The Dichotomy of Probabilistic Inference for Unions of Conjunctive Queries

Nilesh Dalvi
Yahoo!Research
and
Dan Suciu
University of Washington

We study the complexity of computing the probability of a query on a probabilistic database. The queries that we consider are unions of conjunctive queries, UCQ: equivalently, these are positive, existential First Order Logic sentences, or non-recursive datalog programs. The databases that we consider are tuple-independent. We prove the following dichotomy theorem. For every UCQ query, either its probability can be computed in polynomial time in the size of the database, or is hard for $FP^{\#P}$. Our result also has applications to the problem of computing the probability of positive, Boolean expressions, and establishes a dichotomy for such classes based on their structure. For the tractable case, we give a very simple algorithm that alternates between two steps: applying the inclusion/exclusion formula, and removing one existential variable. A key, and novel feature of this algorithm is that it avoids computing terms that cancel out in the inclusion/exclusion formula, in other words it only computes those terms whose Mobius function in an appropriate lattice is non-zero. We show that This simple feature is a key ingredient needed to ensure completeness. For the hardness proof, we give a reduction from the counting problem for positive, partitioned 2CNF, which is known to be #P-complete. The hardness proof is non-trivial, and uses techniques from logic and from classical algebra.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query Processing*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Mobius function, Mobius inversion formula, probabilistic database

1. INTRODUCTION

We study the complexity of computing the probability of a query on a probabilistic database. We are interested in identifying tractable cases, when the query probability can be computed in polynomial time in the size of the input databases, as well as identifying intractable cases, when computing the probability of the query is provably hard. Our work is motivated by probabilistic databases [Dalvi and Suciu 2004; Dalvi and Suciu 2007b; Olteanu et al. 2009; Olteanu and Huang 2009], as well as the probabilistic inference problem for Boolean expressions (which generalizes the model counting problem) [Creignou and Hermann 1996; Darwiche 2000; Darwiche and Marquis 2002; Wegener 2004; Golumbic et al. 2006].

A *probabilistic database* is a pair $\mathbf{D} = (D, P)$ where D is a database instance (i.e. a set of tuples) and $P : D \to [0, 1]$ associates a probability to each tuple $t \in D$. The probabilistic database defines a probability space, where the outcomes are the subsets W of D, and their probabilities are given by:

$$P_{\mathbf{D}}(W) = \prod_{t \in W} P(t) \times \prod_{t \in D-W} (1 - P(t)) \qquad (1)$$

Given a query Q (i.e. sentence in First Order Logic), $P(Q)$ is the *marginal* probability of Q, i.e. the probability that Q is true on a randomly chosen subset W of D: $P_{\mathbf{D}}(Q) = \sum_{W \subseteq D : W \models Q} P_{\mathbf{D}}(W)$.

The problem we address in this paper is the following. For a fixed query Q, what is the complexity of computing $P_{\mathbf{D}}(Q)$ as a function of the size of $\mathbf{D}$. We consider the data complexity only, where Q is fixed and the input consists only of $\mathbf{D}$. The queries that we study are those in the positive, existential fragment of FO, which are sentences that can be constructed from positive atoms using the connectives $\wedge, \vee, \exists$. Each such sentence can be expressed as $Q = q_1 \vee q_2 \vee \ldots \vee q_k$, where each q_i is a conjunctive query (i.e. uses only the connectives $\wedge, \exists$), and therefore such a query is also known as a *union of conjunctive queries*, UCQ. Alternatively, the query can be expressed as a *non-recursive datalog* program [Abiteboul et al. 1995].

We prove in this paper the following Dichotomy Theorem (Theorem 4.11). For any UCQ Q, one of the following holds: either $P_{\mathbf{D}}(Q)$ can be computed in PTIME in the size of $\mathbf{D}$, or the problem "given $\mathbf{D}$, compute $P_{\mathbf{D}}(Q)$" is hard for $FP^{\#P}$.

Using simple transformations, our results can also be applied to the positive, universal subset of First Order Logic, i.e. to sentences using only the connectives $\wedge, \vee, \forall$. Such sentences are used often in knowledge representation. For example, in Markov Logic Networks [Richardson and Domingos 2006], where First Order Logic is extended with probabilities in order to capture degrees of confidence in uncertain data, the knowledge base consists of a set of universally quantified clauses. Our results can also be applied to sentences with a limited use of negation: the negation must be applied directly to atoms, and every relation symbol must be *unnate* [Golumbic et al. 2006], i.e. either all its occurrences are positive, or all its occurrences are negative.

Given a query Q and a database instance $\mathbf{D}$, the *lineage* of Q on $\mathbf{D}$ is a Boolean Expression Φ_Q^D, which captures the evaluation of the query on the database instance. The lineage has one Boolean variable for each tuple in the database, and expresses which tuples must be present in the database in order for the query to be true; we give the formal definition is in Sect. 2. The lineage is a special form of provenance [Green et al. 2007], where it is called *PosBool*, because for a UCQ, the lineage is a positive Boolean expression. Furthermore, it has a DNF representation whose size is polynomial in $\mathbf{D}$. The probability of the query Q is equal to the probability of the lineage $\Phi_Q^{\mathbf{D}}$, when each Boolean variable is set independently to `true` with a probability $P(t)$, where t is the tuple corresponding to the Boolean variable. In other words, $P_{\mathbf{D}}(Q) = P(\Phi_Q^D)$, and the query evaluation problem is the same as the problem of computing the probability of a Boolean expression. Therefore, our dichotomy theorem is also a dichotomy theorem for classes of positive Boolean expressions, defined by their structure. Each query Q defines the class of Boolean expressions consisting of all lineage expressions of the form Φ_Q^D: for each such class, the problem "given a formula Φ in this class, compute $P(\Phi)$" is either in PTIME or is hard for $FP^{\#P}$. While the natural representation of lineage expression is in DNF, our results extend immediately to their dual CNF counterparts, obtained by negating the formula, and substituting each variable with its negation. Equivalently, the dual is obtained by simply switching $\wedge$ and $\vee$. For a taste of how our results apply to Boolean expressions in CNF, Table I shows several queries Q, and

Query	The Dual of the Lineage	$P(\Phi)$
$q_J = R(x_1), S(x_1, y_1), S(x_2, y_2), T(x_2)$	$\Phi = \bigwedge_{ijkl}(Y_i \vee X_{ij} \vee X_{kl} \vee Z_k)$	PTIME
$q_U = R(x_1), S(x_1, y_1) \vee S(x_2, y_2), T(x_2)$	$\Phi = \bigwedge_{ij}(Y_i \vee X_{ij}) \wedge \bigwedge_{ij}(X_{ij} \vee Z_i)$	PTIME
$h_1 = R(x_1), S(x_1, y_1) \vee S(x_2, y_2), T(y_2)$	$\Phi = \bigwedge_{ij}(Y_i \vee X_{ij}) \wedge \bigwedge_{ij}(X_{ij} \vee Z_j)$	$FP^{\#P}$-hard
$q_V = R(x_1), S(x_1, y_1) \vee S(x_2, y_2), T(y_2) \vee R(x_3), T(y_3)$	$\Phi = \bigwedge_{ij}(Y_i \vee X_{ij}) \wedge \bigwedge_{ij}(X_{ij} \vee Z_j) \wedge \bigwedge_{ij}(Y_i \vee Z_j)$	PTIME
$q_H = (R(x_1), S_1(x_1, y_1) \vee S_2(x_2, y_2), S_3(x_2, y_2)) \wedge (S_1(x_3, y_3), S_2(x_3, y_3) \vee S_3(x_4, y_4), T(y_4))$	$\Phi = \bigwedge_{i_1 j_1 \ldots i_4 j_4} [((Y_{i_1} \vee X^1_{i_1 j_1}) \wedge (X^2_{i_2 j_2} \vee X^3_{i_2 j_2})) \vee ((X^1_{i_3 j_3} \vee X^2_{i_3 j_3}) \wedge (X^3_{i_4 j_4} \vee Z_{j_4}))]$	$FP^{\#P}$-hard
$q_W = (R(x_1), S_1(x_1, y_1) \vee S_2(x_2, y_2), S_3(x_2, y_2)) \wedge (R(x_3), S_1(x_3, y_3) \vee S_3(x_4, y_4), T(y_4)) \wedge (S_1(x_5, y_5), S_2(x_5, y_5) \vee S_3(x_6, y_6), T(y_6))$	$\Phi = \bigwedge_{i_1 j_1 \ldots i_6 j_6} [((Y_{i_1} \vee X^1_{i_1 j_1}) \wedge (X^2_{i_2 j_2} \vee X^3_{i_2 j_2})) \vee ((Y_{i_3} \vee X^1_{i_3 j_3}) \wedge (X^3_{i_4 j_4} \vee Z_{j_4})) \vee ((X^1_{i_5 j_5} \vee X^2_{i_5 j_5}) \wedge (X^3_{i_6 j_6} \vee Z_{j_6}))]$	PTIME

Table I. Some simple applications of the dichotomy result. The first column shows a query: all variables are existentially quantified. The middle column shows the Boolean expression obtained by taking the dual of their lineage expression (switching the roles of $\wedge, \vee$). The Boolean variables Y_i, X^k_{ij}, Z_j correspond to the tuples $R(i), S_k(i, j), T(j)$ respectively. The first four expressions are already in CNF. The last two expressions are not in CNF, but can be rewritten in CNF with only a constant factor increase in their size. The last column indicates whether $P(Q)$, or, equivalently, $P(\Phi)$, is tractable or not. Consider the row for h_1, where $P(\Phi)$ is hard for $FP^{\#P}$. Compare it to q_U, where Z_j becomes Z_i, and the complexity of $P(\Phi)$ is PTIME. Also, compare it to q_V, where a new set of clauses is added to Φ and the complexity is also PTIME. A similar comparison can be done for q_H and q_V.

the class of Boolean expressions Φ corresponding to the dual of their lineage. For each class, we indicate whether its complexity is PTIME or hard for $FP^{\#P}$.

A special case of the probability computation problem is the counting problem: given a Boolean expression Φ, count the number of satisfying assignments $\#\Phi$. Valiant [Valiant 1979] showed that #SAT, the counting version of the SAT problem is #P-complete. Provan and Ball [Provan and Ball 1983] have shown that for a very simple class of Boolean expressions, called *partitioned, positive, 2CNF* (reviewed in

Sect. 8.1), the counting problem is already #P-complete. Our PTIME algorithm for $P(\Phi)$ extends immediately to the counting problem, since $\#\Phi = 2^n P(\Phi)$, where n is the number of Boolean variables, and the probability $P(\Phi)$ is computed by setting each Boolean variable independently to `true` with probability 1/2. Our hardness results do not extend immediately to the counting version; more precisely, our hardness proof is not a parsimonious reduction from #SAT, but one that uses an oracle: our hardness result proves that computing $P(\Phi)$ is hard for $FP^{\#P}$, the class of functions computable by a polynomial-time Turing Machine with access to an oracle for a #P hard problem.

Creignou and Hermann proved a dichotomy theorem [Creignou and Hermann 1996] for the *generalized satisfiability counting problem*, #GenSAT. They consider Boolean expressions that are conjunctions of *logical relations*, also known as generalized clauses, and establish a dichotomy for the counting problem based on the type of logical relations: they show that the counting problem is in PTIME iff every logical relation is affine. Thus, they restrict the formulas based on the generalized clauses, and allow otherwise arbitrary structure. We do the opposite: we restrict the structure, and obtain many interesting tractable cases that are not identified by Creignou and Hermann's result. This is illustrated in Table I: all clauses in this table are of the form $x \vee y \vee z \vee \ldots$ and therefore fall in the #P-class of Creignou and Hermann.

We discuss now in some more details the two parts of our dichotomy result: the PTIME algorithm for the tractable queries, and the hardness proof for the intractable queries.

The Algorithm We have first described the PTIME algorithm in [Dalvi et al. 2010a]: we review it here in Sect. 3, with some minor changes to improve its presentation and to also improve its performance. The algorithm has two simple steps. In the first step it applies the inclusion/exclusion formula to remove the outer-most $\wedge$ operations in a query. In the second step it removes one outermost $\exists$, by choosing a variable that satisfies a certain property and is called a *separator variable*. Thus, the algorithm alternates between the inclusion/exclusion step, and the separator variable step, until it reaches ground atoms, for which it looks up their probabilities in $\mathbf{D}$.

The inclusion/exclusion is the dual of the popular version: the algorithm computes $P(q_1 \wedge q_2 \wedge \ldots)$ in terms of $P(q_{i_1} \vee q_{i_2} \vee \ldots)$, rather than the other way around. For that reason, we must first rewrite Q from its traditional representation of a union of conjunctive queries, into a conjunction of disjunctive queries, a format that we call the CNF representation of a query. In particular, even if one starts with simple conjunctive queries, like q_J in Table I, the algorithm may eventually introduce $\vee$. In other words, the class of conjunctive queries is not a natural class to study in terms of probabilities: the natural class is that of unions of conjunctive queries. The reliance on the CNF rather than the DNF representation, and the use of the dual inclusion/exclusion formula are surprising features of the algorithm.

In order for the algorithm to be complete, it needs to trace terms in the inclusion/exclusion formula that may cancel out: we do this by replacing the inclusion/exclusion formula with Mobius' inversion formula in a lattice. Recognizing when terms cancel out is of crucial importance, because sometimes all hard terms

cancel out, and we can compute the query in PTIME, although some terms in the inclusion/exclusion formula are hard. The concept that allows us to do that is Mobius' function in a lattice [Stanley 1997]. We define a certain *CNF-lattice* for a query Q, whose elements correspond to queries that can be thought of as generalized subexpressions of Q. In this case the Mobius function of a subexpression q_u is precisely the coefficient of the term $P(q_u)$ in the inclusion/exclusion formula. If the Mobius function is 0, then that term cancels out, and can thus be discarded when computing $P(Q)$. In some sense, the Mobius function acts like a syntactic feature of Q, which must be examined in order to determine if Q is in PTIME or hard for $FP^{\#P}$. We show that this is unavoidable: for every lattice L, one can construct a query Q whose CNF lattice is L, and where all subqueries are in PTIME except the query that is the minimal element of L. Therefore, Q is in PTIME iff the Mobius function at that minimal element is 0. This is a surprising connection between the Mobius function and the computational complexity of a query. The inclusion/exclusion formula, or, equivalently, Mobius' inversion formula, have been used before in probabilistic inference [Knuth 2005]. However, the strong connection between the Mobius function's zeroes is new.

The power of our simple algorithm has been highlighted by recent results in [Jha and Suciu 2010]. Several notions of tractability for computing $P(\Phi)$ were discussed there: read-once Boolean expressions [Golumbic et al. 2006], polynomial-size OBDDs and FBDDs [Wegener 2000; Wegener 2004], and polynomial-size d-DNNFs [Darwiche 2000; Darwiche and Marquis 2002]. It was shown that they form a strict hierarchy for unions of conjunctive queries: in particular, the queries q_J, q_V, q_W, q_9 separate these classes (e.g. q_J is not read-once, but has a polynomial-size OBDD, etc). It is further conjectured that q_9 (Fig. 2), whose probability can be computed in PTIME using our algorithm, does not have a polynomial size d-DNNF.

In our earlier work [Dalvi and Suciu 2004; Dalvi and Suciu 2007b] we have given a PTIME algorithm for a much more restricted query language: conjunctive queries without self-joins. The version of the algorithm that is described in the paper is a conservative extension of that algorithm over tuple-independent databases.

Hardness proof The hardness proof of our dichotomy theorem is entirely new, and forms the main technical contribution of this paper. We prove the following. If Q is any query on which the algorithm fails (called an *unsafe query*), then there exists a PTIME Turing machine with an oracle for computing $P_{\mathbf{D}}(Q)$ that solves the counting problem for the partitioned-positive-2CNF problem. The latter was shown to be #P-hard by Provan and Ball [Provan and Ball 1983]. This implies that $P_{\mathbf{D}}(Q)$ is hard for $FP^{\#P}$. In our proof, the oracle for $P_{\mathbf{D}}(Q)$ is called polynomially many times, on the same database instance D, but with varying probabilities P. The proof consists of three main steps, which we explain next.

The first step, described in Sect. 6, is called *leveling*, and it transforms any unsafe query Q into another unsafe query Q', such that the hardness of Q' implies the hardness of Q, and Q' is a *leveled* query, meaning that it has the following property. Every attribute of every relational symbol in Q' can be associated a "type", called *level*, such that every variable x occurs only on attribute positions of the same level, and, moreover, for any relational symbol, all its attributes have distinct levels. This first step allows us to restrict the hardness proof to leveled

queries.

The second step, described in Sect. 7, applies the following *rewriting* step to a leveled query Q: choose certain variables $z_1, \ldots, z_k$ and substitute them with constants. If Q' is the rewritten query and Q' is hard, then it is not difficult to prove that Q is also hard. In this second step we show that, if Q is an unsafe query with at least three levels, then it is always possible to rewrite it to another unsafe query Q'. This rewriting process always terminates, hence the second step allows us to restrict the hardness proof to unsafe, leveled queries with two levels, which we call *forbidden queries*. The difficulty of this second step consists of choosing the variables $z_1, \ldots, z_k$ such that Q' is still unsafe, which turns out to be quite subtle.

The third step, described in Sect. 8, proves that every unsafe, forbidden query Q is hard for $FP^{\#P}$. After several technical preparation steps, here we construct a direct reduction of Provan and Ball's partitioned-positive-2CNF counting problem $\#\Phi$ to the problem $P_{\mathbf{D}}(Q)$. From Φ we construct a certain database $\mathbf{D}$ and show that $P_{\mathbf{D}}(D)$ depends on $\#\Phi$, as well as on (polynomially many) other parameters of Φ. By changing the tuple probabilities in $\mathbf{D}$ and computing $P_{\mathbf{D}}(D)$ repeatedly, we can construct a linear system of equations and compute all parameters, including $\#\Phi$, using Gaussian elimination. The difficult step, and the crux of the entire proof, is to show that the matrix of the system of equations is non-singular. In fact, if the original query Q is safe, then the matrix *is* singular. Here, we use techniques from classical algebra, such as irreducible polynomials in the ring of multivariate polynomials, and Cauchy's double alternant determinant.

Related work Grädel et al. [Grädel et al. 1998] were the first to given an example of a query Q for which $P(Q)$ is $FP^{\#P}$-hard; their work was done in the context of query reliability. In the following years, several studies [Dalvi and Suciu 2004; Dalvi and Suciu 2007b; Olteanu et al. 2009; Olteanu and Huang 2009], sought to identify classes of tractable queries. These works provided conditions for tractability only for conjunctive queries without self-joins. The only exception is [Dalvi and Suciu 2007a], which considers conjunctive queries with self-joins. We extend those results to a larger class of queries, and at the same time provide a very simple algorithm. Some other prior work is complimentary to ours, e.g., the results that consider the effects of functional dependencies [Olteanu et al. 2009].

2. BACKGROUND AND OVERVIEW

In this paper we discuss *unions of conjunctive queries* (UCQ), which are expressions defined by the following grammar:

$$Q ::= R(\bar{x}) \mid \exists x.Q_1 \mid Q_1 \wedge Q_2 \mid Q_1 \vee Q_2 \tag{2}$$

$R(\bar{x})$ is a relational atom with variables and/or constants, whose relation symbol R is from a fixed vocabulary. We replace $\wedge$ with comma, and drop $\exists$, when no confusion arises: for example we write $R(x), S(x)$ for $\exists x.(R(x) \wedge S(x))$.

A *query* is an expression up to logical equivalence. We consider only Boolean queries in this paper. A *conjunctive query* (CQ) is a query without $\vee$.

Let D be a database instance. Denote X_t a distinct Boolean variable for each tuple $t \in D$. Let Q be a UCQ. The *lineage* of Q on D is the Boolean expression Φ_Q^D,

or simply Φ_Q if D is understood from the context, defined inductively as follows, where $ADom$ denotes the active domain of the database instance:

$$\Phi_{R(\bar{a})} = X_{R(\bar{a})} \qquad \Phi_{\exists x.Q} = \bigvee_{a \in ADom} \Phi_{Q[a/x]} \tag{3}$$

$$\Phi_{Q_1 \wedge Q_2} = \Phi_{Q_1} \wedge \Phi_{Q_2} \qquad \Phi_{Q_1 \vee Q_2} = \Phi_{Q_1} \vee \Phi_{Q_2} \tag{4}$$

The lineage expression is a particular instance of a provenance expression in semi-rings [Green et al. 2007]: in the terminology of provenance expressions over semi-rings, our lineage expresses are called *PosBool*, because for queries without negations they are positive Boolean expressions. Notice that, for a fixed Q, the size of the Boolean expression Φ_Q defined above is polynomial in the size of D. Furthermore, since the depths of the expression is $O(1)$ and the outdegree of every $\wedge$ operators is $O(1)$, by applying the distributivity law we can obtain a DNF expression for Φ_Q whose size is polynomial in the size of D.

The *query evaluation problem on probabilistic databases* is the following. Given numbers $P(t) \in [0, 1]$, for every tuple t in the database D, compute the probability that the formula Φ_Q is `true`, if each Boolean variable X_t is set to `true` independently, with probability $P(t)$; the resulting probability is denoted $P(Q) = P(\Phi_Q^D)$.

We call the pair $\mathbf{D} = (D, P)$ a *probabilistic database instance*, and sometimes use $\mathbf{D}$ in the subscript, $P_{\mathbf{D}}(Q)$, to emphasize that the query's probability is computed on the instance $\mathbf{D}$.

Definition 2.1. *$UCQ(P)$ the class of UCQ queries Q s.t. for any probabilistic database $\mathbf{D}$, the probability $P_{\mathbf{D}}(Q)$ can be computed in PTIME in the size of $\mathbf{D}$.*

In this paper we give a complete syntactic characterization of the class $UCQ(P)$, and establish a dichotomy, by proving that for every query not in $UCQ(P)$, computing $P(Q)$ is hard for $FP^{\#P}$. Notice that each query Q defines a family of Boolean expressions, namely the family Φ_Q^D where D ranges over all finite databases. Thus, our results are also a characterization of families of Boolean expressions that are computable in PTIME, and establish a dichotomy on these families.

We will assume without any loss of generality that there are no constants in a query Q. Otherwise, we rewrite Q into an equivalent query without constants, over an extended vocabulary. For example, $R(x, a), S(x) \vee R(x, y), T(x)$ is rewritten as $R_1(x), S(x) \vee R_1(x), T(x) \vee R_2(x, y), T(y)$, where $R_1(x) = \pi_x(\sigma_{y=a}(R(x, y)))$ and $R_2(x, y) = \sigma_{y \neq a}(R(x, y))$. Thus, throughout the paper we will assume that queries do not have constants, unless otherwise stated.

We start with some basics:

—A *component*, c, is a conjunctive query that is *connected*[1], i.e. whenever q_1, q_2 are two conjunctive queries such that $q_1 \wedge q_2 \Rightarrow c$, then either $q_1 \Rightarrow c$ or $q_2 \Rightarrow c$. Given a component c, we denote D_c the *canonical database* for c: its active domain consists of all constants and variables in c, and it has a tuple for each

[1]The traditional definition of a connected conjunctive query is the following: c is connected if the following undirected graph is connected. The nodes are the atoms of c, and the edges are pairs (g, g') s.t. the atoms g and g' share a common variable. For queries without constants, this is equivalent to our definition.

atom in c [Abiteboul et al. 1995]. Given two components c, c', the following three statements are equivalent: the logical implication $c \Rightarrow c'$ holds; there exists a homomorphism $c' \rightarrow c$; $D_c \models c'$ [Chandra and Merlin 1977].

—A *conjunctive query* is a conjunction of components, $q = c_1, c_2, \ldots, c_k$. Given two conjunctive queries q, q', the implication $q \Rightarrow q'$ holds iff $\forall j.\exists i$ s.t. $c_i \rightarrow c'_j$.

—A *disjunctive query* is a disjunction of components, $d = c_1 \vee \ldots \vee c_k$. The classic result by Sagiv and Yannakakis [Sagiv and Yannakakis 1980] implies that an implication $d \Rightarrow d'$ holds iff $\forall i.\exists j$ s.t. $c_i \Rightarrow c'_j$.

—A UCQ in DNF is an expression of the form $Q = q_1 \vee \ldots \vee q_m$. An implication $Q \Rightarrow Q'$ holds iff $\forall i.\exists j$ s.t. $q_i \Rightarrow q'_j$ [Sagiv and Yannakakis 1980].

—A UCQ in CNF is an expression of the form $Q = d_1 \wedge \ldots \wedge d_m$.

