) ranking does not require the computation of exact probabilities of query answers, and that (2) the computation of approximate probabilities of query answers for the purpose of ranking can be shared across query answers. We experimentally confirm these observations using TPC-H and our own synthetic data sets as well as a set of queries that are either hard for ranking (and thus also for query evaluation), or easy for query evaluation (and thus also for ranking). In all these settings, we show that our ranking technique **shared** can be more than five orders of magnitude faster than **KL**, and up to two orders of magnitude faster than **dtree**. We explain this performance gap by analysing the amount of sharing and, where applicable, the amount of separation and decomposition steps needed for each of these techniques. The conclusion is that sharing computation across query answers, as exposed by share plans, in addition to incremental approximate probability computation, which is done as much as needed to ensure separation of tuples in the critical rank region, can be extremely effective.

Data sets and Queries. We consider two data sets: TPC-H and our own synthetic data. TPC-H databases are generated by a modified version of the TPC-H data generator (version 2.8), where each tuple is associated with a fresh Boolean variable with a random probability distribution [18], [10]. We consider scale factors 0.1, 0.5, and 1 (scale 1 means the database size is 1 GB on disk), and four tractable and four hard TPC-H queries, which are modified by dropping aggregations and by adding ranking. Query Q_6 is a selection on the large lineitem table, the other three tractable queries, namely Q_{15}, Q_{16}, Q_{17}, are joins of two tables. The hard queries Q_2, Q_9, Q_{20}, and Q_{21} are joins on four to six relations. Fig. 5 gives characteristics of the events of these queries. Their events are in the critical region of variable-to-clause ratios for which probability computation is hard in general [18].

To study the effect of an increase in the degree of sharing across answers on the performance of ranking, we evaluated the hard queries Q_a and Q_b discussed in Sections II-B and III on synthetic data. Fig. 6 gives characteristics of their events.

The data sets are generated as follows. Let $\mathrm{dom}(A)$ be the active domain of a query variable A. For $\bar{A} = (A_1, \ldots, A_n)$, we let $\mathrm{dom}(\bar{A}) = \overset{n}{\underset{i=1}{\times}} \mathrm{dom}(A_i)$. For both queries Q_a and Q_b, let Y be the split variable, $\bar{X}$ be the head variables, and

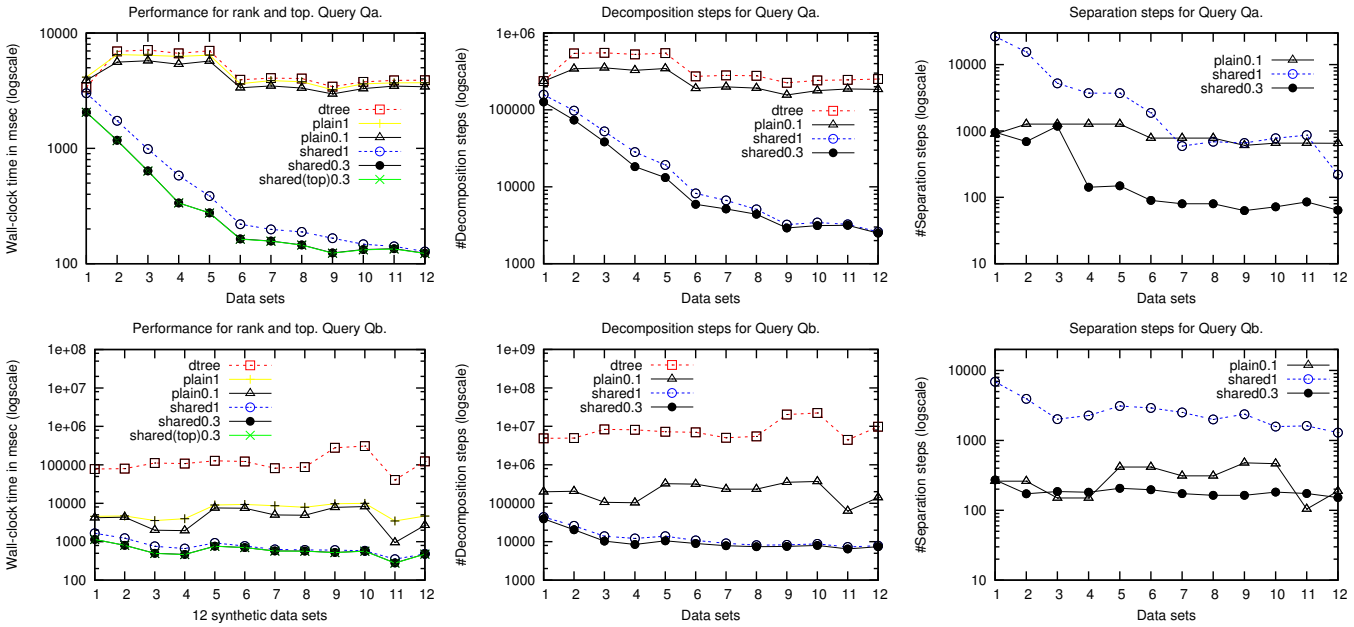


Fig. 8: Ranking in the synthetic data scenario: Comparison of performance and number of separation/decomposition steps. In the legend: 0.2, 0.3, and 1 mean that 20%, 30%, and respectively 100% of the top most probable answers are identified and ranked amongst themselves; (d) and (s) stand for decomposition steps and separation steps respectively.