The terms *DNF expression* and *CNF expression* are justified by viewing the components as atoms. Then the DNF expression is a disjunction of conjunction of atoms, while the CNF expression is a conjunction of disjunctions of atoms. One should keep in mind, however, that the components have their own conjunction, thus a CNF expression is, in fact, a conjunction of disjunctions of existential quantifiers of conjunctions.

Throughout this paper we will use the letters c, q, d, Q to denote a component, conjunctive query, disjunctive query, and union of conjunctive queries respectively, unless otherwise stated.

For CNF expressions we prove the following:

PROPOSITION 2.2. *If $Q = \bigwedge_i d_i$ and $Q' = \bigwedge_j d'_j$ are two UCQ queries in CNF, and have no constants, then the implication $Q \Rightarrow Q'$ holds iff $\forall j.\exists i$ s.t. $d_i \Rightarrow d'_j$.*

PROOF. Suppose the contrary: there exists an index j such that $\forall i$, $d_i \not\Rightarrow d'_j$. Thus, forall i, by Sagiv and Yannakakis' criteria we obtain that $\exists k_i$ s.t $c_{ik_i} \not\Rightarrow d'_j$, where c_{ik_i} is one of the components of the disjunctive query d_i. On the other hand, $\forall i$, $c_{ik_i} \Rightarrow d_i$, implying that $\bigwedge_i c_{ik_i} \Rightarrow Q \Rightarrow Q' \Rightarrow d'_j$. The left side is a conjunctive query, the right side a disjunctive query. Applying Sagive and Yannakakis' criteria a second time, we obtain a component c'_{jl} of d'_j s.t. $\bigwedge_i c_{ik_i} \Rightarrow c'_{jl}$. Now we use the fact that $c_{1k_1}, c_{2k_2}, \ldots$ have no constants, and that c'_{jl} is a component: hence there exists i s.t $c_{ik_i} \Rightarrow c'_{jl}$, contradicting the fact that $c_{ik_i} \not\Rightarrow d'_j$. $\square$

The proposition fails if the queries have constants: for example $R(x, a), S(a, z) \Rightarrow R(x, y), S(y, z)$ (where a is a constant), but neither $R(x, a) \not\Rightarrow R(x, y), S(y, z)$ nor $S(a, z) \not\Rightarrow R(x, y), S(y, z)$. This is the main reason why we eliminate constants.

We will assume throughout the paper that a query is *minimized*. We review briefly: a component c is minimized if no component c_0 exists with strictly fewer atoms s.t. $c_0 \Rightarrow c$; a conjunctive query $\bigwedge c_i$ (disjunctive query $\bigvee c_i$) is minimized if each c_i is minimized and $i \neq j$ implies $c_i \not\Rightarrow c_j$; a CNF $\bigvee_i q_i$ (DNF $\bigwedge d_i$) is minimized if each q_i (d_i) is minimized and $i \neq j$ implies $q_i \not\Rightarrow q_j$ ($d_i \not\Rightarrow d_j$).

As a consequence of the containment criteria for the DNF and CNF representation, each UCQ admits a unique (up to isomorphism) representation in DNF, and a unique (up to isomorphism) representation in CNF that are minimized. From now on, whenever we say that a query is given in DNF or in CNF, we assume it is given in its minimized representation, which is unique up to isomorphism. Furthermore, we will constantly move back and forth between the DNF and the CNF

Algorithm 1 Algorithm from [Dalvi and Suciu 2004; Dalvi and Suciu 2007b] computing $P(q)$ for a conjunctive query without self-joins

```
1: Write q as a conjunction of components q = c_1, ..., c_m.
2: if m ≥ 2 then
3:    return P(c_1) · P(c_2) ··· P(c_m) /* independent join */
4: end if
5:
6: /* q is a single component: denote it c */
7: if c has no variables then
8:    /* c consists of a single ground tuple t */
9:    return P(t) /* look up in the probabilistic database */
10: end if
11:
12: if c has a root variable z then
13:    return 1 − ∏_{a∈ADom}(1 − P(c[a/z])) /* independent project */
14: end if
15:
16: Otherwise FAIL
```

representation of a UCQ: this may come at the cost of an exponential increase in the size of the expression, but does not affect the complexity of evaluating $P(Q)$, which is measured only in terms of the size of the database. To give an example of a DNF to CNF conversion, consider the query:

$$q_V = R(x_1), S(x_1, y_1) \vee S(x_2, y_2), T(y_2) \vee R(x_3), T(y_3)$$

The third conjunctive query has two components, $R(x_3)$ and $T(y_3)$. We convert it into CNF, then minimize:

$$\begin{aligned} q_V =&(R(x_1), S(x_1, y_1) \vee S(x_2, y_2), T(y_2) \vee R(x_3)) \\ &\wedge(R(x_1), S(x_1, y_1) \vee S(x_2, y_2), T(y_2) \vee T(y_3)) \\ =&(R(x_3) \vee S(x_2, y_2), T(y_2)) \wedge (R(x_1), S(x_1, y_1) \vee T(y_3)) \end{aligned}$$

Finally, we define a root variable. Recall that the *scope* of a variable x is the part of the expression where the variable is defined: for example, in $q_1 \wedge \exists x.q_2$, the scope of x is q_2.

Definition 2.3. *Consider a query expression Q given by the grammar in Eq. 2. A variable z is called a* root variable *if it occurs in all atoms within its scope.*

Prior Work A conjunctive query is said to be without *self-joins* if no two atoms have the same relational symbol. The algorithm 1 was first described in [Dalvi and Suciu 2004; Dalvi and Suciu 2007b] and computes the probabilities for conjunctive queries without self-joins. The algorithm proceeds inductively on the structure on the query. If the query is disconnected, then it multiplies the probabilities of its components. If the query is a single ground tuple t then it looks up and returns the tuple's probability in the probabilistic database. If the query is connected and has

variables, then it chooses a root variable z and substitutes it successively with all constants in the active domain: the resulting queries $c[a/z], a \in ADom$ are independent, and their probabilities are combined to compute $P(c)$. Finally, if the query has no root variable, then the algorithm fails. The two steps are called *independent join* and *independent project* respetively, because they can be implemented as a join, or as a project operator in Relational Algebra, over independent probabilistic events [Dalvi and Suciu 2004].

The algorithm was proven in [Dalvi and Suciu 2004; Dalvi and Suciu 2007b] to be complete for conjunctive queries without self-joins, in the following sense. For any query q, the algorithm either succeeds in computing the probability of q, or, if it fails, then the query is hard for $FP^{\#P}$. For example, the query $q = R(x), S(x,y), T(y)$ is connected and has no root variable: therefore it is hard for $FP^{\#P}$. We briefly illustrate the algorithm on an example.

EXAMPLE 2.4. *Consider $q = R(x), S(x,y) = \exists x.R(x) \wedge \exists y.S(x,y)$. algorithm 1 computes its probability as follows:*

$$
\begin{aligned}
P(q) &= 1 - \prod_{a \in ADom} (1 - P(R(a), \exists y.S(a,y))) \\
P(R(a), \exists y.S(a,y)) &= P(R(a)) \cdot P(\exists y.S(a,y)) \\
P(\exists y.S(a,y)) &= 1 - \prod_{b \in ADom} (1 - P(S(a,b)))
\end{aligned}
$$

Here we have shown the existential quantifiers $\exists y$ explicitly. In most of the rest of the paper we will drop quantifiers.

However, the algorithm cannot be applied beyond conjunctive queries without self-joins. For example, consider the query $R(x_1), S(x_1,y_1), S(x_2,y_2), T(x_2)$. Here we cannot apply multiply the probabilities of the two components, because they share the common symbol S.

3. A SIMPLE ALGORITHM

We start by describing a very simple algorithm for computing the probability of a union of conjunctive queries. The algorithm is incomplete yet, but only two adjustments (shown in Sect. 4) are sufficient to make it complete. We motivate the algorithm with an example:

EXAMPLE 3.1. *Consider the query $q = R(x_1), S(x_1,y_1), S(x_2,y_2), T(x_2)$, which can be written as $q = c_1, c_2$, where $c_1 = R(x_1), S(x_1,y_1)$, $c_2 = S(x_2,y_2), T(x_2)$. To compute $P(q)$ we use the inclusion-exclusion formula:*

$$P(c_1, c_2) = P(c_1) + P(c_2) - P(c_1 \vee c_2)$$

We have seen in Example 2.4 how to compute $P(c_1)$, and $P(c_2)$ is computed similarly. We show here how to compute $P(c_3)$, where $c_3 = c_1 \vee c_2$, by writing it as

follows:

$$c_3 = \exists z.((R(z) \vee T(z)) \wedge \exists y.S(z,y)) = \bigvee_{a \in ADom} ((R(a) \vee T(a)) \wedge \exists y.S(a,y))$$

$$P(c_3) = 1 - \prod_{a \in ADom} (1 - P((R(a) \vee T(a)) \wedge \exists y.S(a,y)))$$

The last line holds because the queries $(R(a) \vee T(a)) \wedge \exists y.S(a,y)$ for $a \in ADom$ are independent. Next, $P((R(a) \vee T(a)) \wedge \exists y.S(a,y)) = P((R(a) \vee T(a))) \wedge P(\exists y.S(a,y))$. Thus, $P(c_3)$ can be computed in PTIME in the size of the database.

The example illustrates an important point: in order to compute the probability of a conjunctive query with self-joins, we had to compute the probability of a disjunctive query $c_1 \vee c_2$ as an intermediate step. In other words, the class of conjunctive queries is not a natural class to study: the natural class is that of unions of conjunctive queries, UCQ.

To describe the algorithm for UCQ we need two definitions. First:

Definition 3.2. *A* separator variable *for a query Q is a variable z s.t. $Q \equiv \exists z.Q_1$ and (a) z is a root variable (i.e. it appears in every atom), (b) for every relation symbol R, there exists a number i_R s.t. every atom with symbol R contains exactly one occurrence of z, and that is on the position i_R.*

Only a disjunctive query can have a separator variable. To see this, write $Q \equiv \exists z.Q_1$ then expand Q_1 in DNF: $Q \equiv \exists z.q_1 \vee \exists z.q_2 \vee \ldots \vee \exists z.q_m$. Since z is a root variable in each expression $\exists z.q_i$, it follows that each such expression is a component, and therefore Q is a disjunctive query.

On the other hand, not every disjunctive query has a separator variable. A trivial example is $R(x), S(x,y), T(y)$, which has no separator variable because it doesn't even have a root variable. More subtly, $R(x_1), S(x_1,y_1) \vee S(x_2,y_2), T(y_2)$ does have a root variable if one writes it as $\exists z.(R(z), \exists y_1.S(z,y_1) \vee T(z), \exists x_2.S(x_2,z))$. But z is not a separator variable because it occurs on the first position in $S(z,y_1)$ and on the second position in $S(x_2,z)$.

We have:

Proposition 3.3. *Let d be a disjunctive query with a separator variable z. Then the events $d[a/z]$, $a \in ADom$ are independent probabilistic events. In particular:*

$$P(d) = 1 - \prod_{a \in ADom} (1 - P(d[a/z])) \tag{5}$$

Proof. Let $a \neq b$ be two distinct constants in the active domain. Consider the lineage expressions of the queries $d[a/z]$ and $d[b/z]$. We claim that these two Boolean expressions do not share any common Boolean variables. Indeed, suppose they share X_t, where t is a ground tuple: $t = R(c_1, c_2, \ldots)$. Since z is a separator variable, every ground tuple of R occurring in the lineage of $d[a/z]$ has the constant a on position i_R: in other words, $c_{i_R} = a$. Reasoning similarly for $d[b/z]$, we also conclude that the ground tuple contains b on position i_R, i.e. $c_{i_R} = b$. This is a contradiction because $a \neq b$. □

The second definition is:

Definition 3.4. *Let $\mathbb{Q} = \{Q_1, Q_2, \ldots, Q_k\}$ be a set of queries. The* co-occurrence graph *of $\mathbb{Q}$ is the following undirected graph: the nodes are $1, 2, \ldots, k$, and the edges are pairs (i, j) s.t. there exists a relational symbol that occurs both in Q_i and in Q_j. Let $K_1, \ldots, K_m$ be the connected components (they form a partition on $[k]$).*

Let $Q = d_1 \wedge \ldots \wedge d_k$ be a UCQ query written in CNF, and $K_1, \ldots, K_m$ the connected components of set of queries $\{d_1, \ldots, d_k\}$. The *symbol-components* of Q are:

$$Q_1 = \bigwedge_{i \in K_1} d_i, \qquad Q_2 = \bigwedge_{i \in K_2} d_i \qquad \ldots \qquad Q_m = \bigwedge_{i \in K_m} d_i$$

Then we have:

$$P(Q) = P(Q_1) \cdot P(Q_2) \cdots P(Q_m) \tag{6}$$

If $m = 1$, then we say that Q is *symbol-connected*.

Similarly, if $d = c_1 \vee \ldots \vee c_k$ is a disjunctive query, and $K_1, \ldots, K_m$ are connected components of the set of queries $\{c_1, \ldots, c_k\}$, then the *symbol-components* of d are:

$$d_1 = \bigvee_{i \in K_1} c_i, \qquad d_2 = \bigvee_{i \in K_2} c_i \qquad \ldots \qquad d_m = \bigvee_{i \in K_m} c_i$$

Then we have:

$$P(d) = 1 - (1 - P(d_1)) \cdot (1 - P(d_2)) \cdots (1 - P(d_m))$$

If $m = 1$, then we say that d is *symbol-connected*.

We can now describe a very simple algorithm for computing $P(Q)$ for a UCQ Q: algorithm 2 either computes $P(Q)$ or fails. The algorithm proceeds inductively on the structure of Q: each expressions $P(Q')$ represents a recursive call, with a simpler query Q'. There are four main steps: an independent join, the inclusion/exclusion formula, an independent union, and an independent project. The first and last step generalize those of algorithm 1, and are justified by Eq. 6 and Eq. 5 respectively. In fact, when run on a conjunctive queries without self-joins, algorithm 2 is identical to algorithm 1.

The inclusion/exclusion formula in step 9 is the dual of the more familiar one, because it is applied to a conjunction, like:

$$P(d_1 \wedge d_1 \wedge d_3) = P(d_1) + P(d_2) + P(d_3) - P(d_1 \vee d_2) - P(d_1 \vee d_3) - P(d_2 \vee d_3) + P(d_1 \vee d_2 \vee d_3)$$

This is the dual of the more familiar inclusion/exclusion formula:

$$P(d_1 \vee d_1 \vee d_3) = P(d_1) + P(d_2) + P(d_3) - P(d_1 \wedge d_2) - P(d_1 \wedge d_3) - P(d_2 \wedge d_3) + P(d_1 \wedge d_2 \wedge d_3)$$

Note how the algorithm reaches the end of the recursion. If one of the c_i's in a disjunctive query $d = \bigvee_i c_i$ is a ground tuple, then it is a symbol-component by itself: this ground tuple cannot occur in any other component c_j, because c_j is connected. Thus, c_i will be isolated by step 15, and at the next step it is treated as a query without variables: the algorithm simply looks up its probability in the database.

Finally, if the algorithm ever reaches a disjunctive query that has no separator, then it fails.

Algorithm 2 Algorithm for Computing $P(Q)$
`Input:` UCQ Q, and Probabilistic database with active domain $ADom$
`Output:` $P(Q)$
`Comments:` Two small adjustments are needed to make the algorithm complete: (1) rank the query before running the algorithm, see Sect. 4.1, and replace the inclusion-exclusion formula in Line 9 with Mobius's inversion function in Line 11, see Sect. 4.2.

1: Compute the symbol-components: $Q = Q_1 \wedge \ldots \wedge Q_m$
2: **if** $m \geq 2$ **then**
3:    **return** $P(Q_1) \cdot P(Q_2) \cdots P(Q_m)$ /* independent join */
4: **end if**
5:
6: /* Q is symbol-connected */
7: Write Q in CNF: $Q = d_1 \wedge \ldots \wedge d_k$
8: **if** $k \geq 2$ **then**
9:    **return** $-\sum_{s \subseteq [k], s \neq \emptyset} (-1)^{|s|} P(\bigvee_{i \in s} d_k)$ /* inclusion/exclusion */
10:       /* replaced with Mobius' inversion formula in Sect. 4.2: */
11:       /* **return** $-\sum_{v < \hat{1}, \mu(v,\hat{1}) \neq 0} \mu_L(v, \hat{1}) P(d_v)$ */
12: **end if**
13:
14: /* Q is a disjunctive query. Denote it d */
15: Compute the symbol-components: $d = d_1 \vee \ldots \vee d_m$
16: **if** $m \geq 2$ **then**
17:    **return** $1 - (1 - P(d_1)) \cdots (1 - P(d_m))$ /* independent union */
18: **end if**
19:
20: /* d is symbol-connected */
21: Write d as $d = c_1 \vee \ldots \vee c_k$
22: **if** d has no variables (i.e. d is a single ground tuple t) **then**
23:    **return** $P(t)$ /* look up the probability of t */
24: **end if**
25:
26: **if** d has a separator variable z **then**
27:    **return** $1 - \prod_{a \in ADom}(1 - P(d[a/z]))$ /* independent project */
28: **end if**
29: Otherwise **FAIL**

We assumed that the query has no constants, yet in the last step of the algorithm we construct the query $d[a/z]$, where a is a constant. But for all practical purposes, we can view this also as a query without constants. Indeed, for every relation symbol R, every atom referring to R in $d[a/z]$ will have a on the same attribute position i_R. We can simply remove that attribute from R, thus reducing its arity by one, and restrict the database instance to only those tuples that have the constant a on attribute i_R.

Clearly algorithm 2 is sound: whenever it succeeds, it computes correctly the probability $P(Q)$ and does this in PTIME in the size of the input database. We

want the algorithm to be complete, i.e. to not miss out any query in $UCQ(P)$. However, the algorithm is not yet complete: in the next section we make two adjustments to complete it.

4. THE COMPLETE ALGORITHM

There are two reasons why algorithm 2 is not complete, and each requires a different adjustment.

4.1 Ranking

First, the algorithm 2 may not find a separator variable, yet the query is still in $UCQ(P)$. For example, consider $q = R(x, y), R(y, x)$. The query has no separator. The root variable x is not a separator because it occurs on the first position in $R(x, y)$ and on the second position in $R(y, x)$; as a consequence the events $q[a/x]$, $a \in ADom$ are no longer independent. The same holds for y: it is not a separator for q. Thus, algorithm 2 fails immediately on q. But $q \in UCQ(P)$ by the following argument. Transform a probabilistic database over the vocabulary $R(A, B)$ into another probabilistic database, over the vocabulary $R_{A=B}(A)$, $R_{A<B}(A, B)$ and $R_{B<A}(B, A)$, as follows:

$$R_{A=B} = \Pi_A(\sigma_{A=B}(R)) \quad R_{A<B} = \Pi_{AB}(\sigma_{A<B}(R)) \quad R_{B<A} = \Pi_{BA}(\sigma_{B<A}(R))$$

Then rewrite the query q to $R_{A=A}(x) \vee R_{A<B}(x, y), R_{B<A}(x, y)$. Now x is a separator variable, allowing us to compute the query in PTIME (using algorithm 2).

This justifies the following definition. We will assume that the domain of constants is an ordered domain.

DEFINITION 4.1. *A query Q is* ranked *if it is consistent after the following transformation: for any atom, and any two attribute positions $i < j$ in that atom, add the predicate $x_i < x_j$ to the query, where x_i, x_j are the variables occurring on positions i and j respectively. (Note: x_i, x_j do not need to be distinct variables.)*

In a ranked query no variable is repeated in the same atom (as in $R(\ldots x, \ldots x, \ldots)$). Furthermore, if an atom contains both variables x, y and x comes before y, then in all other atoms that contain both variables, x comes before y (that is, no two atoms $R(\ldots x, \ldots y, \ldots)$ and $S(\ldots y, \ldots, x, \ldots)$ may exists).

For example, $R(x, y), R(y, z), R(x, z)$ is ranked because $x < y \wedge y < z \wedge x < z$ is consistent. The query $R(x, y), R(y, x)$ is not ranked because $x < y \wedge y < x$ is inconsistent; also $R(x, x, y)$ is not ranked because $x < x \wedge x < y$ is inconsistent.

We prove:

PROPOSITION 4.2. *Every UCQ query Q is computationally equivalent to a ranked UCQ query Q' over an extended vocabulary; that is, the problem "given $\mathbf{D}$ compute $P_{\mathbf{D}}(Q)$" can be reduced in polynomial time to "given $\mathbf{D}'$ compute $P_{\mathbf{D}'}(Q')$", and vice versa.*

PROOF. (Sketch) The proof idea is the same as in the example given earlier. For every symbol R, define a *ranking* for R to be a surjective function $\tau : [k] \to [m]$, where k is the arity of R, and m is a number such that $1 \le m \le k$. The extended vocabulary will consists of symbols R^τ, where R is a relation name and τ is a ranking for R: the arity of R^τ is m.

Given a ranking τ for R and two number $i, j \in [k]$, denote $op^{\tau}_{i,j}$ the unique relation $<$ or $=$ or $>$ for which $\tau(i)\ op^{\tau}_{i,j} \tau(j)$ holds. Let $g = R(x_1, \ldots, x_k)$ be an atom in Q, where $x_1, \ldots, x_k$ are variables, not necessarily distinct, and define the predicate $\rho^{g,\tau} = \bigwedge_{1 \leq i < j \leq k} x_i\ op^{\tau}_{i,j} x_j$. Let τ^{-1} be any left inverse of τ (choose one arbitrary), and define $\tau^{-1}(x_1, \ldots, x_k) = (x_{\tau^{-1}(1)}, \ldots, x_{\tau^{-1}(m)})$. Then, given a query Q, we define the new query Q' inductively on the structure of Q, following the grammar Eq. 2:

$$(R(\bar{x}))' = \bigvee_{\tau} (R^{\tau}(\tau^{-1}(\bar{x})) \wedge \rho^{R(\bar{x}),\tau})$$

$$(\exists x.Q_1)' = \exists x.(Q_1)' \quad (Q_1 \vee Q_2)' = Q_1' \vee Q_2' \quad (Q_1 \wedge Q_2)' = Q_1' \wedge Q_2'$$

Next we prove that Q and Q' are "equivalent" more precisely that their computation problem is equivalent. Given a database D, we transform into a new database D', as follows. For each symbol R and for each ranking τ, define:

$$R^{\tau} = \Pi_{\tau^{-1}(1) \cdots \tau^{-1}(m)} \sigma_{\bigwedge_{1 \leq i < j \leq k} i\ op^{\tau}_{i,j} j}(R)$$

The tuples in D are in 1-1 correspondence with those in D'. Therefore, when D is a probabilistic database (meaning that each tuple t has a probability $p(t)$), then so is D', and they have the same sets of possible worlds. It is straightforward to check that $P_D(Q) = P_{D'}(Q')$. For the converse, given a database D' for Q', define a new database D where each relation R consists of all tuples of the form $(a_{\tau(1)}, \ldots, a_{\tau(k)})$, for all tuples $(a_1, \ldots, a_m) \in R^{\tau}$ and for all rankings τ. Here, too, it is easy to check that the tuples of D and D' are in 1-1 correspondence, and $P_D(Q) = P_{D'}(Q')$.

It remains to remove the predicates $<, =, >$ from Q', while ensuring that it is ranked. First, write it in DNF, $Q' = \bigvee_i q_i$. For each q_i, one of two things may happen. Either the relational predicates are inconsistent, e.g. contain $x < x$, or a cycle in the strict order: then remove q_i from the definition of Q'. Or, the relational predicates are consistent: in this case, from each equivalence class defined by $=$ choose a unique variable x, and substitute all other variables in the same equivalence class with x; then remove all $<, =, >$ predicates. The resulting query no longer has any predicates $<, =, >$, it is ranked (since by re-introducing the predicates $<$ and $>$ it will remain consistent), and its semantics on the database D' has not changed (since each relation in D' already enforces the predicates $<$ and $>$ that we removed from the query). □

The *first adjustment* that we make to algorithm 2 is to rank the query before running the algorithm. Ranking needs to be done only once: if Q is ranked, then all subqueries processed recursively by algorithm 2 are also ranked. Therefore, one does not have to change the actual algorithm, one just needs to rank the query once, before running the algorithm.

DEFINITION 4.3. *A disjunctive query d is called* immediately unsafe *if it is symbol-connected, has variables, and has no separators.*

In other words, d is immediately unsafe if the algorithm gets stuck on d immediately, i.e. it fails on d immediately.

THEOREM 4.4. *If d is ranked and immediately unsafe then computing $P(d)$ is hard for $FP^{\#P}$.*

The theorem is the hardest technical result in this paper, and most of the paper consists of the proof of this result, starting with Sect. 5. As we have seen, the condition that d be ranked is necessary: a counterexample is $R(x, y), R(y, x)$, which is immediately unsafe, but is in $UCQ(P)$.

We illustrate the theorem with some queries that will be important further in the paper:

EXAMPLE 4.5. *All queries below are ranked, and immediately unsafe. Hence, none of these queries is in $UCQ(P)$, unless $P = \#P$.*

$$
\begin{aligned}
h_0 =& R(x_0), S_1(x_0, y_0), T(y_0)\\
h_1 =& R(x_0), S_1(x_0, y_0) \vee S_1(x_1, y_1), T(y_1)\\
h_2 =& R(x_0), S_1(x_0, y_0) \vee S_1(x_1, y_1), S_2(x_1, y_1) \vee S_2(x_2, y_2), T(y_2)\\
h_2 =& R(x_0), S_1(x_0, y_0) \vee S_1(x_1, y_1), S_2(x_1, y_1) \vee S_2(x_2, y_2), S_3(x_2, y_2) \vee S_3(x_3, y_3), T(y_3)\\
& \ldots\\
h_k =& R(x_0), S_1(x_0, y_0) \vee S_1(x_1, y_1), S_2(x_1, y_1) \ \vee \ldots \vee S_k(x_k, y_k), T(y_k)
\end{aligned}
$$

4.2 Mobius Inversion Formula

The second problem in algorithm 2 is that the inclusion-exclusion formula attempts to compute the probability of all queries of the form $\bigvee_{i \in s} d_i$, for all nonempty subsets s. If any of these queries is hard, then the algorithm eventually fails. But, in some cases, some of these queries are not needed in the inclusion/exclusion formula, because they cancel out with other, equivalent queries. We illustrate this with the query $q_W = d_1 \wedge d_2 \wedge d_3$ where:

$$
\begin{aligned}
d_1 =& R(x_1), S_1(x_1, y_1) \vee S_2(x_2, y_2), S_3(x_2, y_2)\\
d_2 =& R(x_3), S_1(x_3, y_3) \vee S_3(x_4, y_4), T(y_4)\\
d_3 =& S_1(x_5, y_5), S_2(x_5, y_5) \vee S_3(x_6, y_6), T(y_6)
\end{aligned}
$$

The inclusion/exclusion formula iterates over these three queries, as well as the following:

$$
\begin{aligned}
d_{12} =& d_1 \vee d_2 = R(x_1), S_1(x_1, y_1) \vee S_2(x_2, y_2), S_3(x_2, y_2) \vee S_3(x_4, y_4), T(y_4)\\
d_{23} =& d_2 \vee d_3 = R(x_3), S_1(x_3, y_3) \vee S_1(x_5, y_5), S_2(x_5, y_5) \vee S_3(x_4, y_4), T(y_4)\\
d_{123} =& d_1 \vee d_3 = d_1 \vee d_2 \vee d_3 \equiv h_3
\end{aligned}
$$

Two facts are interesting here. First, d_{123} is equivalent to h_3 in Example 4.5, and therefore is hard. The naive algorithm would simply apply the inclusion-exclusion formula and fail when it reaches d_{123}. Second, we have the following equivalence: $d_1 \vee d_3 \equiv d_1 \vee d_2 \vee d_3$; in other words, $d_{13} = d_{123}$ and the two hard queries cancel out in the inclusion-exclusion formula:

$$
\begin{aligned}
P(q_W) =& P(d_1) + P(d_2) + P(d_3) - P(d_1 \vee d_2) - P(d_1 \vee d_3) - P(d_2 \vee d_3) + P(d_1 \vee d_2 \vee d_3)\\
=& P(d_1) + P(d_2) + P(d_3) - P(d_{12}) - P(d_{23})
\end{aligned}
$$

All five queries on the last line have a separator, hence all are in $UCQ(P)$. It

follows that q_W is in $UCQ(P)$, and in order to compute $P(q_W)$ in PTIME we must "cancel out" the two hard terms in the inclusion/exclusion formula.

The algebraic definition of the inclusion-exclusion formula is Mobius inversion formula on lattices. We review here the basic definitions, following Stanley [Stanley 1997]. A finite *lattice* is a finite ordered set $(L, \leq)$ where every two elements $u, v \in L$ have a greatest lower bound $u \bigtriangleup v$ and a least upper bound $u \bigtriangledown v$, called *meet* and *join*. (The standard notations are $\wedge$ and $\vee$, but we reserve these for logical OR and AND operations on queries.) Since the lattice is finite, it has a minimum and a maximum element, denoted $\hat{0}, \hat{1}$. We say that v *covers* u if $u < v$ and there is no w such that $u < w < v$. The *atoms* are all elements that cover $\hat{0}$; the *co-atoms* are all elements covered by $\hat{1}$; u is called *atomic* if it is the join of atoms; u is called *co-atomic* if it is the meet of co-atoms; the entire lattice is atomic (co-atomic) if all elements are atomic (co-atomic).

The Mobius function on the lattice[2] L is the function $\mu_L : L \times L \to \mathbf{Z}$ defined by:

$$\mu_L(u, u) = 1$$
$$\mu_L(u, v) = -\sum_{w: u < w \leq v} \mu_L(w, v)$$

(It follows that, whenever $u \not\leq v$, $\mu(u, v) = 0$.) We drop the subscript and write μ when L is clear from the context.

The Mobius function has the following important property. Let $f : L \to \mathbf{R}$ be any real function defined on the lattice. Define a new function g as $g(v) = \sum_{u \leq v} f(u)$. Then, one can recover f from g by:

$$f(v) = \sum_{u \leq v} \mu(u, v) g(u) \tag{7}$$

Let $Q = d_1 \wedge \ldots \wedge d_k$ be a query in CNF. For any set $s \subseteq [k]$, define $\bar{s} = \{i \mid d_i \Rightarrow \bigvee_{j \in s} d_j\}$. The following three properties follow immediately: $s \subseteq s' \Rightarrow \bar{s} \subseteq \bar{s}'$; $s \subseteq \bar{s}$; $\bar{\bar{s}} = \bar{s}$. Hence, $s \mapsto \bar{s}$ is a closure operator. A set s is *closed* if $\bar{s} = s$. Denote $L(Q)$ the set of closed sets.

Definition 4.6. *Given a UCQ query Q, its* CNF lattice *is $(L(Q), \leq)$, where $L(Q)$ consists of all closed subsets of $[k]$, and $u \leq v$ if $u \supseteq v$.*

The CNF lattice has the following properties:

—Each element $u \in L(Q)$, $u \neq \hat{1}$, represents a disjunctive query $d_u = \bigvee_{i \in u} d_i$. By convention, we denote $d_{\hat{1}} = Q$. Note that the lattice order $u \leq v$ corresponds to reverse implication, $d_v \Rightarrow d_u$.

—$Q \equiv \bigwedge_{u \in L, u \neq \hat{1}} d_u$.

—For all $u, v \in L$, $u \leq v$ iff $d_v \Rightarrow d_u$. In particular, $d_u \equiv d_v$ iff $u = v$.

[2]The standard definition of the Mobius function is on a poset, i.e. one does not need it to be a lattice. But in this paper we are only interested in the Mobius function on lattices.

—For all $u, v \in L(Q)$, $d_{u \triangle v} = d_u \vee d_v$. Here $\triangle$ denotes the meet operation in the lattice (traditionally denoted $\wedge$), while $\vee$ is query disjunction. In other words, the lattice-meet (traditionally written $\wedge$) is the query-or ($\vee$).

—The disjunctive queries $d_1, \ldots, d_k$ in the CNF representation of Q correspond to the co-atoms of $L(Q)$. This follows from our assumption that the CNF representation is minimized: if d_j is not a co-atom, then $d_l \Rightarrow d_j$ for some $l \neq j$, hence d_j it can be removed from the CNF expression $Q = \bigwedge_{i=1,k} d_i$, contradicting the fact that the CNF expression was minimized.

—The lattice is co-atomic.

Proposition 4.7 Mobius inversion formula for queries in CNF. *Let Q be a UCQ, with CNF lattice $(L(Q), \leq)$. Then:*

$$P(Q) = -\sum_{v < \hat{1}} \mu_L(v, \hat{1}) P(d_v) \tag{8}$$

Proof. Denote $R = \neg Q$, and, for every $v \in L$, denote $e_v = \neg d_v$. The following three properties hold: (a) $R \equiv e_{\hat{1}} \equiv \bigvee_{v \in L(Q), v \neq \hat{1}} e_v$; (b) $u \leq v$ iff $e_u \Rightarrow e_v$; (c) for all $u, v \in L(Q)$, $e_{u \triangle v} = e_u \wedge e_v$. We first prove:

$$P(R) = -\sum_{v < \hat{1}} \mu_L(v, \hat{1}) P(e_v) \tag{9}$$

Here we follow [Stanley 1997]. For all $u \in L(Q)$, denote $f(u) = P(e_u \wedge \neg(\bigvee_{v<u} e_v))$. Then:

$$P(e_u) = \sum_{v \leq u} f(v) \qquad \Rightarrow \qquad f(u) = \sum_{v \leq u} \mu_L(v, u) P(e_v)$$

The claim Eq. 9 follows by setting $u = \hat{1}$ and noting $f(\hat{1}) = 0$. From here, Eq. 8 follows from the following three observations: $P(Q) = 1 - P(R)$; for all $v \in L(Q)$, $P(d_v) = 1 - P(e_v)$; and $\sum_{v \in L} \mu(v, \hat{1}) = 0$. □

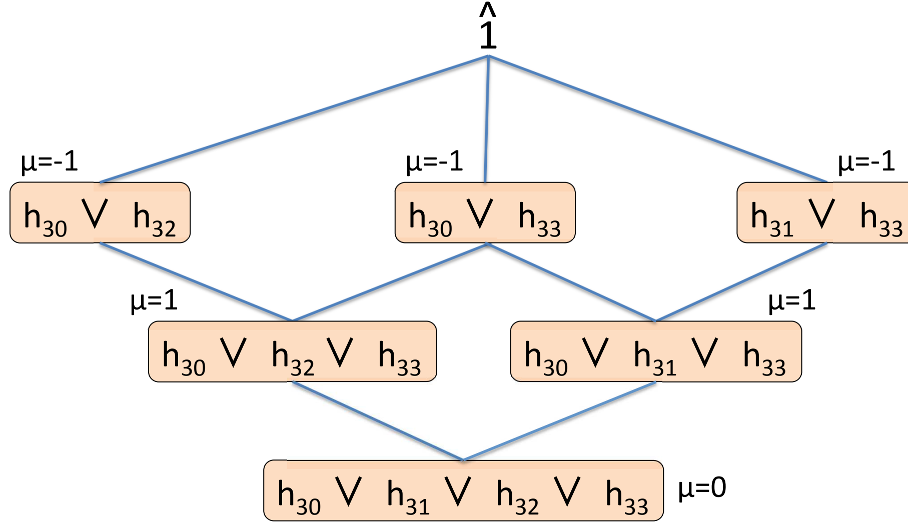
It should be clear that Mobius' inversion formula generalizes inclusion/exclusion.

Therefore, *the second change* that we need to make to algorithm 2 is to replace the inclusion-exclusion formula with Mobius' inversion formula: that is, replace Line 9 with Eq. 8. Notice that when we apply the Mobius' inversion formula we must explicitly avoid the lattice elements whose Mobius function is 0: this is shown in Line 11 of the algorithm. This is a key technical detail that allows us to prove that the algorithm is complete.

Example 4.8. *Consider q_W from the beginning of this section, $q_W = (h_{30} \vee h_{32}) \wedge (h_{30} \vee h_{33}) \wedge (h_{31} \vee h_{33})$, where:*

$$\begin{aligned} h_{30} =& R(x_0), S_1(x_0, y_0) \\ h_{31} =& S_1(x_1, y_1), S_2(x_1, y_1) \\ h_{32} =& S_2(x_2, y_2), S_3(x_2, y_2) \\ h_{33} =& S_3(x_3, y_3), T(y_3) \end{aligned}$$

Note that $h_{30} \vee h_{31} \vee h_{32} \vee h_{33} \equiv h_3$, where h_3 is given in Example 4.5, and is hard for $FP^{\#P}$. Figure 1 shows the CNF lattice for q_W. All five queries other than the

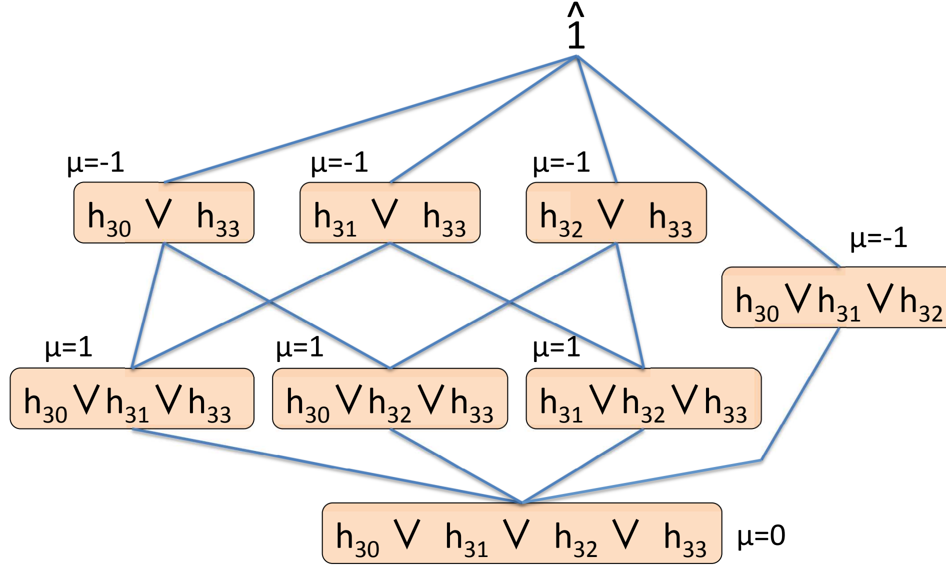


$$q_W = (h_{30} \vee h_{32}) \wedge (h_{30} \vee h_{33}) \wedge (h_{31} \vee h_{33})$$

Fig. 1. The CNF Lattice for q_W; see Example 4.8.

bottom query have a separator, and hence these five queries are in $UCQ(P)$. At the bottom we have h_3, which is not in $UCQ(P)$. The Mobius function $\mu(v, \hat{1})$ is shown for each node: in particular, it is 0 for the bottom element. That means that we can compute $P(q_W)$ using Mobius' inversion formula, by summing over the five queries (which form a W, hence the name of the query): we don't need $P(h_3)$. Notice that the lattice in Figure 1 is not atomic: neither $h_{30} \vee h_{32}$ nor $h_{31} \vee h_{33}$ are atomic. By exploiting this fact [Jha and Suciu 2010], it has been shown that $P(q_W)$ can also be computed in PTIME using a d-DNNF [Darwiche 2000; Darwiche and Marquis 2002], as an alternate technique to Mobius' inversion formula.

Figure 2 shows the CNF lattice of another query, called q_9. The lattice has 9 points (hence the name). Here, too, the query at the bottom of the lattice is h_3, while all other queries are in $UCQ(P)$. The Mobius function at the bottom element is $= 0$, hence we can compute $P(q_9)$ using Mobius' inversion formula without the need to compute $P(h_3)$. Therefore, q_9 is in $UCQ(P)$. The interesting fact about this lattice is that it is both atomic and coatomic, yet it has $\mu(\hat{0}, \hat{1}) = 0$. The technique used in [Jha and Suciu 2010] to derive a polynomial size d-DNNF no



$$q_9 = (h_{30} \vee h_{33}) \wedge (h_{31} \vee h_{33}) \wedge (h_{32} \vee h_{33}) \wedge (h_{30} \vee h_{31} \vee h_{32})$$

Fig. 2. The CNF Lattice for q_9; see Example 4.8.

longer works if the lattice is atomic. In fact, it is conjectured in [Jha and Suciu 2010] that this query does not have a polynomial size d-DNNF. To the best of our knowledge, the only technique to date that can compute $P(q_9)$ in PTIME is Mobius' inversion function, as in algorithm 2.

4.3 The Dichotomy Theorem

We prove now the dichotomy theorem for unions of conjunctive queries: every query is either in $UCQ(P)$, or it is provably hard for $FP^{\#P}$. For the proof of the dichotomy theorem, we use algorithm 2 and Theorem 4.4. We assume throughout this section that the query is ranked: the results immediately extend to unranked queries, since we have shown that every query can be ranked without affecting its membership in $UCQ(P)$.

If Q is a query and R a relational symbol, then $Q[R = \texttt{false}]$ denotes the result of replacing in Q every atom that refers to R with `false`.

Definition 4.9. *Define the following rewrite rule on UCQ, denoted $Q \to Q'$:*

$$\begin{aligned} Q \to & Q[R = \texttt{false}] && \text{where } R \text{ is a relational symbol} \\ Q \to & d_v && \text{where } v \in L(Q) \text{ and } \mu(v, \hat{1}) \neq 0 \\ d \to & d[a/z] && \text{where } z \text{ is a separator and } a \text{ a constant} \end{aligned}$$

Denote $\xrightarrow{*}$ the reflexive and transitive closure of $\to$.

As in algorithm 2, we make the assumption that the query $d[a/z]$ is rewritten such that it does not mention the constant a. This is possible because z is a separator variable: for every symbol R, every atom with the symbol R will have a on the same position, so we simply eliminate that position, decreasing the arity of R by one, for every R.

Definition 4.10. *A query Q is called* unsafe *if there exists a rewriting $Q \xrightarrow{*} Q'$, s.t. Q' is immediately unsafe (Def. 4.3). Otherwise it is called* safe.

For a simple illustration, the following query is unsafe:

$$d = R(z_0, x_0), S(z_0, x_0, y_0) \vee S(z_1, x_1, y_1), T(z_1, y_1)$$

To see this, use the separator $z = z_0 = z_1$ (i.e. write d equivalently as $d[z/z_0, z/z_1]$: now z is a separator), then rewrite $d \to d[a/z]$: the latter is the same as h_1 in Example 4.5.

Theorem 4.11 Dichotomy. *For every UCQ query Q the following holds:*

— *If Q is safe, then $Q \in UCQ(P)$.*
— *If Q is unsafe, then computing $P(Q)$ is hard for $FP^{\#P}$.*

Before proving the theorem we give a technical lemma. Call a rewriting step $Q \to d_v$ *maximal*, if for every u s.t. $v < u$, if $\mu(u, \hat{1}) \neq 0$ then d_u is safe. We call a sequence of rewritings $Q \xrightarrow{*} Q'$ *maximal* if every rewriting step of type $Q \to d_v$ is maximal (there are no restrictions on the other two types of rewritings). Then:

Lemma 4.12. *If $Q \xrightarrow{*} Q'$ is a maximal rewriting, then the computation problem of $P(Q')$ can be reduced in polynomial time to the computation problem of $P(Q)$. In particular, if Q' is hard for $FP^{\#P}$, then Q is hard for $FP^{\#P}$.*

Proof. We prove the lemma by induction on the length of the rewriting. When the length is 0 then it is trivial, so assume the length is > 0 and consider the first rewriting step $Q \to Q''$. There are three cases.

The rewriting is $Q \to Q[R = \texttt{false}]$. By induction, $Q'' = Q[R = \texttt{false}]$ and the computation problem of $P(Q')$ can be reduced to that of $P(Q'')$. On the other hand, one can compute $P(Q'')$ given an oracle for computing $P(Q)$: indeed, given a database D'' for Q'', modify it by setting the probabilities of all tuples in R to 0. This change does not affect $P(Q'')$ (since Q'' does not mention R), and on the new database, $P(Q) = P(Q'')$.

The rewriting is $d \to d[a/z]$. Here $Q = d$ is a disjunctive query. By induction, we know that the computation problem of $P(Q')$ can be reduced to that of $P(d[a/z])$.

The latter further reduced to $P(d)$ as follows. Consider a database D'' for $d[a/z]$. Recall that in the schema for $d[a/z]$, each relation R has one missing attribute, namely the attribute corresponding to the separator variable z: modify D'' by re-inserting that attribute, and setting its value to the constant a in all tuples. Call D the resulting database. It is easy to see that $P_D(d) = P_{D''}(d[a/z])$.

The rewriting is $Q \rightarrow d_u$, where $u \in L(Q)$ and $\mu(u, \hat{1}) \neq 0$. Since the rewriting is maximal, we know that for all queries d_w for $w > u$, either $\mu(w, \hat{1}) = 0$ or d_w is computable in PTIME: we will use this below. By induction, we know that the computation problem for $P(Q')$ can be reduced to $P(d_u)$; we show that the latter can be reduced to $P(Q)$. Assume we have access to an oracle computing $P(Q)$. Let D'' be a probabilistic database for d_u. We construct a new probabilistic database D consisting of D'' and the union of several deterministic databases D_z, one for each $z \in L(Q)$, $u \not\leq z < \hat{1}$. We will define D_z such that $D_z \models d_z$, yet D_z does not affect any query d_w, for $u \leq w$. We start by noting that $u \not\leq z$ implies $d_z \not\Rightarrow d_u$; since d_z is a disjunctive query $d_z = c_1 \vee c_2 \vee \ldots$, where each c_i is a component, there exists i such that $c_i \not\Rightarrow d_u$ (such an i exists because of Sagiv and Yannakakis theorem [Sagiv and Yannakakis 1980], as we explained in Sect. 2). Then define D_z to be the canonical database of c_i. We further ensure that all constants in D_z are distinct from all other constants used in D'' or in other D_z's: this is possible since c_i has no constants, so we can simply replace its variables with fresh constants in the canonical database. By adding D_z to D'' we ensure that the query d_z is true: $D_z \cup D'' \models d_z$. On the other hand, this does not affect any of the queries d_w for $u \leq w$. Indeed, if a valuation maps a component c'_j of d_w to $D_z \cup D''$, then it must map it either entirely to D_z or entirely to D'', because c'_j is connected and D_z, D'' do not share constants: the former is not possible, because it would imply $c_i \Rightarrow c'_j$, which implies $c_i \Rightarrow d_u$ (because $c'_j \Rightarrow d_w$ and $d_w \Rightarrow d_u$), which contradicts $u \not\leq z$. Let $D = D'' \cup \bigcup_{u \not\leq z < \hat{1}} D_z$. We have:

$$\begin{aligned}
P_D(Q) =& P_D(\bigwedge_{w<\hat{1}} d_w) \\
=& P_D(\bigwedge_{u \leq w < \hat{1}} d_w) && \text{because if } u \not\leq z < \hat{1} \text{ then } D \models d_z \\
=& P_{D''}(\bigwedge_{u \leq w < \hat{1}} d_w) && \text{because } d_w \text{ is not affected by any } D_z \\
=& -\sum_{u \leq w < \hat{1}} \mu(w, \hat{1}) P_{D''}(d_w)
\end{aligned}$$

Given $P_D(Q)$, we can compute $P_{D''}(d_u)$ because $\mu(u, \hat{1}) \neq 0$ and for all w s.t. $u < w < \hat{1}$, either $\mu(w, \hat{1}) = 0$ and then we ignore that term, or the probability $P_{D''}(d_w)$ can be computed in PTIME. □

PROOF. of Theorem 4.11. To prove the first item we show that algorithm 2 never gets stuck on a safe query. Indeed, starting with Q, the algorithm processes recursively various queries Q'. We prove that for each such Q', $Q \xrightarrow{*} Q'$. Indeed, when the algorithm narrows to a symbol-component in Lines 3 and 15, then we rewrite $Q \rightarrow Q'$ by setting $R =$ `false` for all relations outside that symbol-component.

When the algorithm computes the probability recursively on a lattice element d_v s.t. $\mu(v, \hat{1}) \neq 0$ (Line 11), then we use the rewrite rule $Q \rightarrow d_v$; and when the algorithm substitutes a separator with a constant (Line 27) then we rewrite $d \rightarrow d[a/z]$. Thus, if the algorithm ever gets stuck, it gets stuck on some Q' s.t. $Q \xrightarrow{*} Q'$; since Q' is immediately unsafe, we conclude that Q is unsafe, contradicting the assumption that Q was safe. Therefore, the algorithm never gets stuck on a safe query.