$\bar{V}$ be the set of remaining query variables. We define the following parameters: f is the number of distinct Y-values that can be paired with the same $\bar{X}$-value, s is the number of distinct $\bar{X}$-values that can be paired with the same Y-value. We let $|\text{dom}(\bar{X})| = N$, where N is an input parameter, and $|\text{dom}(Y)| = N \cdot f/s$. We generate relations $R_i(\bar{X}, Y, \bar{V})$ as follows. If neither $\bar{X}$ nor Y are empty, we first draw N random values from $\text{dom}(\bar{X})$, and for each distinct $\bar{X}$-value, we randomly draw f values from $\text{dom}(Y)$ to pair with it. Then for each distinct $(\bar{X}, Y)$-value, we draw one random $\bar{V}$-value from its active domain. If $\bar{X}$ is empty, then we first draw $|\text{dom}(Y)|$ random Y-values, and for each distinct value we draw a number of c random $\bar{V}$-values. We fixed $N = 100$, $f = 3$, and $c = 50$ and obtained 12 data sets by varying s over 2, 5, 10, 20, 30 ,40, 50, 60, 70, 80, 90, 100. We thus decrease the domain of Y-values and increase the likelihood that more $\bar{X}$-values are paired with the same Y-value, and with this also the likelihood of sharing across query answers.

Experiment 1: Ranking for Tractable TPC-H Queries. Fig. 7 compares the performance of **KL**, **dtree**, and **shared** for the four tractable TPC-H queries. In all but two cases, **KL** does not finish to rank the top 10% answers for the smallest scale factor 0.1 within the allocated time of 600 minutes, which is up to five orders of magnitude larger than the time used by the other two techniques. **dtree** and **shared** have comparable performance. The events of Q_6 are independent sums of independent variables, and both **dtree** and **shared** detect this and avoid decomposition; **shared** needs slightly more time due to the overhead of sharing detection. For the other tractable queries, there is more sharing across answers and thus **shared** performs better than **dtree** even when all answers need to be ranked, cf. Experiment 4. The number of decomposition steps (Fig. 7 top right) for both approaches is less than the number of event variables, which confirms that both techniques are able to exploit the tractability of query evaluation. Moreover, **shared** needs at least tenfold less decomposition steps than **dtree** due to approximation and sharing; the separation steps needed by **shared**, but inexistent in case of **dtree**, use up most of the performance gain brought by the reduction in the number of decomposition steps of **shared**.

Experiment 2: Ranking for Hard TPC-H Queries. In contrast to tractable queries, for hard queries the performance gap between **dtree** and **shared** widens to at least one order of magnitude, cf. Fig. 7. This is because the events of hard queries are more complex and, on one hand, exact probability computation (**dtree**) can require exponentially many steps in the number of its variables, and on the other hand, **shared** employs approximate probability computation and exploits sharing across query answers. In case of Q_9, the difference in the number of decomposition steps becomes five orders of magnitude, the other queries witness a difference of up to two orders. **KL** did not finish for any query and scale factor and is only shown in the figure for scale 0.1.

Experiment 3: Ranking for Hard Queries on the Synthetic Data Sets. For $Q_a(X) \text{ :- } R_1(X, Y), Q_{a2}(Y)$, the performance gap between **dtree** and **shared** increases as we move from data set 1 to 12. This is explained by an increase in sharing across query answers; due to sharing, the large events corresponding to different Y-values of $Q_{a2}(Y)$ are only decomposed once for all answers. Fig. 8 shows this effect for time performance and number of decomposition steps. For Q_b, the relation between **dtree** and **shared** remains roughly the same for all 12 data sets,

which is about two orders of magnitude for both performance and number of decomposition steps.

Experiment 4: Ranking All Answers vs. Exact Probability Computation. Figs. 7 and 8 show that ranking *all* answers using our technique (**shared1**) is faster than exact probability computation followed by sorting (**dtree**) for both TPC-H and synthetic data sets. This is more evident in case of hard queries - with gains of up to two orders of magnitude, since for hard queries the probability of the events cannot be computed in general by a number of steps bounded in a polynomial of the number of event variables. This experiment clearly shows that for ranking we need not compute the exact probabilities, but incrementally approximate lower and upper bounds on the probabilities until we can separate them. The right-bottom plot in Fig. 7 shows that the difference in the number of decomposition steps between **shared1** and **dtree** can be up to five orders of magnitude for Q_9.