We prove now the second item. For that, we will prove the following claim: for any unsafe query Q there exists a maximal rewriting $Q \xrightarrow{*} Q'$ s.t. Q' is immediately unsafe. The claim proves the second item, because Theorem 4.4 ensures that Q' is hard, then Lemma 4.12 proves that Q is hard. We prove the claim by induction on the structure of Q. Indeed, suppose Q is unsafe, and suppose that the first step in a rewriting to an immediately unsafe query is $Q \rightarrow d_v$. Consider all elements $u \in L(Q)$ with the properties: $\mu(u, \hat{1}) \neq 0$, and d_u is unsafe: there exists at least one such element, namely v. Choose u to be any maximal element with this property. Since d_u is simpler than Q, by induction hypothesis there exists a maximal rewriting $d_u \xrightarrow{*} Q'$, with Q' immediately unsafe. Then the rewriting $Q \rightarrow d_u \xrightarrow{*} Q'$ is maximal, and proves the claim. If the first step of the rewriting any of the other two kinds, $Q \rightarrow Q''$, then we simply use induction hypothesis on Q'' to argue that there exists a maximal rewriting $Q'' \xrightarrow{*} Q'$, which gives us a maximal rewriting $Q \xrightarrow{*} Q'$. This concludes the proof of the claim and the theorem. □

4.4 Representation Theorem

Finally, we show that testing for zeros of the Mobius function is a necessary step in ensuring that an algorithm is complete. For that we prove the following:

THEOREM 4.13 REPRESENTATION THEOREM. *Let L be any finite lattice. Then there exists a UCQ query Q such that (a) the CNF lattice of Q is isomorphic to L, (b) if $u = \hat{0} \in L$, then the query d_u is unsafe, and (c) if $u \in L$, $u \neq \hat{0}, \neq \hat{1}$, then d_u is safe.*

In general, there are many queries Q for which $L(Q)$ is isomorphic to a given lattice L. The query stated by the theorem is one that has the special property that $P(Q)$ is in PTIME iff $\mu_L(\hat{0}, \hat{1}) = 0$. In other words, the theorem says that there is no substitute to checking $\mu_L(\hat{0}, \hat{1}) = 0$.

PROOF. Call an element $r \in L$ *join irreducible* if whenever $v_1 \bigtriangledown v_2 = r$, then either $v_1 = r$ or $v_2 = r$. (Recall that $\bigtriangledown$ is join: traditionally written $\vee$ in lattice theory.) This includes all atoms (since every atom is join irreducible), and possibly more elements. Let $R = \{r_0, r_1, \ldots, r_k\}$ be all join irreducible elements in L. For every $u \in L$ denote $R_u = \{r \mid r \in R, r \leq u\}$, and note that $R_{u \triangle v} = R_u \cup R_v$. Define the following components (see also Example 4.8):

$$\begin{aligned} h_{k0} &= R(x_0), S_1(x_0, y_0) \\ h_{ki} &= S_i(x_i, y_i), S_{i+1}(x_i, y_i) \quad i = 1, k-1 \\ h_{kk} &= S_k(x_k, y_k), T(y_k) \end{aligned}$$

Define $Q = \bigwedge_{u < \hat{1}} \bigvee_{r_i \in R_u} h_{ki}$. Then the query at $\hat{0}$ is $\bigvee_i h_{ki} = h_k$ and is unsafe; any query at some $u \neq \hat{0}$ misses at least one component of h_k, and thus is safe. This proves the claim. □

Both queries q_W and q_9 were obtained by this construction from the lattices in Fig. 1 and Fig. 2. Note that the query Q defined in the proof is given in a CNF expression which is not necessarily minimized. This is needed in order to ensure that $L(Q)$ is isomorphic to L. If one minimizes Q first, then $L(Q)$ is isomorphic to the set of co-atomic elements of L, which form a meet-sublattice, denote it L_0. Thus, algorithm 2 operates on the lattice L_0 rather than L. However, the test $\mu_L(\hat{0}, \hat{1}) = 0$ is still unavoidable in order to check if $P(Q)$ is in PTIME, because of the following two properties, which can be proved using standard lattice-theoretic techniques [Stanley 1997]: (a) if $\hat{0} \not\in L_0$ then $\mu_L(\hat{0}, \hat{1}) = 0$; in that case $P(Q)$ is in PTIME (because all elements in L_0 are safe queries), and (b) if $\hat{0} \in L_0$ then $\mu_L(\hat{0}, \hat{1}) = \mu_{L_0}(\hat{0}, \hat{1})$; in that case $P(Q)$ is in PTIME iff $\mu_L(\hat{0}, \hat{1}) = 0$.

5. OUTLINE OF THE HARDNESS PROOF

In the rest of the paper we prove Theorem 4.4. The proof has three parts:

Query Leveling. We assign to each relation attribute a "type" s.t. every join is between the same type. We call such a type a "level". Call a query "leveled" if all attributes in a relation symbol are on distinct levels. This prevents two distinct attributes of a symbol to join, even indirectly. For example, $R(x, y), R(y, z)$ is not leveled, since both attributes of R must have the same type. In Sect. 6 we prove that if a query d is immediately unsafe, then it can be transformed into a leveled query d' that is immediately unsafe, and the computation problem for $P(d')$ can be reduced to that for $P(d)$. Thus, it suffices to prove hardness for all leveled queries, and we restrict the discussion to leveled queries in the rest of the proof.

Rewriting to forbidden queries. If a query is unsafe, then the rewrite rules in Def. 4.9 allow us to simplify it. But if d is immediately unsafe, then none of those rules further applies. To further simplify a query d, we introduce a new rewrite step, where we replace all variables on a level with constants. If d' is the resulting query, then it is not difficult to prove that the computation problem for $P(d')$ can be reduced to that for $P(d)$. The difficulty is to choose the level such that, after the rewriting, d' is still unsafe. This is only possible for leveled queries: for example it is not possible for $R(x, y), R(y, z)$ because there all attributes have the same level, and substituting that level with constants leads to a safe query (without variables). This explains why we leveled the query in the first part. In Sect. 7 we prove that, if d is leveled and immediately unsafe, then, as long as it has three or more levels, one can choose a level such that, after substituting it with constants, the new query d' is also unsafe. This allows us to simplify the query until it has only two levels. The proof of this step is rather subtle, because choosing the right level to substitute such that the resulting query is still unsafe requires a careful case analysis.

Hardness of forbidden queries. Call a 2-leveled query that is immediately unsafe, a *forbidden query*. Based on the previous two parts, it suffices to prove hardness for forbidden queries, which we do in Sect. 8: we show that every forbidden query is hard for $FP^{\#P}$. This is by far the most difficult part of the hardness proof. Our reduction is from the *partitioned, positive 2CNF* problem, which was shown to be #P-hard by Provan and Ball [Provan and Ball 1983]. We show that if d is any forbidden query, then one can solve the pp-2CNF problem with a polynomial-time

Turning Machine with an oracle for $P(d)$. The proof is difficult because the general structure of a forbidden query is quite different from that of the pp-2CNF problem.

6. QUERY LEVELING

Fix a relational vocabulary. An *attribute* is a pair (R, i), where R is a relation name and $i \in [arity(R)]$. Denote *Attr* the set of all attributes in the vocabulary.

Let d be a disjunctive query, $d = c_1 \vee c_2 \vee \ldots$ We usually call c_i a "component", but in this section we will prefer the longer term "connected conjunctive query", or just "conjunctive query". Construct the following undirected graph, called the *attribute graph of d*. The nodes are *Attr*, and the edges are pairs $((R, i), (S, j))$ s.t. there exists a conjunctive query c_k that contains two atoms $R(\ldots)$ and $S(\ldots)$ s.t. the same variable x occurs in position i in the first atom and in position j in the second atom. In other words, two attributes are connected in the graph if they are joined in some of the queries c_k.

Definition 6.1. *A* level *of d is a connected component in the attribute graph of d.*

Thus, a level is a set of attributes, and every attribute belongs to exactly one level Z. By some abuse of notation we say that a variable x belongs to a level Z, and write $x \in Z$, when x occurs in an attribute position that is at the level Z. Clearly a variable x belongs to exactly one level. In other words, one can view the levels as a partition of the set of variables, and each level Z as a set of variables.

We start by establishing the connection between a separator and a level. Recall that $d = c_1 \vee c_2 \vee \ldots$ is called *symbol-connected* if its co-occurrence graph is connected Def. 3.4. A variable x in c_i is a *root variable* if it occurs in all atoms of c_i.

Proposition 6.2. *Let d be a disjunctive query that is symbol-connected. Then d has a separator iff there exists a level Z that contains only root variables.*

Proof. To prove the "if" direction, assume Z is a level that contains only root variables. Then Z contains at least one variable from each component c_i, because the query is symbol-connected; also, Z contains at least one attribute from each relational symbol R. We prove that it contains exactly one attribute from each symbol R. For each R, let i_R be the smallest attribute in Z: that is, $(R, i_R) \in Z$, and, if $(R, j) \in Z$ then $i_R \leq j$. Call (R, i_R) the minimal Z-attribute in R. We claim that if a minimal Z-attribute (R, i_R) is connected to some other attribute (S, j) in the attribute graph, then (S, j) is also a minimal Z-attribute. Indeed, consider a component containing $R(\ldots), S(\ldots)$ and a common variable x on position i_R in R and on position j in S. Clearly x is a root variable. Suppose $i_S < j$: then the variable y on position i_S in S is also a root variable, and must also occur in R, on some position i, and obviously $(R, i) \in Z$. Because the query is ranked (Def. 4.1), we must have $i < i_R$, contradicting the fact that i_R was minimal. Thus, Z contains exactly one attribute in each relation, and exactly one root variable in each component. We prove now that it is a separator. Let x_i be the root variable in c_i on level Z. Write d as $\exists z.(c_1[z/x_1] \vee c_2[z/x_2] \vee \ldots)$. Now z is a separator variable: it is clearly a root variable, and in every atom with relation symbol R, the variable z occurs precisely on the attribute that is at the level Z. This proved

the "if" direction. For the "only if" direction, write d as $Q = \exists z.(q_1 \vee q_2 \vee \ldots)$ where z is a separator variable. Let Z be the set of attributes where z occurs: we prove that Z is a level, and that it satisfies the proposition. First, Z is a level, because if two attributes (R, i) and (S, j) join, and (R, i) is in Z, then every atom with symbol R contains z on position i: since it joins with (S, j), the atom with symbol S also contains z on position j, hence (S, j) is in Z. Obviously, Z contains only root variables (namely z). $\square$

In the rest of the paper we will use the criteria in Prop. 6.2 as the definition of a separator in a leveled query, instead of the official Def. 3.2.

Definition 6.3. *A disjunctive query is called* leveled *if every level Z contains at most one attribute from every relational symbol R.*

Example 6.4. *All queries h_k in Example 4.5 are leveled. Queries q_W, q_9 in Fig. 1, Fig. 2 are leveled.*

The following two queries are not leveled:

$$d_1 = R(x, y), R(y, z)$$
$$d_2 = R(x, y), S(y, z) \vee R(x', y'), S(x', y')$$

The main result in this section is:

Theorem 6.5 Leveling. *If d is a disjunctive query that is immediately unsafe, then there exists a leveled, disjunctive query d' s.t. d' is also immediately unsafe, and the computational problem for $P(d')$ can be reduced to that of $P(d)$.*

In the rest of the section we prove Theorem 6.5. We start by constructing the leveled query. Fix a number $L > 0$, call a number $t \in [L]$ a *level number*. Define a new vocabulary, called the *L-vocabulary*, where each symbol is a pair (R, τ), denoted R^τ, where R is a relation symbol in the original vocabulary, and $\tau \in [L]^{arity(R)}$, is a sequence of *distinct* numbers, called *level numbers*. Thus, there are $L!/(arity(R))!$ relation names in the new vocabulary from each relation name R in the old vocabulary. Each attribute i of R^τ is associated with a number τ_i, called the level number of that attribute.

Consider a connected, conjunctive query c' over the L-vocabulary. We call this query *well annotated* if each of its variables can be mapped to a level number $t \in [L]$, such that the variable occurs only on attribute positions with level number t. A disjunctive query $d' = c'_1 \vee c'_2 \vee \ldots$ is well annotated if all its conjunctive queries are. The following is straightforward:

Proposition 6.6. *If a disjunctive query is well annotated, then it is leveled.*

Indeed, if the query is well annotated then every level consists of attributes with the same level number. (The converse does not hold in general.) The claim follows from the fact τ in R^τ is a sequence of *distinct* level numbers.

Start from a connected conjunctive query c. We will associate to it several well annotated queries c^T over the L-vocabulary, as follows. Let $T : Vars(c) \rightarrow [L]$ be a function that associates level numbers to variables, s.t. whenever x, y co-occur in a common atom, then $T(x) \neq T(y)$. Define c^T as follows: for each atom $R(x_1, x_2, \ldots)$ in c, there is one atom $R^{T(x_1)\cdot T(x_2)\cdots}(x_1, x_2, \ldots)$ in c'. In other words, c^T has the

same atoms as c, except that the relation symbols are now annotated with level sequences. Clearly, c^T is well annotated.

DEFINITION 6.7. *The* leveling *of a connected, conjunctive query* c *is* $d' = \bigvee_T c^T$, *there* T *ranges over all functions associating level numbers to the variables of* c. *The* leveling *of a disjunctive query* $d = c_1 \vee c_2 \vee \ldots$ *is* $d' = d'_1 \vee d'_2 \vee \ldots$ *where* d'_i *is the leveling of* c_i.

To prove Theorem 6.5 we show two properties: the computation problem for $Pd')$ can be reduced to that of $P(d)$, and if d is immediately unsafe, then d' is also immediately unsafe.

EXAMPLE 6.8. *We illustrate the main idea on* d_1, d_2 *in Example 6.4. Choosing* $L = 4$ *for both* d_1 *and* d_2, *we could construct the following levelings:*

$$d'_1 = R^{12}(x^1, y^2), R^{23}(y^2, z^3) \vee R^{23}(x^2, y^3), R^{34}(y^3, z^4)$$
$$d'_2 = R^{23}(x^2, y^3), S^{34}(y^3, u^4) \vee R^{12}(x^1, y^2), S^{23}(y^2, z^3) \vee R^{23}(x'^2, y'^3), S^{23}(x'^2, y'^3)$$

We have indicated in a superscript the level annotation for each variable, making it easy to check that both queries are well annotated. This is more frugal leveling than that defined by Def. 6.7, since it does not include many possible annotations, e.g. R^{13}, R^{21}, R^{31}, *etc. But in these examples, the two main properties still hold. Neither* d'_1, d'_2 *have a separator. Furthermore, the reader may check that one can use an oracle for* $P(d_1)$ *in order to compute* $P(d'_1)$ *(but not in the other direction); and similarly for* d_2, d'_2. *Notice that the choice of* L *matters: it has to be "large enough". If we choose* $L = 3$ *for leveling* d_1, *then* $d'_1 = R^{12}(x^1, y^2), R^{23}(y^2, z^3)$ *has the separator* y^2.

PROPOSITION 6.9. *Let* d' *be the leveling of* d. *The computation problem for* $P(d')$ *can be reduced to that of* $P(d)$.

PROOF. We use an oracle for $P(d)$ in order to compute $P(d')$. Let D' be a database over the L-vocabulary. For every $t \in [L]$, let $ADom_t$ be the set of all constants in D that occur in some attribute with level number t. W.l.o.g. we can assume that the domains $ADom_1, ADom_2, \ldots$ are disjoint. Indeed, suppose some active domain $ADom_t$ shares some constants with some other active domain. Then we re-map all constants in $ADom_t$ to a new domain $ADom'_t$, disjoint from all others, by using a bijective function $f : ADom_t \to ADom'_t$. More precisely, we modify the database D' to D'', by replacing every value $a \in ADom_t$ of an attribute with level number t with the new value $f(a) \in ADom'_t$, and leaving all other attribute values unchanged. Since d' is well annotated, we have $D' \models d'$ iff $D'' \models d'$; moreover, if D' is a probabilistic database, then so is D'' and $P_{D'}(d') = P_{D''}(d')$. Thus, we can assume w.l.o.g. that $ADom_1, ADom_2, \ldots$ are disjoint. Construct a new database D over the original vocabulary, by simply erasing the level number annotation of all tuples in D'. The tuples in D are in 1-1 correspondence with those in D', because no two tuples can become the same after erasing their level annotation, since attributes with different level annotations have different active domains. Furthermore, we prove that $D \models d$ iff $D' \models d'$. The "if" direction is straightforward: any valuation from a component c^T of d' to D', it is also a valuation of the corresponding component c to D. For the "only if" direction,

consider a valuation from a component c to D. It maps every atom in c to some tuple in D. Every variable x in c must be mapped to some $ADom_t$: define $T(x) = t$. It is easy to see that this valuation is also a valuation from c^T to D'. □

PROPOSITION 6.10. *Let d be a disjunctive query that is immediately unsafe (i.e. symbol-connected, and without separators, see Def. 4.3). Then there exists $L > 0$ such that the L-leveling d' of d has a symbol-component that is also immediately unsafe.*

PROOF. Let $d = c_1 \vee c_2 \vee \ldots$ and $d' = c'_1 \vee c'_2 \vee \ldots$. Let G be the co-occurrence graph of $c_1, c_2, \ldots$, and G' the co-occurrence graph of $c'_1, c'_2, \ldots$ By assumption G is connected, but G' is not necessarily connected[3]. Let K be any connected component in G' that has a separator. That is, K consists of a subset of $d'_1, d'_2, \ldots$ that is symbol-connected, and their disjunction has a separator. By Prop. 6.2 there is a level Z' s.t. each variable on level Z' is a root variable.

Let Z be the image of Z' in d: that is Z contains all attributes (R, i) for which there exists τ s.t. (R^τ, i) in Z'. We will prove that Z contains only root variables: then, by Prop. 6.2, d has a separator, which contradicts our assumption. Hence the component K cannot have a separator, proving the proposition.

To check the conditions of Prop. 6.2, start by showing that Z is a level: indeed, if $(R, i) \in Z$, because of $(R^\tau, i) \in Z'$, and is connected in the attribute graph to another attribute (S, j) because there exists a conjunctive query c_i containing the atoms $R(\ldots x \ldots)$ and $S(\ldots x \ldots)$, then we can level c_i to c_i^T such that the leveling annotation of the first atom becomes R^τ (hence c^T belongs to the connected component K); then the second atom will receive some annotation S^σ, and from $(S^\sigma, j) \in Z'$ we obtain that $(S, j) \in Z$. Thus, Z is a level.

Finally, we check that every variable on level Z is a root variable. Let (R, i) be an attribute in Z, and consider a query c_i with an atom R. Let x be the variable on position i. There must exists some attribute $(R^\tau, i) \in Z'$, and we define a function $T : Vars(c_i) \rightarrow [L]$ s.t. the atom R in c_i becomes R^τ in c_i^T (to be able to construct T we assume L is large enough). Then R^τ contains x on position i, hence x must be a root variable in c_i^T, hence it is a root variable in c_i. □

7. REWRITING TO A FORBIDDEN QUERY

From now on we will assume that the queries are leveled. If a query has L levels, then we call it an L-leveled query.

If a query is unsafe, then the rewrite rules in Def. 4.9 allow us to simplify it. But if d is immediately unsafe, then none of those rules further applies. To further simplify a query d, we introduce in this section a new rewrite step, where we replace all variables on a level with constants, resulting in a new unsafe query d'. As long as d has at least three levels, then either this rule, or one of the rules in Def. 4.9 allows us to further simplify it. At the end we reach a query with only two levels, which we call a forbidden query:

DEFINITION 7.1. *A* forbidden query *is a 2-leveled, disjunctive query that is immediately unsafe.*

[3]For a trivial example, the 2-leveling of $d = R(x)$ is $d' = R^1(x^1) \vee R^2(x^2)$ which is not symbol-connected.

We describe now the new rewriting. Let $d = c_1 \vee c_2 \vee \ldots$ be a disjunctive query. Recall that each c_i is a connected, conjunctive query, which we call a *component*. As usual, we assume d to be minimized. Let Z be a level, and denote $Vars_Z(c_i)$ the set of variables in c_i that are in level Z; call them Z-variables. Since the query is leveled, for any fixed Z, each atom contains 0 or 1 Z-variables. Let $A = \{a_1, a_2, \ldots\}$ be a set of constants (A is at most as large as the number of variables in d). Denote:

$$\begin{aligned}
\Theta_Z(c_i, A) =& \{\theta \mid \theta : Vars_Z(c_i) \to A\} \\
c_i[A/Z] =& \bigvee_{\theta \in \Theta_Z(c_i,A)} c_i[\theta] \\
d[A/Z] =& \bigvee_i c_i[A/Z]
\end{aligned} \tag{10}$$

$\Theta_Z(c_i, A)$ is the set of substitutions of Z-variables, and $d[A/Z]$ is obtained by substituting the Z-variables in all possible ways with constants from A.

Our new rewrite step is the following: choose a level Z and rewrite d to $d[A/Z]$. It is easy to prove that, for any Z, A, if $d[A/Z]$ is hard, then d is hard; the proof is similar to Lemma 4.12. There we showed that, if $d[a/z]$ is hard then d is hard, and this can be extended to the case when Z is a level and A is a set of constants. We omit the proof.

The difficulty is to choose the level Z such that $d[A/Z]$ is still unsafe. In fact, if d is not leveled, then no such level exists: for example, in $d = R(x, y, z), R(y, z, u)$ there is a single level containing all three attributes of R, and for any set of constants A, $d[A/Z]$ consists only of ground tuples, hence is safe. This was the reason why we leveled the query. Even if d is leveled, we cannot choose Z arbitrarily. The main result in this section is:

THEOREM 7.2. *Let d be a disjunctive query, with ≥ 3 levels, which is immediately unsafe. Then there exists a level Z and set of constants A (no larger than the set of variables in d) such that $d[A/Z]$ is unsafe.*

We make the same assumptions about $d[A/Z]$ as we made about $d[a/z]$ in algorithm 2 and in Def. 4.9: $d[A/Z]$ is not a query with constants, but instead it is a query without constants over a new vocabulary. If R is a relation name and Z contains one attribute of R, then we create new relation names $R_1, R_2, \ldots$, one for each constant $a_1, a_2, \ldots$ in A, and replace each atom $R(\ldots a_i \ldots)$ with R_i, removing the constant a_i and thus decreasing the arity by 1. Since Z is a level, all atoms with symbol R will have some a_i on that attribute position, hence we no longer have the old symbol R in the rewritten query, only the new symbols $R_1, R_2, \ldots$

In the rest of the section we prove Theorem 7.2. But first, let us see how the above results lets us reduce any query to a forbidden query.

COROLLARY 7.3. *Let d be an unsafe query. Then, there exists a forbidden query q which is polynomial time reducible to d.*

PROOF. The idea is to start from d and repeatedly apply either the rules in Def. 4.9 or the rewrite step in Theorem 7.2, as long as there are at least 3 levels. We prove that the process terminates: then we arrive at a query that has at most 2 levels, i.e., a forbidden query.

Suppose, on the contrary, the process never terminates. Then, there must exist an infinite sequence of queries

$$d_0 \rightarrow d_1 \rightarrow d_2 \rightarrow \cdots$$

such that each is obtained from the previous either by one of the rewrite rules in Def. 4.9 or is a step that replaces d with $d[A/Z]$. There must be infinitely of the latter kind of steps, so assume w.l.o.g. that the sequence $d_0, d_1, \ldots$ includes only the queries obtained after a $d[A/Z]$ step. Let l be the maximum arity of the set of relational symbols in d_0. Any rewriting $d[A/Z]$ takes a set of relational symbols, and for each relation R in the set, it replaces all occurrences of R with new relations of strictly smaller arity. Thus, the arity of any symbol in any query in the sequence is at most l. Consider a function seq from queries to $\mathbb{N}^l$ such that $seq(q)$ is a sequence of non-negative integers with i^{th} integer is equal to the number of distinct relational symbols in q with arity exactly i. Also, consider the lexicographical order $>$ on $\mathbb{N}^l$, where, for $a, b \in \mathbb{N}^l$, the order between a and b is determined by the order of numbers in the largest index where they differ. Since a rewriting replaces all occurrences of a relational symbol R with a set of relations of smaller arity, it follows that:

$$seq(d_0) > seq(d_1) > seq(d_2) > \cdots$$

Also, since the natural order is a well-order for $\mathbb{N}$, the lexicographical order is a well-order for $\mathbb{N}^l$, which follows a known property of lexicographical orders. Also, we know that in a well-ordered set, there cannot be an infinite sequence of strictly decreasing elements. This proves that the sequence of rewritings must terminate after finite number of steps, and result in a forbidden query. □

Note that the corollary holds only if we substitute with constants an entire level: this guarantees that at least one relation of arity k is replaced completely with relations of arities $k-1$. If we don't substitute all variables on a level with constants, then the rewriting process may never terminate.

Now, we return to Theorem 7.2. More precisely, we prove the following: given the assumptions in the theorem, there exists Z, A s.t. in the (minimized) CNF expression $d[A/Z] = \bigwedge_l d'_l$, there exists l s.t. d'_l has a symbol-component without a separator d'. Then Def. 4.9 gives us the following two rewrite steps $d[A/Z] \rightarrow d'_l \xrightarrow{*} d'$: the first step replaces $d[A/Z]$ with d'_l which is a co-atom in its CNF lattice (recall that for any coatom, we have $\mu = -1$); the second step sets $R = \texttt{false}$ for all symbols that are not in the symbol-component d'. Furthermore, d' is immediately unsafe. Hence, $d[A/Z]$ is unsafe, which proves the theorem. In the rest of the section we will prove that one can always find such Z and A.

We illustrate here the main idea, and also show why we need sets A with cardinality larger than 1:

Example 7.4. *Consider:*

$$d = R(x, z_1), S(x, y_1, z_1), S(x, y_2, z_2), U(x, z_2) \vee S(x, y, z), T(y) \vee R(x, z), U(x, z)$$

The query is immediately unsafe, hence none of the rewriting steps in Def. 4.9 applies. To further simplify d, we use the level $Z = \{z_1, z_2, z\}$, and rewrite $d \rightarrow$

$d[A/Z]$. *However, we cannot use a single constant, because then* $d[a/Z]$ *minimizes:*

$$\begin{aligned} d[a/Z] =& R(x,a),S(x,y_1,a),S(x,y_2,a)U(x,a) \vee S(x,y,a),T(y) \vee R(x,a),U(x,a) \\ =& S(x,y,a)T(y) \vee R(x,a)U(x,a) \end{aligned}$$

Here $d[a/Z]$ *is safe. Instead, we use two constants,* $A = \{a,b\}$*:*

$$\begin{aligned} d[A/Z] =& R(x,a),S(x,y_1,a),S(x,y_2,b),U(x,b) \vee R(x,b),S(x,y_1,b),S(x,y_2,a),U(x,a) \vee \\ & S(x,y,a),T(y) \vee S(x,y,b),T(y) \vee R(x,a),U(x,a) \vee R(x,b),U(x,b) \end{aligned}$$

Now $d[A/Z]$ *has no separator; in fact, it is immediately unsafe.*

7.1 Properties of $d[A/Z]$

We start by describing the structure of $d[A/Z]$. Let c_i be a component and Z a level. Define the Z-subcomponents of c_i to be the connected components of the following graph: the nodes are the atoms in c_i, and edges are pairs of atoms that share at least one variable that is not in Z. Denote $sc_Z(c_i) = \{s_1, \ldots, s_m\}$ the Z-subcomponents of c_i. In other words, if we substitute the Z-variables with constants, then c_i will be decomposed into several connected components: these are precisely the Z-subcomponents. Formally, for $\theta \in \Theta_Z(c_i, A)$:

$$c_i[\theta] = \bigwedge_{s \in sc_Z(c_i)} s[\theta] \tag{11}$$

and each $s[\theta]$ is a separate connected component. Now we will write $d[A/Z]$ in CNF. Starting from Eq. 10:

$$d[A/Z] = \bigvee_{i,\theta} c_i[\theta] = \bigvee_{i,\theta} \bigwedge_{s \in sc_Z(c_i)} s[\theta] \tag{12}$$

Here and in the sequel, the disjunction $\bigvee_{i,\theta}$ is taken over all pairs i, θ, where $\theta \in \Theta_Z(c_i, A)$. Next, we convert this DNF expression to CNF, as follows. For each pair i, θ, choose one component $s \in sc_Z(c_i)$; there are $M = \prod_i |\Theta_Z(c_i, A)|$ choices. For each choice $k = 1, 2, 3, \ldots, M$, define the following disjunctive query:

$$d'_k = \bigvee_{i,\theta} s[\theta] \tag{13}$$

Then the CNF expression is $d[A/Z] = \bigwedge_k d'_k$. However, this expression is not minimized: the minimized expression contains only a subset of the d_k's, and these may be further minimized.

Throughout this section we will denote d'_k a disjunctive query in Eq. 13, keeping in mind that it may not necessarily occur in the minimized CNF expression. Note that the logical implication $d[A/Z] \Rightarrow d'_k$ holds, for every k. We also denote d'_l a disjunctive query that *is* part of the minimized CNF expression $d[A/Z] = \bigwedge_l d'_l$.

We give now a technical lemma, which shows that $d[A/Z]$ preserves any single sub-component: if $s \in sc_Z(d_i)$ is a sub-component then the lemma says that we will still find s in the minimized CNF expression of $d[A/Z]$. For example, if s plays a role in making d unsafe, then we can use the lemma to argue that $d[A/Z]$ is unsafe. While we use the lemma in this form only once, we will later extend the proof technique to prove that sets of subcomponents $s_1, s_2, \ldots$ occur together in the minimized CNF expression, and thus we will refer back to this proof.

Lemma 7.5. *Let $\theta \in \Theta_Z(c_i, A)$ be an injective function (hence A needs to be large enough for such a function to exists), and $s \in sc_Z(c_i)$ a subcomponent, and consider the minimized CNF expression $d[A/Z] = \bigwedge_l d'_l$. Then there exists l such that the disjunctive query d'_l has a component isomorphic to $s[\theta]$.*

Proof. We start by showing that there exists a disjunctive query d'_k given by Eq. 13 with the following properties: (1) $s[\theta]$ is a component of d'_k, and (2) for any other component $s'[\theta']$ in d'_k, the logical implication $c_i[\theta] \Rightarrow s'[\theta']$ does not hold: equivalently, there is no homomorphism $s'[\theta'] \rightarrow c_i[\theta]$. In particular $s[\theta] \not\Rightarrow s'[\theta']$ (because $s[\theta]$ is a component of $c_i[\theta]$, see Eq. 11), hence, after minimizing d'_k, it still contains $s[\theta]$. Recall that each d'_k is determined by choosing, for every pair j, θ', one subcomponent $s' \in sc_Z(c_j)$. We choose as follows. For the pair i, θ we will choose s: thus, d'_k contains $s[\theta]$. For every other pair j, θ' we choose $s' \in sc_Z(d_j)$ s.t. there is no homomorphism $s'[\theta'] \rightarrow c_i[\theta]$. We claim that this is always possible. Otherwise, if for all $s' \in sc_Z(d_j)$, there exists a homomorphism, then we obtain a homomorphism $c_j[\theta'] \rightarrow c_i[\theta]$ (because of Eq. 11): since θ is injective, this means that there exists a homomorphism $c_j \rightarrow c_i$, contradicting the fact that the original query d was minimized. We have shown that for each pair j, θ' there exists some $s' \in \Theta_Z(c_j)$ s.t. there is no homomorphism $s'[\theta'] \rightarrow s[\theta]$: this is the s' that we choose to include in d'_k, completing the construction of d'_k.

Let $d[A/Z] = \bigwedge_l d'_l$ be the minimized CNF expression for $d[A/Z]$; d'_k is not necessarily one of the d'_l, because d'_k may disappear during the minimization. The following logical implications hold obviously:

$$c_i[\theta] \Rightarrow \bigwedge_l d'_l \Rightarrow d'_k$$

From Prop. 2.2 we deduce that there exists l such that:

$$c_i[\theta] \Rightarrow d'_l \Rightarrow d'_k$$

Using Sagiv and Yannakakis' containment criteria, we find a component c'_{lj} in d'_l and a component $s'[\theta']$ in d'_k s.t.:

$$c_i[\theta] \Rightarrow c'_{lj} \Rightarrow s'[\theta']$$

Because of the way we constructed d'_k, the only component $s'[\theta']$ that can be logically implied by $c_i[\theta]$ is $s[\theta]$, hence we have:

$$c_i[\theta] \Rightarrow c'_{lj} \Rightarrow s[\theta]$$

Next we write $c_i[\theta]$ as a conjunction of components, $\bigwedge_{s' \in sc_Z(c_i)} s'[\theta]$ (Eq. 11), and since both c'_{lj} and $s[\theta]$ are components themselves, we conclude that there exists a subcomponent $s' \in sc_Z[c_i]$ s.t.

$$s'[\theta] \Rightarrow c'_{lj} \Rightarrow s[\theta]$$

But the only subcomponent s' s.t. $s'[\theta] \Rightarrow s[\theta]$ is $s' = s$: otherwise, if there exists a different component s', then the query c_i could be further minimized (by removing the subcomponent s), contradicting the fact that d was minimized.

Thus, we have proven that $c'_{lj} \equiv s[\theta]$, which proves the lemma. □

7.2 Preliminary: No Root Variable

We first prove Theorem 7.2 when d contains some component c_i that has no root variable. Here we use Lemma 7.5 to prove that the minimized CNF for $d[A/Z]$ also contains a query without a root variable, hence is unsafe.

Given a variable $x \in Vars(c_i)$, let $at(x)$ denote its set of atoms. Let x be a variable for which $at(x)$ is a maximal set; $at(x)$ does not contain all atoms in c_i, because x is not a root variable. Since c_i is connected, there exists some other variable y s.t. all three sets $at(x) \cap at(y)$, $at(x) - at(y)$ and $at(y) - at(x)$ are nonempty. (A query with this property is called non-hierarchical in [Dalvi and Suciu 2004; Dalvi and Suciu 2007b]). Choose Z to be any level distinct from the two levels containing x and y: there exists such a level since we assumed that d has at least three levels[4]. Choose a set of constants A as large as $Vars(c_i)$. The variables x, y will occur in the same subcomponent of c_i, i.e. there exists $s \in sc_Z(c_i)$ s.t. $x, y \in Vars(s)$. Let $\theta \in \Theta_Z(c_i, A)$ be any injective substitution. Then the component $s[\theta]$ has no root variable: this is because x is not a root variable, since $at(x)$ does not contain $at(y)$, and no other variable u can have $at(u)$ strictly larger than $at(x)$. By Lemma 7.5 there exists a conjunct d'_l in the minimized CNF expression for $d[A/Z]$ that contains $s[\theta]$ as a component. Then, obviously, the symbol-component to which $s[\theta]$ belongs has no separator, proving Theorem 7.2 for this case.

In the rest of the section we prove Theorem 7.2, assuming that each component has a root variable.

7.3 Case 1: Non-splitting Level

Call a level Z non-splitting for c_i if $|sc_Z(c_i)| = 1$. If Z splits c_i, i.e. $|sc_Z(c_i)| > 1$, then c_i has exactly one root variable, and that is on level Z. This is because c_i must have a root variable (as we assumed), and if it has at least one root variable on a level other than Z, then that variable will continue to make c_i connected even if we substitute Z with constants.

Call a level Z *non-splitting* if for all i, Z is non-splitting for c_i. Obviously, if Z is non-splitting, then $d[A/Z]$ is a disjunctive query (since each $c_i[\theta]$ in Eq. 12 is a connected conjunctive query).

In this section we prove Theorem 7.2 in the case when there exists a non-splitting level. We pick any non-splitting level Z, and prove our claim for $d[A/Z]$.

LEMMA 7.6 CASE 1. *Let Z be any non-splitting level. Let A be a set of constants s.t. $|A| \geq |Var_Z(c_i)|$, for all i. Then $d[A/Z]$ is a disjunctive query, and it has a symbol-component d' s.t. d' has no separator.*

PROOF. For any substitution $\theta \in \Theta_Z(c_i, A)$, $c_i[\theta]$ is connected, and therefore $d[A/Z]$ given by Eq. 12 is a disjunctive query by definition. Moreover, from Lemma 7.5 we know that for any injective substitution $\theta_i : Vars_Z(c_i) \to A$, $C_i[\theta]$ is not redundant.

Let K be any symbol-component in $d[A/Z]$. (We have assumed that d is symbol-connected, but $d[A/Z]$ may not be.) We show that K has no separator. Suppose

[4] It is entirely possible that Z has no attributes in c_i. For example c_i may be $R(x), S(x, y), T(y)$, while Z contains attributes from other relations. We still make progress simplifying d to $d[A/Z]$.

it had a separator: then there exists a level V' that contains only root variables: then we show that the "corresponding level" in d also contains only root variables, showing that d has a separator. To define the "corresponding level", let V be the set of all attributes (R, i) s.t. there exists an attribute $(R_a, i) \in V$, where R_a is a symbol occurring in the symbol-component K. Here R_a denotes the new symbol representing R with the constant a on position Z: different constants a result in different symbols R_a. We will prove that V is a level, and that it contains only root variables.

To prove that V is a level, consider an edge (R, i), (S, j) in the attribute graph, and suppose that V contains (R, i). Thus, there exists a symbol R_a s.t. $(R_a, i) \in V'$. Let c_k be the component in d that connects these two attributes, i.e. c_k contains two atoms R and S, with the same variable on positions i and j respectively. Consider any substitution $\theta : Vars_Z(c) \to A$ that maps R to R_a: then in $c[\theta]$ we have an edge from (R_a, i) to (S_b, j) (or to (S, j) if S has no attributes in level Z), proving that $(S, j) \in Z$. (Obviously S_b is also in K, because it is connected to R_a.)

Finally, we show that every variable in V is a root variable. Let $(R, i) \in V$; by definition, there exists $(R_a, i) \in V'$. Consider some atom R in some component c_k, and let x be the variable on position i. Define $\theta \in \Theta_Z(c_k)$ to be s.t. it is injective, and it maps the atom R to R_a. By Lemma 7.5, the minimized query $d[A/Z]$ contains the component $c_k[\theta]$, and this further belongs to K (because it contains the symbol R_a): hence x is a root variable in $c_k[\theta]$, and it must also be a root variable in c_k (because $c_k[\theta]$ and c_k are isomorphic, since we chose θ to be injective). □

Example 7.7. *We illustrate Case 1. Consider:*

$$d = c_1 \vee c_2 = R(x), S(x, y, z) \vee S(x', y', z'), T(y')$$

Level $Z_1 = \{x, x'\}$ *splits* c_1 *into two subcomponents* $R(x)$ *and* $S(x, y, z)$, *while level* $Z_2 = \{y, y'\}$ *splits* c_2 *into two subcomponents* $S(x', y', z')$ *and* $T(y')$. *Therefore* $Z_3 = \{z, z'\}$ *is the only non-splitting level. We choose level* Z_3 *and rewrite*[5]

$$d[a/Z_3] = R(x), S(x, y, a) \vee S(x', y', a), T(y')$$

and this query is still without separator.

Note that it would have been a mistake to choose Z_1*:*

$$\begin{aligned} d[a/Z_1] =& R(a), S(a, y, z) \vee S(a, y', z'), T(y') \\ =& (R(a) \vee S(a, y', z'), T(y')) \wedge (S(a, y, z) \vee S(a, y', z'), T(y')) \\ =& (R(a) \vee S(a, y', z'), T(y')) \wedge S(a, y, z) \end{aligned}$$

and $d[a/Z_1]$ *is a safe query (because all three elements of its* V*-shaped CNF lattice have separators). Thus, Case 1 is necessary.*

From now on we will assume that Case 1 does not apply, i.e. every level Z splits some component.

[5] In this example it suffices to choose $A = \{a\}$; choosing a larger set $A = \{a, b\}$ leads to a longer but similar query.

7.4 Case 2: Two Related Levels

By assumption there are at least three distinct levels. Choose any three distinct levels, and call them Z_1, Z_2, Z_3. We will fix these three levels, for both cases 2 and 3.

For each $i = 1, 2, 3$, let C_i be the set of components c_j that are split by the level Z_i. For every $i = 1, 2, 3$, we have $C_i \neq \emptyset$: otherwise, Z_i is non-splitting, and we have assumed that every level splits some component.

We say that two distinct levels Z_i, Z_j are *related* if there exists $s \in sc_{Z_i}(c_i)$ and a homomorphism $s \to c_j$ for $j \neq i$. In this section we prove Theorem 7.2 in the case when there exists two related levels. Here we pick Z to be the other (third) level among Z_1, Z_2, Z_3, and prove our claim for $d[A/Z]$. If there are multiple pairs of related levels, we choose arbitrarily one such a pair, then let Z be the third level.

LEMMA 7.8 CASE 2. *Suppose Z_1, Z_2 are related levels: that is, there exists $s_1 \in sc_{Z_1}(c_1)$ and a homomorphism $h : s_1 \to c_2$. Let A be a set of constants such that $|A| \geq |Var_{Z_3}(c_1)| + |Var_{Z_3}(c_2)|$. Consider the CNF expression for $d[A/Z_3] = \bigwedge_l d'_l$. Then there exists l s.t. some symbol-component in d'_l has no separator.*

PROOF. The plan is this. Consider c_1, c_2 the components witnessing the fact that Z_1, Z_2 are related. We will chose certain substitutions θ_1, θ_2 s.t. $c_1[\theta_1] \vee c_2[\theta_2]$ are symbol-connected, and, together, have no separator; we also show that they occur in one of the disjunctive queries d'_l in the minimized query $d[A/Z_3]$.

Let $\theta_2 \in \Theta_{Z_3}(c_2, A)$ be any injective substitution, and denote $A_2 = Im(\theta_2)$. Define $\theta_1 \in \Theta_{Z_3}(c_1, A)$ as follows. Recall that $s_1 \in sc_{Z_1}(c_1)$ is the subcomponent for which we have a homomorphism $h : s_1 \to c_2$. For every $x \in Var_{Z_3}(s_1)$, define $\theta_1(x) = \theta_2(h(x))$. For every other Z_3-variable y of c_1, define $h(y)$ to be a fresh constant, which is not in A_2. In other words, we start by splitting c_1 in two parts, using the level Z_1: the subcomponent s_1, and everything else. All the Z_3 variables in the subcomponent s_1 are mapped by θ_1 to the values $\theta_2 \circ h$; all the Z_3 variables in the rest, are mapped injectively to fresh new constants, not used by θ_2. Denote the latter $A_1 = Im(\theta_1) - A_2$. Note that both $c_1[\theta_1]$ and $c_2[\theta_2]$ are connected conjunctive queries. This is because level Z_1 disconnects c_1, and the only way this can happen is if c_1 has a unique root variable, which is on level Z_1: therefore, level Z_3 cannot disconnect c_1. Similarly for c_2. Furthermore, the two queries $c_1[\theta_1]$ and $c_2[\theta_2]$ are symbol-connected, because of the homomorphism $h : s_1[\theta_1] \to c_2[\theta_2]$. Furthermore, any disjunctive query that contains $c_1[\theta_1] \vee c_2[\theta_2]$ cannot have a separator: this is because the only root variable in $c_1[\theta_1]$ is on level Z_1, and the only root variable in $c_2[\theta_2]$ is on level Z_2, while a separator must be a single level that contains root variables from all components (Prop. 6.2). Thus, to prove the lemma, it suffices to show that there exists a disjunctive query d'_l in the minimized CNF expression of $d[A/Z_3]$ that has both components $c_1[\theta_1] \vee c_2[\theta_2]$. We prove this by extending the proof of Lemma 7.5.

Like in Lemma 7.5, we first prove that there exists some d'_k in Eq. 13 that (1) contains both queries $c_1[\theta_1]$ and $c_2[\theta_2]$, and (2) for any other query $s'[\theta']$ in d'_k, there is no homomorphism $s'[\theta'] \to c_1[\theta_1]$ and there is no homomorphism $s'[\theta'] \to c_2[\theta_2]$. Once we prove this, then we use exactly the same argument as at the end of Lemma 7.5 to prove that there exists a disjunctive query d'_l in the minimized CNF

of $d[A/Z_3]$ that contains both $c_1[\theta_1] \vee c_2[\theta_2]$: we omit this step of the proof since it is identical to Lemma 7.5. Thus, it suffices to show how to construct d'_k with the two properties above.

Fix a pair j, θ'. We need to show that there exists a subcomponent $s' \in sc_{Z_3}(c_j)$ s.t. there is no homomorphism from $s'[\theta']$ to either $c_1[\theta_1]$ or to $c_2[\theta_2]$.

If Z_3 does not split c_j, then the only component $s' \in sc_{Z_3}(c_j)$ is c_j itself. In that case there cannot be any homomorphism $c_j[\theta'] \rightarrow c_1[\theta_1]$, because that would give us a homomorphism $c_j \rightarrow c_1$, contradicting the fact that the query d was minimized; similarly there is no homomorphism $c_j[\theta'] \rightarrow c_2[\theta_2]$.

So we assume wlog that Z_3 splits c_j: then c_j has a root variable z on level Z_3, and that is the only root variable in c_j. We know the following from the proof of Lemma 7.5: (a) there exists $s' \in sc_{Z_3}(c_j)$ s.t. there exists no homomorphism $s'[\theta'] \rightarrow c_1[\theta_1]$, and (b) there exists $s' \in sc_{Z_3}(c_j)$ s.t. there exists no homomorphism $s'[\theta'] \rightarrow c_2[\theta_2]$. This is not sufficient yet, because the two s' may not be the same. To prove that they can be chosen the same, we use the special structure of $c_1[\theta_1]$ and $c_2[\theta_2]$ and the fact that c_j has the root variable z on level Z_3.

We consider two cases. First, when $\theta'(z) \in A_2$. Then let s' be given by (b) above, i.e. there is no homomorphism $s'[\theta'] \rightarrow c_2[\theta_2]$. We prove that there is also no homomorphism $g : s'[\theta'] \rightarrow c_1[\theta_1]$. Suppose such a g exists: every atom of s' contains the variable z, hence every atom in $s'[\theta']$ contains the constant $\theta'(z) \in A_2$, the image of g is contained entirely in $s_1[\theta_1]$. But, by assumption there is a homomorphism $h : s_1 \rightarrow c_2$ (that "relates" the levels Z_1, Z_2) and this can be extended to a homomorphism $h : s_1[\theta_1] \rightarrow c_2[\theta_2]$ (because of the way we constructed θ_1): by composing $h \circ g$ we obtain a homomorphism $s'[\theta'] \rightarrow c_2[\theta_2]$, which is a contradiction. Thus, there is no homomorphism $s'[\theta'] \rightarrow c_1[\theta_1]$, and this proves the claim for the first case.

The second case is when $\theta'(z) \in A_1$. Then we choose s' given by (a) above, i.e. there is no homomorphism $s'[\theta'] \rightarrow c_1[\theta_1]$. Clearly there cannot be a homomorphism $s'[\theta'] \rightarrow c_2[\theta_2]$, since every atom in $s'[\theta']$ has a constant from A_1, while $c_2[\theta_2]$ has no such constants. This proves the claim for the second case. □

Example 7.9. *We illustrate Case 2 with:*

$$d = c_1 \vee c_2 \vee c_3 = R(x_1, y_1), S(x_1, z_1) \vee R(x_2, y_2), T(y_2, z_2) \vee S(x_3, z_3), T(y_3, z_3)$$

Level $Z_1 = \{x_1, x_2, x_3\}$ splits c_1 into $R(x_1, y_1)$ and $S(x_1, z_1)$; similarly level $Z_2 = \{y_1, y_2, y_3\}$ splits c_2, and level $Z_3 = \{z_1, z_2, z_3\}$ splits c_3. Thus, every level is splitting, and Case 1 does not apply. Case 2 applies here, because every two levels are related. For example there is a homomorphism from the subcomponent $R(x_1, y_1)$ to c_2, showing that levels Z_1, Z_2 are related. We therefore choose level Z_3, and two constants $A = \{a, b\}$ and rewrite to:

$$\begin{aligned} d[A/Z_3] = & c_1[A/Z_3] \vee c_2[A/Z_3] \vee S(x_3, a), T(y_3, a) \vee S(x_3, b), T(y_3, b) \\ = & (c_1[A/Z_3] \vee c_2[A/Z_3] \vee S(x_3, a) \vee S(x_3, b)) \wedge \\ & (c_1[A/Z_3] \vee c_2[A/Z_3] \vee S(x_3, a) \vee T(y_3, b)) \wedge \\ & (c_1[A/Z_3] \vee c_2[A/Z_3] \vee T(y_3, a) \vee S(x_3, b)) \wedge \\ & (c_1[A/Z_3] \vee c_2[A/Z_3] \vee T(y_3, a) \vee T(y_3, b)) \end{aligned}$$

The CNF expression is given in the last four rows, and is the conjunction of four disjunctive queries. Consider the second:

$$d'_2 = c_1[A/Z_3] \vee c_2[A/Z_3] \vee S(x_3, a) \vee T(y_3, b)$$

Expanding the first two expressions results in 4 components. Two of these are redundant, because one contains $S(x_1, a)$ and the other contains $T(y_2, b)$. However, the following two components are not redundant:

$$d'_2 = \ldots \vee R(x_1, y_1), S(x_1, b) \vee R(x_2, y_2), T(y_2, a) \vee \ldots$$

Obviously, d'_2 has no separator.