Moreover, even without sharing, our technique **plain** can perform much better than **dtree**, cf. Experiment 7.

Experiment 5: Ranking All Answers vs. Ranking A Few Top Answers. Figs. 7 and 8 suggest that in our experiments ranking all answers is about a factor of two more expensive than ranking the top 20% or 30% answers only, in terms of execution time and number of decomposition steps; the numbers of separation steps in the two cases witness a larger gap. This is because the process of ranking the top answers usually requires to work on the probability bounds of all answers in order to separate the top ones; after that, it is only a matter of finding the possibly many new critical regions of answers that require separation, and for each such region, a few decomposition steps would suffice to obtain separation.

Experiment 6: Ranking vs. Top-k. We compared the performance of our technique for the two problem flavours: **rank** and **top** in both TPC-H and synthetic data scenarios. For tractable TPC-H queries, the difference is rather insignificant: for Q_6 and Q_{15} this is because they only have seven answers; Q_{16} and Q_{17} have 50 answers and the number of decomposition and separation steps increase twofold when moving from computing the top-30% (i.e., 15) answers to also ranking these answers amongst themselves. Similar observations also apply to the synthetic data set, cf. Fig. 8.

Experiment 7: Effect of Sharing. To understand the impact of sharing on performance (time and number of steps), we compared **shared** against **plain** on the synthetic data sets. Fig. 8 shows that sharing accounts for most of the performance gain: Ranking all answers using sharing is up to two (one) orders of magnitude faster than ranking only the top 10% answers without sharing for the query Q_a (respectively Q_b). The difference in the number of decomposition steps needed by the two techniques reaches about two orders of magnitude for data set 12. Without sharing, **plain** is still faster than **dtree**. The speedup in case of Q_b is more than tenfold: Although both algorithms do not exploit sharing across answers, **plain** incrementally seeks approximate probabilities that are enough to rank the answers, whereas **dtree** needs to compute the exact probabilities of answers even if this is not always necessary.

VII. Conclusion

In this paper we introduce a technique for ranking query answers in probabilistic databases and discuss the complexity of the ranking problem. This technique is based on share plans and incremental approximate probability computation.

Acknowledgments

Olteanu's work was funded by the FP7 ERC grant FOX number FP7-ICT-233599 and EPSRC grant PrOQAW. Wen's work was funded by EPSRC EP/G069557/1 FRESNEL project. The authors would like to thank Robert Fink, Dan Suciu, and Martin Theobald for useful discussions and the reviewers for their helpful comments.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[2] D. Angluin. On counting problems and the polynomial-time hierarchy. *TCS*, 12, 1980.
[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8), 1986.
[4] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross. An optimal algorithm for monte carlo estimation. *SIAM J. Comput.*, **29**(5):1484–1496, 2000.
[5] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, **16**(4), 2007.
[6] N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *Proc. PODS*, 2007.
[7] N. N. Dalvi, C. Re, and D. Suciu. Queries and materialized views on probabilistic databases. *J. Comput. Syst. Sci.*, 77(3), 2011.
[8] A. Darwiche and P. Marquis. A knowlege compilation map. *Journal of AI Research*, 17:229–264, 2002.
[9] R. Fink and D. Olteanu. On the optimal approximation of queries using tractable propositional languages. In *ICDT*, 2011.
[10] R. Fink, D. Olteanu, and S. Rath. Providing support for full relational algebra in probabilistic databases. In *ICDE*, 2011.
[11] W. Gatterbauer and D. Suciu. Optimal upper and lower bounds for boolean expressions by dissociation. *CoRR*, abs/1105.2813, 2011.
[12] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.*, 6(4), 1977.
[13] I. F. Ilyas and M. A. Soliman. *Probabilistic Ranking Techniques in Relational Databases*. Morgan & Claypool, 2011.
[14] C. Koch and D. Olteanu. Conditioning probabilistic databases. *PVLDB*, 1(1), 2008.
[15] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *VLDB Journal*, 20(2), 2011.
[16] D. Olteanu and J. Huang. Using obdds for efficient query evaluation on probabilistic databases. In *Proc. SUM*, 2008.
[17] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *Proc. ICDE*, 2009.
[18] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation for probabilistic databases. In *Proc. ICDE*, 2010.
[19] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *Theory and Practice of Provenance (TaPP)*, 2011.
[20] D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. In *ICDT*, 2012.
[21] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4), 1983.
[22] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. ICDE*, 2007.
[23] P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. *VLDB*, 2008.
[24] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. In *VLDB*, 2010.
[25] J. Simon. *On some central problems in computational complexity*. PhD thesis, Cornell University, 1975.
[26] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probablistic Databases*. Morgan & Claypool, 2011.
[27] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.