In the next example we show why it is important to pick the level according to our rule (as the third level, if two levels are related). Consider:

$$d = c_1 \vee c_2 \vee c_3 =$$
$$U(x, y', z'), V(x, y'', z'') \vee R(y), S(x'', y, z''), U(x'', y, z'') \vee V(x', y', z), S(x', y', z), T(z)$$

Here $Z_1 = \{x, x', x''\}$ splits c_1 into two subcomponents: $U(x, y', z')$ and $V(x, y'', z'')$, and there is a homomorphism from the first to c_2: here we must pick level $Z_3 = \{z, z', z''\}$. There is also a homomorphism from the second component to c_3; here we should pick level $Z_2 = \{y, y', y''\}$. Either choice is fine, but it would be a mistake to pick $Z_1 = \{x, x', x''\}$, since Z_2, Z_3 are unrelated. To see that, consider expanding it with a set of constants $A = \{a, b, c, \ldots\}$:

$$d[A/Z_1] = \bigvee_{v \in A} U(v, y', z')V(v, y'', z'') \vee c_2[A/Z_1] \vee c_3[A/Z_1] = \bigwedge_k d'_k$$

Each d'_k contains $c_2[A/Z_1] \vee c_3[A/Z_1]$ and, for each constant $v \in A$, it contains either $U(v, y', z')$ or $V(v, y'', z'')$ (for a total of $2^{|A|}$ disjunctive queries d_k). However, each d'_k has a separator. Indeed:

$$c_2[A/Z_1] \vee c_3[A/Z_1] = \bigvee_{a \in A} (R(y), S(a, y, z''), U(a, y, z'') \vee V(a, y', z), S(a, y', z), T(z))$$

For any fixed constant a the disjunction of the two components above has no separator: in particular the entire expression above has no separator. However, for every constant a we must include in d'_k either $U(a, y', z')$ or $V(a, y'', z'')$, and either the first or the second component above becomes redundant. If A had a single constant a, then it is obvious that d'_k has a separator; one can check that d'_k continues to have a separator for an arbitrary A (just choose separately, a separator for each constant $a \in A$). This shows that we cannot expand on Z_1. On the other hand, expanding on level Z_3 with a single constant a we obtain:

$$\begin{aligned} d[a/Z_3] &= U(x, y', a), V(x, y'', a) \vee R(y), S(x'', y, a), U(x'', y, a) \vee V(x', y', a), S(x', y', a), T(a) \\ &= (U(x, y', a), V(x, y'', a) \vee R(y), S(x'', y, a), U(x'', y, a) \vee V(x', y', a), S(x', y', a)) \wedge \\ &\quad (U(x, y', a), V(x, y'', a) \vee R(y), S(x'', y, a), U(x'', y, a) \vee T(a)) \end{aligned}$$

Neither conjunct has a separator.

7.5 Case 3: Three Unrelated Levels

We continue to assume that every level is splitting. Further, we use the same notations Z_1, Z_2, Z_3, for three arbitrary, but fixed levels, and C_1, C_2, C_3 for the

non-empty sets of components split by each of these three levels. We assume that no two levels are related. Since the query d is symbol-connected, for each $i \neq j$ there exists a path in the co-occurrence graph from a query in C_i to a query in C_j. Let n_{ij} be the length of the shortest such path. Assume n_{12} is the minimum of n_{12}, n_{13}, n_{23} (break ties arbitrarily). Then we pick level Z_3.

LEMMA 7.10 CASE 3. *Suppose $n_{12} = \min(n_{12}, n_{13}, n_{23})$. Let A be a set of constants such that $|A| \geq \max_i(|Var_{Z_3}(c_i)|)$, for all components c_i of d, and let $d[A/Z_3] = \bigwedge_l d'_l$ be the minimized, CNF expression of $d[A/Z_3]$. Then there exists l such that d'_l has a symbol-component without a separator.*

PROOF. Suppose that the shortest path from C_1 to C_2 is:

$$c_1 = c^0, c^2, \ldots, c^n = c_2$$

where $c_1 \in C_1$ and $c_2 \in C_2$; thus, $n_{12} = n$. That is, any pair of consecutive queries c^{i-1}, c^i share a common relational symbol. Note that, for every $i = 1, n-1$, c^i does not belong to any of C_1, C_2, C_3: otherwise we would have a strictly shorter path between two of the three sets.

Define, inductively, the substitutions $\theta_i \in \Theta_{Z_3}(c^i, A)$ as follows. Start by defining $\theta_0 : Var_{Z_3}(c^0) \to A$ to be any injective substitution. To define $\theta_i : Var_{Z_3}(c^i) \to A$, consider the two atoms in c^{i-1} and c^i that share a common relational symbol R. If the symbol R has no attribute in level Z_3, then choose θ_i an arbitrary injective function. Otherwise, let x be the unique Z_3-variable in the first atom, and y be the unique Z_3 variable in the second atom (obviously, they occur on the same attribute position in R). Choose θ_i to be injective and s.t. $\theta_i(y) = \theta_{i-1}(x)$. Note that $c^i[\theta_i]$ is a connected conjunctive query, for all $i = 0, \ldots, n$, since none of the queries $c^0, \ldots, c^n$ is in C_3. We note two simple properties: the set of queries $c^0[\theta_0], \ldots, c^n[\theta_n]$ is symbol-connected, because of the way we constructed the θ_i's, and there is no homomorphism from $c^i[\theta_i]$ to $c^j[\theta_j]$ for $i \neq j$: otherwise we would have a homomorphism $c^i \to c^j$, contradicting the assumption that d is minimized.

We will show now that there exists a disjunctive sentence d'_l in the minimized CNF expansion of $d[A/Z]$ that contains all components $c^i[\theta_i]$: this proves our claim, because the only root variable in c^0 is on level Z_1, and the only root variable in c^n is on level Z_2, hence d'_l cannot have a separator. To prove this claim, we extend the proof of Lemma 7.5. We show that there exists a d'_k (given by Eq. 13) with the following properties: (1) it contains all queries $c^0[\theta_0], \ldots, c^n[\theta_n]$, and (2) for any other component $s'[\theta']$ in d'_k, there is no homomorphism to any of the queries $c^i[\theta_i]$. If such a d'_k exists, then we use exactly the same argument as at the end of the proof of Lemma 7.5 to show that there exists a d'_l in the minimized CNF expression of $d[A/Z]$ that contains all queries $c^0[\theta_0], \ldots, c^n[\theta_n]$.

Fix j and $\theta' \in \Theta_{Z_3}(c_j, A)$. We need to show that there exists $s' \in sc_{Z_3}(c_j)$ s.t. there is no homomorphism $s'[\theta'] \to c^i[\theta_i]$ for any $i = 0, n$. First, note that if $c_j \notin C_3$, then Z_3 does not split c_j, and the only subcomponent in $sc_{Z_3}(c_j)$ is c_j itself: in that case there is no homomorphism $c_j[\theta'] \to c^i[\theta_i]$, because that would imply a homomorphism $c_j \to c^i$, which contradicts the fact that d was minimized. So if $c_j \notin C_3$ then there is no homomorphism, and then we can choose $s \in sc_{Z_3}(c_j)$ arbitrarily.

Suppose now that $c_j \in C_3$, and suppose there exists $s' \in sc_{Z_3}(c_j)$ s.t. there is a

homomorphism $s'[\theta'] \to c^i[\theta_i]$. Then $i \neq 0$: otherwise levels Z_3 and Z_1 would be related. Similarly, $i \neq n$; otherwise levels Z_3 and Z_2 would be related. (Recall that we assumed that no two levels among Z_1, Z_2, Z_3 are related.) Hence $i = 1, \dots, n-1$. Then, we obtain the following two paths in the co-occurrence graph of d, one from C_1 to C_3 and one from C_3 to C_2:

$$c^0, c^1, \dots, c^i, c_j$$
$$c_j, c^i, c^{i+1}, \dots, c^{n-1}, c^n$$

None of them can be shorter than n, because we assumed n to be the shortest possible path. Since $i \neq 0, i \neq n$, it follows that $n = 2$ and $i = 1$.

Now we can prove our claim, by considering two simple cases. If $n > 2$, then for every pair j, θ', choose an arbitrary subcomponent $s' \in sc_{Z_3}(c_j)$: we know there is no homomorphism to any $c^i[\theta_i]$. If $n = 1$, then we know from the proof of Lemma 7.5 that there exists $s' \in sc_{Z_3}(c_j)$ s.t. there exists no homomorphism to $c^1[\theta_1]$. This s' satisfies our claim, since we already know that there are no homomorphisms $s'[\theta'] \to c^0[\theta_0]$ or $s'[\theta'] \to c^2[\theta_2]$. $\square$

8. FORBIDDEN QUERIES ARE HARD

Recall that a forbidden query is 2-leveled disjunctive query that is symbol-connected and has no separator. They form an important subclass of UCQ, since they are the "simplest" possible queries that are still hard.

8.1 Definitions and Main Result

We denote X, Y the two levels; variables in X are denoted $x, x_1, x_2, \dots$, those in Y are denoted $y, y_1, y_2, \dots$ There are three kinds or relation symbols: (a) Unary, $R(x)$, (b) binary $S(x, y)$, and (c) unary $T(y)$.

Note that if a leveled query contains only relation symbols of arity ≤ 2, then it is not necessarily a forbidden query. This is illustrated by Example 7.9: all symbols have arity 2, yet the query is not forbidden, because it has three levels. We have shown in the example how to rewrite it to a forbidden query.

We prove in this section the following, which is the most difficult result in the paper.

THEOREM 8.1. *If d is a forbidden query, then the problem: given a probabilistic database D compute $P(d)$, is hard for $FP^{\#P}$.*

EXAMPLE 8.2. *Query $h_0 = R(x), S(x, y), T(y)$ is forbidden. All queries h_k, $k \geq 1$ in Example 4.5 are forbidden queries. Below are further examples of forbidden queries:*

$$\begin{aligned} g =& U(x, y_1), S_1(x, y_1), U(x, y_2), S_2(x, y_2), U(x, y_3), S_3(x, y_3) \vee \\ & S_1(x, y), S_2(x, y), S_3(x, y), S'(x, y) \vee \\ & S'(x, y), T(y) \\ j =& R(x), S_1(x, y_1), S_2(x, y_2) \vee S_1(x_1, y), S_2(x_2, y), T(y) \end{aligned}$$

Queries h_k, $k \geq 0$ form an important subset of forbidden queries, but they are not the only ones, as the example showed.

Our proof of Theorem 8.1 consist of reducing the #PP2DNF problem to the problem of computing $P(D)$.

Definition 8.3. *The* Positive Partitioned 2 DNF *problem, #PP2DNF, is the following. Given a Boolean formula* $\Phi = \bigvee_{(i,j)\in E} X_i Y_j$*, compute the number of satisfying assignments, denoted* $\#\Phi$*.*

Provan and Ball [Provan and Ball 1983] have shown[6] that the #PP2DNF problem is hard for #P. Note that the problem is completely defined by the set $E \subseteq [n_1] \times [n_2]$, where n_1 is the number of X_i variables, and n_2 the number of Y_j variables.

As a warmup, we prove Theorem 8.1 by reviewing a simple case, shown in [Dalvi and Suciu 2004; Dalvi and Suciu 2007b]. Recall that a root variable in c_i is a variable that occurs in all atoms of c_i.

Proposition 8.4. *Let* $d = \bigvee_i c_i$ *be a forbidden query. Suppose that there exists a component* c_i *that has no root variables. Then computing* $P(D)$ *is hard for* $FP^{\#P}$*.*

Proof. We illustrate the proof only in the special case when $c_i = R(x), S(x,y), T(y)$: here neither x nor y is a root variable. The general case follows immediately, using the techniques in [Dalvi and Suciu 2004; Dalvi and Suciu 2007b]. Given a PP2DNF instance Φ, defined by $E \subseteq [n_1] \times [n_2]$, construct the following database instance D: R has n tuples, $R(a_1), \ldots, R(a_{n_1})$ and their probabilities are $1/2$; S contains all tuples $S(a_i, b_j)$ for $(i,j) \in E$, and their probabilities are 1; and T consists of all tuples $T(b_1), \ldots, T(b_{n_2})$, and their probabilities are $1/2$. All relation symbols other than R, S, T are empty in D; thus $D \not\models c_j$ for $j \neq i$ (since d is minimized). Then $P(d) = \#\Phi/2^{n_1+n_2}$, proving the proposition. □

In the rest of this section we will assume that every component c_i in d has a root variable. Thus, there are three kinds of components in d:

Left component. One root variable, x; other variables $y_1, y_2, \ldots$

Central component. Two root variables, x, y and no other variables.

Right component. One root variable, y; other variables $x_1, x_2, \ldots$

We write the forbidden query as:

$$d = d^L \vee d^C \vee d^R$$

where d^L, d^C, d^R consists of all left, central, and right components respectively. In Example 8.2, $g^C = S_1(x,y), S_2(x,y), S_3(x,y), S'(x,y)$, and $j^C = \texttt{false}$.

In the rest of this section we will give a reduction from the #PP2DNF problem to the problem of computing $P(d)$, for any forbidden query d. In Sect. 8.2 and Sect. 8.3 we prove some preliminary results about forbidden queries. Section 8.4 refines the #PP2DNF problem to a new problem that we need in the reduction. The core of the reduction is in Sect. 8.5 and Sect. 8.6: this is where we establish the connection between logic (the structure of the query d) and algebra (certain multivariate polynomials associated to d). Finally, Sect. 8.7 proves that the reduction is indeed in PTIME.

[6] Their result is for 2CNF; it extends immediately to 2DNF.

8.2 The Zig-Zag-Zig Block

We start by describing a construction that we need twice during the proof. Given a forbidden query d and two constants a, b, we define a database instance $D^z(a, b)$, called the *zig-zag-zig* block. Let n_x be the maximum number of x_i variables occurring in any component of d, and n_y the maximum number of y_i variables.

The block $D^z(a, b)$ has constants $a, b, c_1, \ldots, c_{n_x}$ and $d_1, \ldots d_{n_y}$. $D^z(a, b)$ contains the following tuples:

—$R(a), R(c_i)$, $i = 1, n_x$, for every left unary symbol R.
—$S(a, d_i)$, $S(c_j, d_i)$, $S(c_j, b)$, $i = 1, n_x, j = 1, n_y$, for every binary symbol S.
—$T(d_j)$, $j = 1, n_y$. $T(b)$, for every right unary symbol T.

The binary symbols S form several zig-zag-zig patterns, from a to c_i to d_j and to b, hence the name. The size of the zig-zag-zig block depends only on the size of the query.

We give here the first application of the zig-zag-zig block: a second application will come at a later point in the proof of Theorem 8.1.

Given a forbidden query d, we classify the relational symbols as follows:

A Left Symbol. is a symbol that occurs in d^L.
A Right Symbol. is a symbol that occurs in d^R.
A Central Symbol. is a symbol that is neither left nor right.

A central symbol occurrs exclusively in d^C. Note that a symbol may be both left and right; for example, S in $h_1 = R(x), S(x, y) \vee S(x, y), T(y)$ (Example 4.5) is both left and right.

DEFINITION 8.5. *d is called* long *if no component c_i contains both a left and a right symbol; in particular, no symbol is both a left and a right symbol. Otherwise, d is called* short.

Queries h_1, h_2 in Example 4.5 are short; queries h_k, $k \geq 3$ are long. Our first transformation is to make any forbidden query long.

PROPOSITION 8.6. *If d is a forbidden query, then there exists a long query d^z, over a different vocabulary, such that the computational problem for $P(d^z)$ can be reduced to the computational problem for $P(d)$. In particular, if d^z is hard for $FP^{\#P}$ then so is d.*

PROOF. We first describe how to translate any database D for d^z into a database D^z for d s.t. their tuples are in 1-1 correspondence. Note that d^z will be 2-leveled; call the two levels X and Y. We define D^z to be the union of all zig-zag-zig blocks $D^z(a, b)$, for every constant a in occurring in an X-attribute, and every constant b occurring in a Y-attribute in D. Now we define the vocabulary for d^z, s.t. the tuples of D and D^z are in 1-1 correspondence. Referring to the tuples of the block $D^z(a, b)$ described above, we want to have the following tuples in D:

Left-Unary:. $R(a)$.
Binary-for-Left-Unary:. $R^i(a, b)$, for $i = 1, n_x$.
Binary:. $S^{0j}(a, b)$, $S^{ij}(a, b)$, $S^{i0}(a, b)$, for $i = 1, n_x, j = 1, n_y$.

Binary-for-Right-Unary:. $T^j(a, b)$, for $j = 1, n_y$.

Right-binary:. $T(b)$.

That is, during the $D \mapsto D^z$ translation, each tuple in this list is mapped precisely to the corresponding tuple in the list describing the block $D^z(a, b)$. For example, tuple $R^3(a, b)$ in D is mapped to $R(c_3)$ in D^z, while tuple $S^{52}(a, b)$ in D is mapped to $S(c_5, d_2)$ in D^z. This defines the new vocabulary, and the translation $D \mapsto D^z$. Since the tuples of D and D^z are in 1-1 correspondence, we simply copy their probabilities too, which allows us to map a probabilistic database D to a probabilistic database D^z.

It remains to do the following. (1) Construct the query d^z s.t. d^z over D is equivalent to d over D^z. More precisely, for any possible world $W \subseteq D$, if $W^z \subseteq D^z$ is the corresponding possible world in D^z, then $W^z \models d$ iff $W \models d^z$. This ensures $P_{D^z}(d) = P_D(d^z)$. (2) Prove that d^z is a forbidden query. (3) Prove that d^z is long. Each of these proofs is rather tedious, but straightforward, and is omitted. Details are in [Dalvi et al. 2010b]. Here, we prefer to illustrate the main ideas with examples. □

EXAMPLE 8.7. *We illustrate Prop. 8.6 with three examples. First, on* $h_1 = R(x), S(x, y) \vee S(x, y), T(y)$. *The long query obtained through the zig-zag construction is:*

$$\begin{aligned} h_1^z = &R(x), S^{01}(x, y) \vee S^{01}(x, y), T^1(x, y) \vee \\ &R^1(x, y), S^{11}(x, y) \vee S^{11}(x, y), T^1(x, y) \vee \\ &R^1(x, y), S^{10}(x, y) \vee S^{10}(x, y), T(y) \end{aligned}$$

The reader may check that the problem $P(d^z)$ *can be reduced to the problem* $P(d)$.

The second example shows how to handle multiple x_i, *or multiple* y_j *variables:*

$$d = S_1(x, y_1), S_2(x, y_2) \vee S_1(x_1, y), S_2(x_2, y)$$

Here $n_x = n_y = 2$ *and therefore the equivalent long query is:*

$$\begin{aligned} d^z = &\bigvee_{j,j' \in [2]} S_1^{0j}(x, y_1) S_2^{0j'}(x, y_2) \vee \bigvee_{i,j \in [2]} S_1^{0j}(x, y), S_2^{ij}(x, y) \vee \bigvee_{i,j \in [2]} S_1^{ij}(x, y), S_2^{0j}(x, y) \\ &\vee \bigvee_{i,j \in [2]} S_1^{ij}(x, y), S_2^{ij}(x, y) \\ &\vee \bigvee_{i,j \in [2]} S_1^{ij}(x, y), S_2^{i0}(x, y) \vee \bigvee_{i,j \in [2]} S_1^{i0}(x, y), S_2^{ij}(x, y) \vee \bigvee_{i,i' \in [2]} S_1^{i0}(x_1, y), S_2^{i'0}(x_2, y) \end{aligned}$$

The left symbols are $S_1^{01}, S_1^{02}, S_2^{01}, S_2^{02}$ *and the right symbols are* $S_1^{10}, S_1^{20}, S_2^{10}, S_2^{20}$. *One can check that the query is forbidden by examining the following subset:* $S_1^{01}(x, y_1), S_2^{01}(x, y_2) \vee S_1^{01}(x, y), S_2^{11}(x, y) \vee S_1^{11}(x, y), S_2^{11}(x, y) \vee S_1^{11}(x, y), S_2^{10}(x, y) \vee S_1^{10}(x, y_1), S_2^{10}(x, y_2)$: *this subset does not have a separator, hence neither does* d^z. *In this example, we did not have to choose two intermediate points, i.e. the query* d^z *remains forbidden if we replace* $[2]$ *with* $[1]$ *above.*

Finally, we explain why the zig-zag-zig block contains several c_i *and several* d_J *constants: if not, then the translated query may minimize, and be no longer forbid-*

den. This is illustrated by:

$$\begin{aligned}d =& S_1(x, y_1), S_2(x, y_1), S_1(x, y_2), S_3(x, y_2), S_2(x, y_3), S_3(x, y_3)\\ &\vee S_1(x, y), S_2(x, y), S_3(x, y)\\ &\vee S_1(x_1, y), S_2(x_1, y), S_1(x_2, y), S_3(x_2, y), S_2(x_3, y), S_3(x_3, y)\end{aligned}$$

Here, if we choose only one copy of intermediate points, then in any new component, two of the three y_i's in the left component are equal, and that component becomes redundant because of the central component $S_1(x, y), S_2(x, y), S_3(x, y)$. Instead, by using $n_x = n_y = 3$, we allow the three y_i's to map to different intermediate constants; the translated query d^z is a forbidden query.

From now on, in the rest of this section we will assume that the forbidden query is long.

8.3 Final Forbidden Queries

In our proof of Theorem 8.1 we must first further simplify the forbidden query, and we do this using three rewrite rules $d \to d'$ described below. We call a forbidden query *final* if it can no longer be rewritten to another forbidden query using these rules. It suffices to prove Theorem 8.1 only for final queries, because whenever $d \to^+ d'$ and d' is hard for $FP^{\#P}$, then d is also hard for $FP^{\#P}$. In this section we define the three rules for $\to$ formally, and prove some structural properties of final queries, which we use later in the proof of Theorem 8.1.

The first two rules are:

$$\begin{aligned}d &\to d[S = \texttt{false}]\\ d &\to d[S = \texttt{true}]\end{aligned}$$

That is, choose a symbol S (unary or binary) and replace it with `false` or with `true`. We have already defined the first in Def. 4.9; the second rule is similar. Notice that after substituting with `true` we may need to minimize the resulting disjunctive query.

In the third rewriting, we fix a left binary symbol S, choose a set of k constants, $B = \{b_1, \ldots, b_k\}$, no larger than the number of distinct y_i variables in any left component, and rewrite the query assuming that it is evaluated on database instances with the following two properties. (a) Every tuple in the relation S has the form $S(x, b_i)$ for some $b_i \in B$, i.e. it has a constant in B on its second attribute. (b) The constants b_i do not occur in any tuple of a central or a right symbol. We rewrite d to a new query, denoted, $d[S =^L B]$, which is equivalent to d over these restricted database instances:

$$d \to d[S =^L B]$$

defined as follows. If $d = \bigvee_i c_i$, then $d[S =^L B] = \bigvee_i c_i[S =^L B]$, where $c_i[S =^L B]$ is defined as follows. If c_i contains at least one central or right symbol, then:

$$\begin{aligned}c_i[S =^L B] =& \texttt{false} && \text{if } c_i \text{ contains } S\\ c_i[S =^L B] =& c_i && \text{otherwise}\end{aligned}$$

If c_i contains only left symbols, then x is a root variable in c_i. Let $sc_X(c_i) = \{s_1, \ldots, s_m\}$ be its X-subcomponents (see Sect. 7), thus:

$$c_i = \exists x.(\exists y_1.s_1(x, y_1) \wedge \exists y_2.s_2(x, y_2), \ldots \wedge \exists y_m.s_m(x, y_m))$$

Assuming w.l.o.g. that S occurs in the first n subcomponents, where $0 \leq n \leq m$, $c_i[S =^L B]$ is defined as:

$$c_i[S =^L B] = \exists x.(\bigvee_i s_1[b_i/y_1]) \wedge \ldots \wedge (\bigvee_i s_n[b_i/y_n]) \wedge \\ (\bigvee_i s_{n+1}[b_i/y_{n+1}] \vee \exists y_{n+1}.s_{n+1}) \wedge \ldots \wedge (\bigvee_i s_m[b_i/y_m] \vee \exists y_m.s_m) \quad (14)$$

This completes the definition of $d[S =^L B]$. Note that in order to obtain its DNF representation, we need to apply the distributivity law to Eq. 14.

As in the previous rewritings in this paper, we do not view $d[S =^L B]$ as a query with constants, but rather as a query over a new vocabulary, where the left binary symbol $S(x, y)$ is removed, and replaced with k unary symbols $R_1(x), \ldots, R_k(x)$, and every other left binary symbol $S'(x, y)$ is kept and, in addition, k new binary symbols are introduced on its behalf.

Example 8.8. *Consider the query:*

$$d = S(x, y_1), S_1(x, y_1), S_1(x, y_2), S_2(x, y_2) \vee S_1(x, y), S_3(x, y) \vee \ldots$$

where S, S_1, S_2 are left binary symbols and S_3 is a central symbol. Let $B = \{b\}$. The rewriting $d[S =^L b]$ is such that (a) all tuples in the relation S are of the form $S(x, b)$, and (b) no tuples of the form $S_3(x, b)$ exists in the database. On the other hand, there may be tuples $S_1(x, b), S_2(x, b)$ as well as $S_1(x, y), S_2(x, y)$ with $y \neq b$. Thus:

$$d[S =^L a] = S(x, b), S_1(x, b), S_2(x, b) \vee S(x, b), S_1(x, b), S_1(x, y_2), S_2(x, y_2) \vee S_1(x, y), S_3(x, y) \vee \ldots \\ = R(x), R_1(x), R_2(x) \vee R(x), R_1(x), S_1(x, y), S_2(x, y) \vee S_1(x, y), S_3(x, y) \vee$$

We define similarly a right rewriting, $d \rightarrow d[S =^R A]$, where S is a right symbol and A a set of constants.

Define $\xrightarrow{*}$ the reflexive and transitive closure of $\rightarrow$. The following property similar to Lemma 4.12 holds

Lemma 8.9. *If $d \xrightarrow{*} d'$ then the computation problem for $P(d')$ can be reduced in polynomial time to the computation problem for $P(d)$. In particular, if d' is hard for $FP^{\#P}$, then so is d.*

The proof is a straightforward adaptation of the proof of Lemma 4.12 and is omitted.

Every sequence of rewritings $\xrightarrow{*}$ terminates, because it either reduces the number of total symbols by 1 (the first two rewritings), or reduces the number of binary symbols by 1 (the third rewriting).

Definition 8.10. *A forbidden query d is called* final *if there exists no rewriting $d \xrightarrow{+} d'$ s.t. d' is a forbidden query.*

Therefore, it suffices to prove hardness for all final queries: using the lemma above, this implies the hardness of all forbidden queries.

In the remainder of this section we prove three important properties of final queries.

8.3.1 *Strict Paths.* Recall that the co-occurrence graph $G(d)$ of d has a node for each components c_i, and an edge (c_i, c_j) for each pair of components that share a common relational symbol. By definition of a forbidden query, $G(d)$ is connected. A *left-right path* is a path $c_0, c_1, \ldots, c_k$ s.t. c_0 is a left component and c_k is a right component. We always have $k \geq 3$ because the query d is long. Since c_0 consists only of left symbols, c_1 must always contain at least one left symbol, because it connects to c_0. Similarly, both c_{k-1} and c_k contain right symbols. Call a left-right path *strict* if (a) forall $i \geq 2$, c_i does not contain any left symbol, and (b) forall $i \leq k-2$, c_i does not contain any right symbol. Thus, a strict path uses left and right symbols as sparingly as possible: it uses left symbols only in c_0, c_1 and right symbols only in c_{k-1}, c_k. Furthermore, in a strict path all components $c_1, c_2, \ldots, c_{k-1}$ are central components. It is easy to check that every left-right path of minimal length is a strict path; thus, a forbidden query has at least one strict path.

PROPOSITION 8.11. *Let d be a long, final query (Def. 8.10). Then every strict path $c_0, c_1, \ldots, c_k$ contains all relational symbols in d.*

In particular, all the left symbols occur in c_0 or in c_1 or in both, and, similarly, all the right symbols occur in c_{k-1} or c_k or both.

PROOF. Let S be a symbol that does not occur on the strict path. Then we can rewrite it to a forbidden query by setting $S = \texttt{false}$. Indeed, in $d[S = \texttt{false}]$ the path $c_0, c_1, \ldots, c_k$ remains unchanged. The query $d[S = \texttt{false}]$ is not a forbidden query in general, because it may be symbol-disconnected. However, the symbol-component that contains the path $c_0, c_1, \ldots, c_k$ is a forbidden query, and we can rewrite $d[S = \texttt{false}]$ to this symbol-component by setting all other symbols to `false`. □

8.3.2 *Type 1 and Type 2.* Let d be a long, forbidden query, and let $d = d^L \vee d^C \vee d^R$ be its left, central, and right parts.

PROPOSITION 8.12. *If d is final, then one of the following two cases holds for d^L:*

—*There is exactly one left unary symbol, R, and every left component c_i in d^L has exactly two variables x, y. We say that d^L is of Type 1.*

—*All left symbols are binary. We say that d^L is of Type 2.*

The similar statement holds for d^R: it can be of type 1 or type 2.

As an example, query g in Example 8.2 is a final query: g^L is of type 2 and g^R is of type 1. Query j in the example is not final.

PROOF. **Step 1** We start by showing that d contains at most one left unary symbol R. Suppose the contrary, that d has two or more left unary symbols. Consider any strict path $c_0, c_1, \ldots, c_k$. All left unary symbols must occur in c_0,

therefore c_0 contains at least two left unary symbols; let R be one of these left unary symbols. We claim that in that case $d[R = \texttt{true}]$ rewrites to a forbidden query, contradicting the fact that d is final. For that, we will prove that $d[R = \texttt{true}]$ contains the path $c_0[R = \texttt{true}], c_1, \ldots, c_k$ (only c_0 contains R), i.e. none of these k queries is redundant in $d[R = \texttt{true}]$; $c_0[R = \texttt{true}]$ is a left component because it has at least one other unary symbol, therefore this path is a left-right path in $d[R = \texttt{true}]$, implying that (the symbol-component containing this path) is a forbidden query, contradicting the fact that d is final. Thus, we only need to show that none of the components of the new path is redundant. We start with $c_0[R = \texttt{true}]$. If there exists a homomorphism $c[R = \texttt{true}] \rightarrow c_0[R\texttt{true}]$ then there exists also a homomorphism $c \rightarrow c_0$, contradicting the fact that d is minimized. Thus, $c_0[R = \texttt{true}]$ is not redundant. Next, consider c_i, $i > 0$. If there exists a homomorphism $c[R = \texttt{true}] \rightarrow c_i$ then c must contain R, hence it is a left component, hence $c, c_i, c_{i+1}, \ldots, c_k$ is a strict path in d, hence c contains all unary symbols, and therefore $c[R = \texttt{true}]$ has at least one other unary symbol, and therefore there cannot be a homomorphism $c[R = \texttt{true}] \rightarrow c_i$. This concludes the proof of Step 1.

Step 2 We prove that, if a left component c has exactly one left unary symbol R, then it has a single y-variable. We consider two cases:

—There exists a strict path starting at c: $c_0 = c, c_1, \ldots, c_k$. Suppose c_0 has two or more y-variables. We consider two sub-cases. First, when c_1 contains all left binary symbols. Let S be any left binary symbol. We claim that $d[S = \texttt{true}]$ rewrites to a forbidden query, contradicting the fact that d is final. Indeed, all queries in the rewritten path $c_0[S = \texttt{true}], c_1[S = \texttt{true}], c_2, \ldots, c_k$ are non-redundant in $d[S = \texttt{true}]$ (the proof is identical to that in step 1, and omitted), and this is indeed a path, because there exists at least one binary symbol other than S in c_0 (otherwise c_0 cannot have two y-variables), and that symbol connects $c_0[S = \texttt{true}]$ and $c_1[S = \texttt{true}]$. Second, when c_1 does not contain all left binary symbols, then we claim that $d[R = \texttt{true}]$ rewrites to a forbidden query. First we note that $c_0[R = \texttt{true}]$ is a left component, because it has two or more y-variables. Next we show that the path $c_0[R = \texttt{true}], c_1, \ldots, c_k$ is non-redundant. We use the same argument as above to show that $c_0[R = \texttt{true}]$ is non-redundant. We show that c_1 is non-redundant: suppose there were a homomorphism $c[R = \texttt{true}] \rightarrow c_1$, this implies that c must contain R, hence it is a left component, and $c, c_1, \ldots, c_k$ is a strict path. Let S be the left binary symbol missing from c_1. Since the path starting at c is strict, S must occur in either c or c_1, implying that it occurs in c. But this contradicts the existence of the homomorphism $c[R = \texttt{true}] \rightarrow c_1$. It is easy to check that c_i, $i \geq 2$ is non-redundant, because they don't contain any left symbols.

—There is no strict path starting at c. Consider any strict path, and let c_0 be its first component. By assumption c_0 contains the left unary symbol R (since c_0, c_1 contain all left symbols and c_1 is a central component), and we have proved in the first case that c_0 has only one y-variable. All symbols in c are left symbols, and therefore they must be occur in c_0 (none can occur in c_1, otherwise we have a strict path starting at c: $c, c_1, c_2, \ldots, c_k$). But this means that there exist a homomorphism $c \rightarrow c_0$ (here we use the fact that c_0 has a single y-variable),

contradicting the fact that d was minimized.

This proves step 2, and completes the proof. □

8.3.3 *More Properties for Type 1 and Type 2.* Finally, we prove two additional properties.

PROPOSITION 8.13. *Let d be a final query and $c_0, c_1, \ldots, c_k$ be a strict path.*

—*If d^L is of Type 1, then c_1 contains every left binary symbol.*
—*If d^L is of Type 2, if c_1 does not contain a left binary symbol U, then U occurs in every X-subcomponent of c_0. We call U* ubiquitous.

The similar statement holds for d^R.

Notice that, if d^L is of Type 2, then there must exists at least one ubiquitous symbol U. Otherwise, c_1 contains all left binary symbols, and we can construct a homomorphism $c_0 \rightarrow c_1$ (since c_0 has only left binary symbols, and c_1 has only two variables, x and y), contradicting the fact that d is minimized.

PROOF. Suppose d^L is of type 1, and let S be a left binary symbol that does not occur in c_1. Then $d[S = \texttt{true}]$ rewrites to a forbidden query. Indeed, all components $c_0[S = \texttt{true}], c_1, \ldots, c_k$ are non-redundant in $d[S = \texttt{true}]$ (by the same argument as in the previous proof). Let S' be the common symbol between c_0 and c_1. Then $c_0[S = \texttt{true}]$ is a left component, because it contains $R(x), S'(x, y), \ldots$ and S' connects $c_0[S = \texttt{true}]$ with c_1. Thus, $c_0[S = \texttt{true}], c_1, \ldots, c_k$ is a left-right path in $d[S = \texttt{true}]$, proving that it rewrites to a forbidden query.

Suppose d^L is of type 2, and let U be a left binary symbol that does not occur in c_1. Using the notations in Eq. 14, let m be the number of X-subcomponents of c_0, and n the number of subcomponents that contain U. By our assumption, $1 \leq n < m$. Let $B = \{b_1, \ldots, b_n\}$ be a set of n constants. We claim that the query $d[U =^L B]$ rewrites to a forbidden query, contradicting the fact that d was final. Start by noting that $c_i[U =^L B] = c_i$ for all $i \geq 1$, because c_i contains at least one central symbol. On the other hand, after applying the distributivity law in Eq. 14 we obtain a union of conjunctive queries. (They are all connected, because each has x as a root variable). From this union, we select only one query, namely:

$$c_0' = \exists x.(s_1[b_1/y_1] \wedge \ldots \wedge s_n[b_n/y_n] \wedge (\exists y_{n+1}.s_{n+1}) \wedge \ldots \wedge (\exists y_m.s_m))$$

The expression above is minimized, because it is isomorphic to c_0 up to the renaming $y_i \rightarrow b_i$ for $i = 1, n$ (this is the reason why we chose n distinct constants in B). Therefore, c_0' is a left component: indeed $s_1[b_1/y_1]$ has at least one unary symbol (namely $U(x, b_1)$), and there exists at least one binary symbol coming from s_m (since $m > n$). Therefore, the path $c_0', c_1, \ldots, c_k$ is a left-right path in $d[U =^L B]$ proving that the latter rewrites to a forbidden query. This contradicts the fact that d is minimized. □

COROLLARY 8.14. *Let d be a final query, c a left component, and $c_0, c_1, \ldots, c_k$ a strict path. Then $c, c_1, \ldots, c_k$ is also a strict path.*

PROOF. If d^L is of Type 1, then we have seen in the last step of the proof of Prop. 8.12 that every left component can be substituted for the first component of

any strict path. Suppose d^L is of Type 2, and consider any strict path $c_0, c_1, \ldots, c_k$. For any symbol U in c, U does not occur in c_1 (otherwise $c, c_1, \ldots, c_k$ is a strict path starting at c). We have seen in the proof of Prop. 8.13 that U must be ubiquitous in c_0. Thus, all symbols in c occur ubiquitously in c_0, i.e. they occur in all subcomponents. Thus, the first subcomponent of c_0 contains the atoms $U_1(x, y_1), U_2(x, y_1), U_3(x, y_1), \ldots$, where $U_1, U_2, \ldots$ are the symbols in c, which implies that there exists a homomorphism $c \rightarrow c_0$ contradicting the fact that d is minimized. □

8.3.4 *Example.* We illustrate the proof with several examples.

EXAMPLE 8.15. *Consider how we remove extra unary symbols:*

$$d = R(x), R_1(x), S_1(x, y) \vee S_1(x, y), S_2(x, y) \vee S_2(x, y), T(y) \vee S_2(x, y), T_1(y)$$

Rewrite it to $d \rightarrow d[R_1 = \texttt{true}, T_1 = \texttt{false}] \equiv h_2$ *(from Example 4.5). Clearly* h_2 *is a final query.*

Removing mixed type 1-and-2 components requires a careful case analysis. One simple case is query j *in Example 8.2: rewrite it to* $j \rightarrow j[S_2 = \texttt{true}]$ *and the new query is equivalent to* h_1*. For a more subtle example, consider:*

$$d = R(x), S_1(x, y_1), S_2(x, y_2) \vee S_1(x, y), S_2(x, y) \vee S_1(x, y), S(x, y) \vee S_2(x, y), S(x, y) \vee S(x, y), T(y)$$

Neither of the simple rewritings $d[S_2 = \texttt{true}]$ *or* $d[S_2 = \texttt{false}]$ *or* $d[R = \texttt{true}]$ *works here, because all these queries minimize to a safe query. Instead, we rewrite to* $d[S_2 =^L \{a\}]$*, which is:*

$$\begin{aligned} & R(x), S_1(x, y_1), S_2(x, a) \vee R(x), S_1(x, a), S_2(x, a) \vee S_1(x, a), S_2(x, a) \vee S_1(x, y), S(x, y) \vee \vee S(x, y), T(y) \\ = & R(x), S_1(x, y_1), S_2(x, a) \vee S_1(x, a), S_2(x, a) \vee S_1(x, y), S(x, y) \vee \vee S(x, y), T(y) \end{aligned}$$

Next, we rewrite by setting $S_1(x, a) = \texttt{false}$ *and* $S_2(x, a) = \texttt{true}$*: the result is equivalent to* h_2*.*

Finally, we show why the ubiquitous symbol needs to appear. If there is no ubiquitous symbol, then we can choose one binary symbol and make it unary. For example:

$$d = S_1(x, y_1), S_2(x, y_2) \vee S_1(x, y), S(x, y) \vee S_2(x, y), S(x, y) \vee S(x, y), T(y)$$

Then rewrite to $d[S_2 =^L \{a\}]$ *and the query becomes equivalent to* h_2*.*

8.4 The Signature Counting Problem

The proof of Theorem 8.1 is by reduction from the *signature counting problem*, SC, which we introduce here as a generalization of the #PP2DNF problem. The input to an SC problem is the same as to the #PP2DNF: two numbers $n_1, n_2 > 0$ and a set $E \subseteq [n_1] \times [n_2]$. The size of the input problem is given by $n = n_1 n_2$; all other parameters below are constants.

We are given two numbers $m_1, m_2 \geq 2$, and call the sets $LL = [m_1], RR = [m_2]$ the *left labels* and *right labels* respectively. Unlike n, the numbers m_1, m_2 are fixed, and not part of the input problem.

DEFINITION 8.16. *A* labeling *is a pair of functions $l = (l_1, l_2)$, where $l_1 : [n_1] \to LL$ and $l_2 : [n_2] \to RR$. There are $m_1^{n_1} \times m_2^{n_2}$ labelings.*

In the case of #PP2DNF, we have $m_1 = m_2 = 2$, and a labeling is a valuation, associating `false` or `true` values to each variable X_i and Y_j, $i \in [n_1], j \in [n_2]$.

DEFINITION 8.17. *A* central-, left-, and right- signature*, are strings:*

$$\begin{aligned} k^C \in& \{0, 1, \ldots, n\}^{LL \times RR} \\ k^L \in& \{0, 1, \ldots, n\}^{LL} \\ k^R \in& \{0, 1, \ldots, n\}^{RR} \end{aligned}$$

Every labeling $l = (l_1, l_2)$ defines uniquely a central, left, and right signature, namely:

$$\begin{aligned} (k^C(l))_{u,v} =& |\{(a, b) \in E \mid l_1(a) = u, l_2(y) = v\}| && \forall (u, v) \in LL \times RR \\ (k^L(l))_u =& |\{a \in [n_1] \mid l_1(a) = u\}| && \forall u \in LL \\ (k^R(l))_v =& |\{b \in [n_2] \mid l_2(b) = v\}| && \forall v \in RR \end{aligned}$$

DEFINITION 8.18. *We define four types of* signatures*:*

- *A signature of type 1-1 is $k = k^C$.*
- *A signature of type 1-2 is $k = (k^C, k^R)$.*
- *A signature of type 2-1 is $k = (k^L, k^C)$.*
- *A signature of type 2-2 is $k = (k^L, k^C, k^R)$.*

For fixed parameters m_1, m_2, and type, let Σ denote the set of all signatures. Note that $|\Sigma| \leq (n + 1)^{m_1 m_2 + m_1 + m_2}$.

Given a signature k, denote $\#k$ the number of labelings that have that signature, in other words:

$$\#k \ = \ |\{l \mid k(l) = k\}|$$

We denote $\#\Sigma = \{\#k \mid k \in \Sigma\}$ the set of all signature counts.

DEFINITION 8.19. *The* signature counting problem*, SC, is the following. Given n_1, n_2 and $E \subseteq [n_1] \times [n_2]$, compute all signature counts $\#\Sigma$.*

PROPOSITION 8.20. *SC is hard for $FP^{\#P}$.*

PROOF. The proof is by reduction from #PP2DNF. We show this only for the SC problem of type 1-1: the extension to SC problems of types 1-2, 2-1, 2-2 is straightforward. Consider first the case $m_1 = m_2 = 2$, and let $\Phi = \bigwedge_{(i,j) \in E} X_i Y_j$. Then a signature k consists of four numbers, $k = (k_{11}, k_{12}, k_{21}, k_{22})$: where k_{11} represents the number of clauses $X_i Y_j$ where both X_i, Y_j are false, k_{12} is the number of clauses where X_i is false and Y_j is true, etc. Clearly, Φ is true iff $k_{22} > 0$. A solution to the SC problems gives us, for each k, the number of truth assignments $\#k$ that satisfy this constraint. It follows:

$$\#\Phi = \sum_{k_{11}=0,n} \sum_{k_{12}=0,n} \sum_{k_{21}=0,n} \sum_{k_{22}=1,n} \#k$$

(Note that, if $k_{11} + k_{12} + k_{21} + k_{22} \neq n_1 n_2$, then by definition $\#k = 0$. In other words the sum above has many redundant terms. We do not care about that.) Furthermore, we can convert a solution to a SC problem with parameters $m_1 \geq 2, m_2 \geq 2$ into a solution of the SC problem with parameters $m_1 = m_2 = 2$ by marginalizing. Denoting Σ the set of all $(2,2)$-signatures and Σ' the set of all (m_1, m_2) signatures, we have:

$$\forall k \in \Sigma : \qquad \#k = \sum_{k' \in \Sigma' : \forall u,v \in [2], k'_{uv} = k_{uv}} \#k'$$

□

8.5 The Expansion Formula

Let d be a forbidden query. The proof of Theorem 8.1 for d is a reduction from a signature counting problem, $E \subseteq [n_1] \times [n_2]$, with signatures Σ. We will construct a certain databse D from E, then repeatedly vary the probabilities of its tuples and compute the probability of d on D. From these results we extract $\#\Sigma$: thus, we will have shown how an oracle for $P(d)$ can be used to compute $\#\Sigma$.

There are several steps in the reduction; in this section we show the first step. For every $a \in [n_1], b \in [n_2]$, we construct a certain database $D(a, b)$ (to be described in in Sect. 8.6), called a *block*. The constants in $D(a, b)$ are a, b and other constants, which are assumed to be disjoint from the sets $[n_1]$ and $[n_2]$. We will describe later exactly how to construct this block (as a union of zig-zag-zig blocks). For the purpose of this section we don't care about the internal structure of $D(a, b)$. In addition to $D(a, b)$, if the type of the query is 2,1, or 1,2, or 2,2 then we also construct blocks $D(a, \cdot)$ and $D(\cdot, b)$. Here $\cdot$ should be interpreted as a new constant, unique for a (b respectively): that is $D(a, \cdot) = D(a, d_a)$, where d_a is a fresh constant. The database D will be the union of all these blocks. The blocks have following properties, which we need in this section:

—All blocks are isomorphic.

—For $a \in [n_1], b \in n_2$, no binary symbol contains both a and b. That is, every binary symbols that contains $a \in [n_1]$ on the first position, has some other constant $\notin [n_2]$ on the second position.

—Two blocks $D(a, b)$ and $D(a', b')$ do not share any constants, except possibly the endpoints, e.g. $D(a, b), D(a, b')$ have in common the constant a. Similarly for the blocks $D(a, \cdot), D(\cdot, b)$. In particular, the only tuples that can be shared between blocks are of the form $R(a)$ and $T(b)$, for $a \in [n_1]$ and $b \in [n_2]$.

—For all $a \in [n_1], b \in [n_2]$ we set $P(R(a)) = P(T(b)) = 1/2$. All other probabilities (of binary tuples, and of unary tuples of the form $R(d)$ and $T(c)$) will be specified below.

Based on these assumptions only, we give in this section a formula that relates $P(d)$ and $\#\Sigma$, called the *expansion formula*.

Given two constants a, b, define the query $d^{-a,-b}$ the query obtained from d as follows: every left component c is replaced with $c \wedge (x \neq a)$; every right component c is replaced with $c \wedge (y \neq b)$; and every center component is left unchanged. That is, $d^{-a,-b}$ is identical to d, except that it does not allow the left components to map x to a, nor the right components to map y to b.

8.5.1 *Expansion formula of type 1-1.* We start with the case when $d = d^L \vee d^C \vee d^R$ is of type 1-1. Here the SC problem will be of type 1,1, and $m_1 = m_2 = 2$. Fix two constants a, b, and two labels $u \in [m_1]$, $v \in [m_2]$. Interpret u, v as truth values for $R(a)$ and $T(b)$ (recall that $1 = \texttt{false}$, $2 = \texttt{true}$). We will define below a query $d_{uv}(a, b)$ s.t. its probability is exactly the probability of d, conditioned on the fact that the truth values of $R(a)$, $T(b)$ are u, v respectively:

$$P(\neg d_{uv}(a, b)) = P(\neg d \mid R(a) \equiv u, T(b) \equiv v) \tag{15}$$

For that, first define:

$$d^L_u = d^L[R = u] \qquad\qquad d^R_v = d^R[T = v]$$

That is, d^L_u is obtained by substituting in d^L the relation symbol R with $\texttt{false}$, when $u = 1$, and with $\texttt{true}$, when $u = 2$ respectively; obviously, $d^L_1 \equiv \texttt{false}$. Abbreviating $d^L_u[a/x]$ with $d^L_u(a)$, we define, for every $u \in [2], v \in [2]$:

$$\begin{aligned} d_{uv}(a, b) &= d^L_u(a) \vee d^{-a,-b} \vee d^R_v(b) \\ y_{uv} &= P_{D(a,b)}(\neg d_{uv}(a, b)) \end{aligned} \tag{16}$$

The reader is invited to check that Eq. 15 holds. Thus, the probability y_{uv} represents the probability that d is false on the block $D(a, b)$, conditioned on the fact that the truth values of $R(a)$ and $T(b)$ are u, v respectively. Note that y_{uv} is the same for all blocks $D(a, b)$, i.e. it does not depend on the constants a, b, because all blocks are isomorphic.

Recall that for each tuple in the database D, $P(t)$ denotes its probability.. A possible world W is a subset $W \subseteq D$, and $P(W) = \prod_{t \in W} P(t) \cdot \prod_{t \in D-W}(1 - P(t))$. Then, by definition (Eq. 1):

$$P(\neg d) = \sum_{W \subseteq D : W \models \neg d} P(W) \tag{17}$$

Fix a labeling $l_1 : [n_1] \to [2]$, $l_2 : [n_2] \to [2]$. We say that a world W has labeling $l = (l_1, l_2)$, if forall $a \in [n_1], b \in [n_2]$, $W \models (R(a) \equiv l_1(a))$ and $W \models (T(b) \equiv l_2(b))$. Let E_l be the event "W has labeling $l = (l_1, l_2)$". Clearly, $P(E_l) = 1/2^{n_1+n_2}$. Conditioned on E_l, the event $W \models \neg d$ is the conjunction of the independent events $W(a, b) \models \neg d_{l_1(a), l_2(b)}(a, b)$, where $W(a, b) = W \cap D(a, b)$. That is, once we fixed the truth value for all $R(a)$'s and all $T(b)$'s (through the labeling l_1, l_2), we need to check that d is false on each block $D(a, b)$ separately; these events are independent, because the blocks share only their endpoints, and there the truth values of $R(a), T(b)$ are already fixed. Furthermore, if $(a, b) \in E$ (and edge in the bipartite graph $E \subseteq [n_1] \times [n_2]$), then the probability of this event is $y_{l_1(a), l_2(b)}$; if $(a, b) \notin E$, then it is 1 (because the block $D(a, b)$ is empty, hence the query is false). Recall that, forall $u \in [2], v \in [2]$, $k_{uv}(l)$ denotes the number of edges in E whose endpoints are labeled with u and v respectively. Then:

$$P(\neg d|E_l) = \prod_{(a,b)\in E} y_{l_1(a)l_2(b)} = \prod_{u\in[2],v\in[2]} y_{uv}^{k_{uv}(l)}$$
$$P(\neg d) = \frac{1}{2^{n_1+n_2}} \sum_l P(\neg d|E_l) = \frac{1}{2^{n_1+n_2}} \sum_{k\in\Sigma} \#k \prod_{u\in[2],v\in[2]} y_{uv}^{k_{uv}} \quad (18)$$

We call Eq. 18 the *expansion formula* for a query d of type 1,1. This is a polynomial in the four variables $\bar{y} = (y_{11}, y_{12}, y_{21}, y_{22})$, and it says this. To compute $P(\neg d)$, proceed as follows. Fix a generic block $D(a,b)$. Set the truth values of $R(a), T(b)$ to all four combinations of `false`/`true`, and for each, compute the probability of $\neg d$ conditioned on that setting of $R(a), T(b)$: this probability is y_{uv}. Then compute the polynomial Eq. 18, where the coefficients $\#k$ are exactly the signature counts of the SC problem. More precisely, the coefficient of a monomial $y_{11}^{k_{11}} \cdot y_{12}^{k_{12}} \cdot y_{21}^{k_{21}} \cdot y_{22}^{k_{22}}$ is the number of labelings that have signature $k = (k_{11}, k_{12}, k_{21}, k_{22})$, i.e. $\#k$. We want to use this process in reverse: given an oracle for $P(\neg d)$, compute the coefficients $\#k$. For that, we vary the probabilities of the block $D(a,b)$, to generate $(n+1)^4$ distinct values for $\bar{y}$ and obtain a linear system with $(n+1)^4$ equations and $(n+1)^4$ unknowns. Let M be the matrix of this system. If M is non-singular, then we can solve for all $\#k$: then we have used the oracle $P(d)$ to solve the SC problem, proving that $P(d)$ is hard for $FP^{\#P}$. Notice that if we could choose values for $y_{11}, y_{12}, y_{21}, y_{22}$ independently, by choosing $n+1$ distinct values for each variable, then forming all $(n+1)^4$ combinations, then M is the Kronecker product of 4 Vandermonde matrices, and is non-singular. In general we cannot choose the y's independently, we can only vary probabilities in the block $D(a,b)$. We show in Sect. 8.6 that the matrix is non-singular.

8.5.2 *Expansion formula of type 2-2.* Next, we will derive the expansion formula for a query d of type 2,2. Here there are no unary symbols. For any constant $a \in [n_1]$, the query $d^L[a/x]$ is a union of conjunctive queries, and similarly $d^R[b/y]$. Denote their CNF lattices $LL = L(d^L[a/x])$, $RR = L(d^R[b/y])$. Let $m_1 = |LL - \{\hat{1}\}|$; by renaming the lattice elements, we assume wlog $LL = [m_1] \cup \{\hat{1}\}$. The left labels in our SC problem will be $[m_1]$. Repeating the same for the right end, we define $RR = [m_2] \cup \{\hat{1}\}$. Recall that for each $u \in [m_1]$, d_u^L denotes the query associated to the lattice element u in LL. Since in this query all atoms are of the form $S(a,y)$, i.e. there is no x variable, and there is a single y-variable, we will always write it as $d_u(a)$, to remind that the constant a appears in all atoms. Similarly, for each $v \in [m_2]$, $d_v^R(b)$ denotes a query associated to the lattice element v in RR.

Thus:

$$d^L[a/x] \equiv \bigwedge_{u\in[m_1]} d_u^L(a) \qquad\qquad d^R[b/y] \equiv \bigwedge_{v\in[m_2]} d_v^R(b) \quad (19)$$

Define:

$$d_{uv}(a,b) = d_u^L(a) \vee d^{-a,-b} \vee d_v^R(b) \qquad d_{u\cdot} = d_u^L(a) \vee d^{-a} \qquad d_{\cdot v} = d^{-b} \vee d_v^L(b)$$
$$y_{uv} = P_{D(a,b)}(\neg d_{uv}(a,b)) \qquad z_{u\cdot} = P_{D(a,\cdot)}(\neg d_{u\cdot}) \qquad z_{\cdot v} = P_{D(\cdot,b)}(\neg d_{\cdot v}) \quad (20)$$

The intuition is that $d_{uv}(a, b)$ is obtained from d by imposing that the end-points a and b satisfy the labels $d^L_u(a)$ and $d^R_v(b)$ respectively.

EXAMPLE 8.21. *We illustrate with two examples. We only show* d^L *in both cases, since the left labels LL depend only on* d^L.

$$d^L = S(x, y_1)S_1(x, y_1), S(x, y_2)S_2(x, y_2), S(x, y_3)S_3(x, y_3)$$

The left labels are:

$$\begin{aligned} d^L_1(a) =& S(a, y), S_1(a, y) \\ d^L_2(a) =& S(a, y), S_2(a, y) \\ d^L_3(a) =& S(a, y), S_3(a, y) \end{aligned}$$

and their closure under disjunctions. There are $m_1 = 7$ *left labels.*

Consider a query with two left components:

$$d^L = S(x, y_1), S_1(x, y_1), S(x, y_2), S_2(x, y_2) \vee S(x, y_1), S_1(x, y_1), S(x, y_3), S_3(x, y_3)$$

The labels are:

$$\begin{aligned} d^L_1(a) =& S(a, y), S_1(a, y) \\ d^L_2(a) =& S(a, y), S_2(a, y) \vee S(a, y), S_3(a, y) \end{aligned}$$

and $d^L_1(a) \vee d^L_2(a)$. *There are 3 left labels.*

Let $l_1 : [n_1] \to [m_1]$ and $l_2 : [n_2] \to [m_2]$ be a labeling. We say that a world W satisfies this labeling, if forall $a \in [n_1]$, $W \models \neg d^L_{l_1(a)}(a)$ and forall $b \in [n_2]$, $W \models \neg d^R_{l_2(b)}(b)$. Let $E_{l_1 l_2}$ denote the event "W satisfies the labeling l_1, l_2".

There are two important differences from the case (type 1-1). First, some worlds do not satisfy any labeling. However, we observe that, if $W \models \neg d$, then W does satisfy some labeling. Indeed, let $a \in [n_1]$. Since $W \models \neg d$, we have $W \models \neg d^L[a/x]$, which implies that there exists $u \in [m_1]$ s.t. $W \models \neg d^L_u(a)$ (because of Eq. 19). Similarly, there exists $v \in [m_2]$ s.t. $W \models \neg d^L_v(b)$. Set $l_1(a) = u, l_2(b) = v$, and repeat this forall a, b: thus, W satisfies the labeling l_1, l_2. The second difference is that the events $E_{l_1 l_2}$ are no longer disjoint. However, they form a lattice, and we will use Mobius' inversion formula to express the probability, instead of conditioning on disjoint events. To do that, we need a brief review of lattice theory.

Given two lattices L_1, L_2, their product $L_1 \times L_2$ is also a lattice. Furthermore, $\mu_{L_1 \times L_2}((u, v), (x, y)) = \mu_{L_1}(u, x) \cdot \mu_{L_2}(v, y)$. The *strict product* of L_1 and L_2 is:

$$(L_1 \times L_2)^0 = (L_1 - \{\hat{1}\}) \times (L_2 - \{\hat{1}\}) \cup \{\hat{1}\}$$

This is indeed a lattice, because $L_i - \{\hat{1}\}$ is a meet semilattice, for $i = 1, 2$, hence $(L_1 - \{\hat{1}\}) \times (L_2 - \{\hat{1}\})$ is a meet semilattice, and after completing with $\hat{1}$ it becomes a lattice. We have:

LEMMA 8.22. $\mu_{(L_1 \times L_2)^0}((u, v), \hat{1}) = -\mu_{L_1}(u, \hat{1}) \cdot \mu_{L_2}(v, \hat{1})$.

Proof.

$$\mu_{(L_1\times L_2)^0}((u,v),\hat{1}) = -\sum_{(u,v)\leq(x,y)<\hat{1}} \mu_{(L_1\times L_2)^0}((u,v),(x,y))$$
$$= -\sum_{u\leq x<\hat{1}, v\leq y<\hat{1}} \mu_{L_1}(u,x)\times\mu_{L_2}(v,y) = -\mu_{L_1}(u,\hat{1})\cdot\mu_{L_2}(v,\hat{1})$$

Here, we used the fact that the interval $[(u,v),(x,y)]$ of the strict product $(L_1\times L_2)^0$ is isomorphic to the same interval in the regular product $L_1\times L_2$, because $x<\hat{1}$ and $y<\hat{1}$. □

Given a lattice L, we denote $(L^n)^0$ the strict cartesian product $(L\times\ldots\times L)^0$. It follows from the lemma that $\mu_{(L^n)^0}(\bar{u},\hat{1}) = -(-1)^n\prod_i \mu_L(u_i,\hat{1})$.

Since each label $l_1(a)$ belongs to $LL-\{\hat{1}\}$, it follows that the space of all left labelings l_1 is the strict product $(LL^{n_1})^0$. In other words, every element of $(LL^{n_1})^0$, except $\hat{1}$, corresponds[7] to a labeling l_1. The Mobius function in this lattice is $\mu(l_1,\hat{1}) = -(-1)^{n_1}\prod_{a\in[n_1]}\mu(l_1(a),\hat{1})$. Similarly, l_2 is in the strict lattice product $(RR^{n_2})^0$, while (l_1,l_2) is in the strict product of the two lattices, hence $\mu((l_1,l_2),\hat{1}) = -\mu(l_1,\hat{1})\cdot\mu(l_2,\hat{1})$.

The last observation that we need to make is that, for a fixed labeling l_1,l_2, the event $W\models E_{l_1,l_2}\wedge\neg d$ is the conjunction of the following: independent events:

$$(\bigwedge_a W(a,\cdot)\models\neg d_{l_1(a)\cdot})\wedge(\bigwedge_b W(\cdot,b)\models\neg d_{\cdot l_2(b)})\wedge(\bigwedge_{a,b} W(a,b)\models\neg d_{l_1(a)l_2(b)})$$

The probabilities of the events above are $z_{l_1(a)\cdot}$, $z_{\cdot l_2(b)}$, and $y_{l_1(a)l_2(b)}$ respectively, see Eq. 20.

Recall that in the case of a signature counting problem of type 2-2, every signature k consists of the left-, center-, and right signature, k^L,k^C,k^L. Therefore:

$$P(\neg d) = P(\bigvee_{l_1,l_2} E_{l_1l_2}\wedge\neg d) = -\sum_{l_1,l_2}(-1)\mu(l_1,\hat{1})\cdot\mu(l_2,\hat{1})P(E_{l_1l_2}\wedge\neg d)$$
$$= (-1)^{n_1+n_2}\sum_{l_1,l_2}(\prod_a \mu(l_1(a),\hat{1})\cdot z_{l_1(a)\cdot})\cdot(\prod_b \mu(l_2(b),\hat{1})\cdot z_{\cdot l_2(b)})\cdot(\prod_{a,b} y_{l_1(a)l_2(b)})$$
$$= (-1)^{n_1+n_2}\sum_{k\in\Sigma}\#k\cdot\prod_{u\in[m_1]}(\mu(u,\hat{1})\cdot z_{u\cdot})^{k_u^L}\cdot\prod_{v\in[m_2]}(\mu(v,\hat{1})\cdot z_{\cdot v})^{k_v^R}\cdot\prod_{u\in[m_1],v\in[m_2]} y_{uv}^{k_{uv}^C}$$

Here $\mu(u,\hat{1})$ is taken in the left lattice LL, while $\mu(v,\hat{1})$ is in the right lattice RR. Now it should become clear why we introduced the blocks $D(a,\cdot)$, $D(\cdot,b)$, and their associated variables $z_{u\cdot},z_{\cdot v}$: they absorb the powers of the mobius function. Note that when $\mu(u,\hat{1})=0$, then the only terms that contribute to the sum above are those for which the exponent $k_u^L=0$. This means that we can simply ignore the labels $u\in LL$ whose Mobius function is zero. Decreasing m_1 appropriately, we will assume that forall $u\in[m_1]$, $\mu(u,\hat{1})\neq 0$. (We no longer need a lattice structure on

[7]If we took l_1 to range over L^n, then we need to exclude many more elements other than $\hat{1}$, and then the Mobius inversion formula no longer applies.

$[m_1]$.) Similarly for m_2. Thus, denoting:

$$y_{u\cdot} = \mu(u, \hat{1}) \cdot z_{u\cdot} \qquad \text{for } u \in [m_1]$$
$$y_{\cdot v} = \mu(v, \hat{1}) \cdot z_{\cdot v} \qquad \text{for } v \in [m_2]$$

we obtain the following expansion formula for $P(\neg d)$ in the case when d is of type 2-2:

$$P(\neg d) = (-1)^{n_1+n_2} \sum_{k \in \Sigma} \#k \cdot \prod_{u \in [m_1]} y_{u\cdot}^{k_u^L} \cdot \prod_{v \in [m_2]} y_{\cdot v}^{k_v^R} \cdot \prod_{u \in [m_1], v \in [m_2]} y_{uv}^{k_{uv}^C} \qquad (21)$$

Thus, as in the 1-1 case, we have expressed $P(\neg d)$ in terms of $m_1 m_2 + m_1 + m_2$ variables $\bar{y}$. As before, we will probe $D(a, b)$ with different probabilities, and form a system of linear equations, with unknowns $\#k$. Its matrix is M. We will show in the next two sections that this matrix is non-singular.

8.5.3 *Expansion formula of types 2-1 and 1-2.* If the query d is of type 2-1 or 1-2, then we use a mixed expansion formula, which is a straightforward comination of the previous two, and omitted.

8.6 Non-Zero Jacobian

This section is at the core of our hardness proof for forbidden queries. So far, we have only assumed that both d^L and d^R are non-empty. In a forbidden query d, they must also be symbol-connected, i.e. connected in the co-occurrence graph: otherwise the query is not forbidden; for example, the left and right in $R(x), S_1(x, y) \vee S_2(x, y), T(y)$ are not connected, and this query is safe. We have not assumed so far that the left and right are connected. The step we describe in this section works if and only if d^L and d^R are connected. In other words, here is where we assume that d is forbidden.

Let m be the number of y-variables in the expansion formula ($m = 4$ for type 1-1, $m = m_1 m_2 + m_1 + m_2$ for type 2-2, etc). We sometimes write these variables as $\bar{y} = (y_1, \ldots, y_m)$ instead of their official index y_{uv}, and will switch back and forth between the two notations. Each y_i is a probability of a query, $\neg d_{uv}(a, b)$, on the block $D(a, b)$, see Eq. 16 and Eq. 20. For each tuple $t_j \in D(a, b)$, denote $x_j = P(\neg t_j)$, then $y_i = F_i(x_1, x_2, \ldots)$ is a multi-linear polynomial. Denote $\bar{F} = (F_1, \ldots, F_m)$.

In this section we show how to construct the block $D(a, b)$ (and $D(a, \cdot)$ and $D(\cdot, b)$) and choose m distinguished tuples in $D(a, b)$ with probabilities $\bar{x} = (x_1, \ldots, x_m)$ (i.e. $x_i = P(\neg t_i)$), with the following property: the Jacobian of the function $\bar{x} \mapsto \bar{F}$ is non-zero. That is, we will construct $D(a, b)$ such that:

PROPOSITION 8.23. *There are m tuples in $D(a, b)$ with associated probabilities $\bar{x}$, such that the Jacobian of the function $\bar{x} \mapsto \bar{y}$ is non-zero at some point in $\bar{x} \in [0, 1]^m$.*

We use this proposition in Sect. 8.7 to prove that M is non-singular: the idea is that the image of the function $\bar{x} \mapsto \bar{y}$ contains an open hypercube of dimension m, hence we can vary each dimension y_i independently, to obtain $(n + 1)^m$ values

in a grid in this hypercube: for these points, M is the Kronecker product of Vandermonde matrices, hence is non-singular. Thus, it is important that the Jacobian $\bar{x} \mapsto \bar{y}$ be non-zero: otherwise the points $\bar{y}$ may lie in a space of lower dimension and then the matrix M is always singular. In the remainder of this section we prove Prop. 8.23.

8.6.1 *The Parallel Sub-blocks* $D(a,b) = \bigcup_i D_i(a,b)$. The $m_1 m_2$ variables y_{uv} depend only on the probabilities in the block $D(a,b)$, while the variables $y_{u\cdot}$ and $y_{\cdot v}$ depend on the blocks $D(a,\cdot)$ and $D(\cdot,b)$. Thus, there are three functions $\bar{x} \mapsto \bar{y}$. It suffices to prove separately that each of these three functions has a non-zero Jacobian: then the combined function also has a non-zero Jacobian. Hence, for the rest of this subsection we will focus only on y_{uv}, and their corresponding block $D(a,b)$; the variables $y_{u\cdot}$ and $y_{\cdot v}$ are treated similarly and omitted.

We construct $D(a,b) = D_1(a,b) \cup D_2(a,b) \cup \ldots \cup D_{m_1 m_2}(a,b)$, to be a union of "sub-blocks". All sub-blocks are disjoint, except for the endpoints a and b: that is, they share the unary symbols $R(a)$, $T(b)$ (if such symbols exists). All sub-blocks are isomorphic. We will choose one distinguished tuple t_1 in $D_1(a,b)$ and assign probability $x_1 = P(\neg t_1)$. (We show below how to choose t_1.) Let $t_2, \ldots, t_m$ the tuples in $D_2(a,b), \ldots, D_{m_1 m_2}(a,b)$ that correspond to t_1 through the isomorphism, and let $x_i = P(\neg t_i)$, for $i = 1, m_1 m_2$. That is, each sub-block has one distinguished tuple, which correspond through the isomorphism, and their probabilities are given by different variables. For every other tuple, its probability will be the same in all sub-blocks (and its value chosen in a way we show below). Recall that $y_{uv} = P_{D(a,b)}(\neg d_{uv}(a,b))$. For any world $W \subseteq D(a,b)$, the event $W \models \neg d_{uv}(a,b)$ is the conjunction of the $m_1 m_2$ events $W \cap D_i(a,b) \models \neg d_{uv}(a,b)$, and these are independent. This is because each component of the query $d_{uv}(a,b)$ has at most one x- and at most one y-variable, hence every valuation maps its atoms to tuples that belong to a single block $W \cap D_i(a,b)$. Denote $f_{uv}(x_i) = P_{D_i(a,b)}(\neg d_{uv}(a,b))$. Since all probabilities except for the distinguished tuples are fixed, this function is a linear function in x_i, $f_{uv}(x_i) = a_{uv} x_i + b_{uv}$, where a_{uv}, b_{uv} are real constants. Thus, based on our discussion so far:

$$y_{uv} = P_{D(a,b)}(\neg d_{uv}(a,b)) = \prod_i P_{D_i(a,b)}(\neg d_{uv}(a,b)) = \prod_i f_{uv}(x_i) \tag{22}$$

Thus, each y_{uv} is the product of linear functions. We show that the Jaobian of such products is non-zero.

8.6.2 *The Jacobian of a Product of Linear Functions.* Two polynomials f, g are called *equivalent* if there exists a constant $c \neq 0$ s.t. $f = c \cdot g$. Call a linear polynomial $ax + b$ *non-degenerate* if it is not equivalent to 1 (in other words, $a \neq 0$). We prove the following:

PROPOSITION 8.24. *Let $f_i(x) = a_i x + b_i$, $i = 1, m$ be m inequivalent, non-degenerate linear polynomials. Define the following m multivariate polynomials, where $x_1, \ldots, x_m$ are m distinct variables:*

$$F_i(x_1, \ldots, x_m) = f_i(x_1) \cdot f_i(x_2) \cdots f_i(x_n)$$

Let $J(\bar{x}) = D(\bar{F})/D(\bar{x})$ be the Jacobian. Let $\bar{v} = (v_1, \ldots, v_m) \in R^m$ be any m distinct values (i.e. $v_i \neq v_j$ for $i \neq j$). Then $det(J(\bar{v})) \neq 0$.

PROOF. We can assume w.l.o.g. that $a_1 = \ldots = a_m = 1$. Then the polynomials can be written as $f_i(x) = x + y_i$, where $y_1, \ldots, y_n$ are distinct values (because the polynomials are inequivalent), and the Jacobian J is:

$$J = \left(\prod_{k \neq j} (x_k + y_i) \right)_{ij}$$

Its determinant can be computed from *Cauchy's double alternant* [Krattenthaler 1999]. Its value is:

$$det(J) = \prod_{i<j} (x_i - x_j)(y_i - y_j)$$

□

To prove Prop. 8.23 it suffices to show that the linear functions $f_{uv}(x_i)$ are non-equivalent, and non-degenerate. Then the Jacobian of Eq. 22 is non-zero for any choice of distinct probabilities $x_i \neq x_j$, for $i \neq j$.

8.6.3 *Form Multilinear to Linear and the Role of Irreducible Factors.* Now we revisit our assumption that all tuples in $D_1(a, b)$ have fixed probabilities. We will let them vary as follows. For each tuple t in $D_1(a, b)$, we choose a distinct variable to denote $P(\neg t)$. Then we fix the same probability for all isomorphic copies of t in $D_1(a, b), D_2(a, b), \ldots$ Thus, there are $|D_1(a, b)|$ free variables, one of which is our distinguished variable. We ask the following question: to what values do we need to set the other $|D_1(a, b)| - 1$ variables such that the resulting linear polynomials f_{uv} are non-equivalent and non-degenerate.

Given m multilinear polynomials $F_1, \ldots, F_m$, over the same set of variables, call a variable x *distinguished* if one can substitute the other variables with some values (same value in all polynomials) such that all resulting linear polynomials are non-degenerate and non-equivalent. For example, in $F_1 = x_1 x_2$ and $F_2 = x_1 x_2 + x_1$ the variable x_2 is distinguished: setting $x_1 = 1$ results in the linear polynomials x_2 and $x_2 + 1$, which are non-equivalent. But x_1 is not distinguished: setting, say, $x_2 = 1$ gives us x_1 and $2x_1$, which are equivalent.

Recall that a multi-variate polynomial F is irreducible if whenever $F = G \cdot H$ then it is equivalent to either G or H. A classical theorem in algebra is that every multi-variate polynomial over a field admits a unique factorization as a product of irreducible polynomials.

PROPOSITION 8.25. *Let $F_1, \ldots, F_m$ multi-linear polynomials over the same set of variables. Then a variable x is distinguished iff each F_i depends on x, and, denoting G_i the irreducible factor in F_i that contains x, the polynomials $G_1, \ldots, G_m$ are non-equivalent.*

In the example above, the factorization is $x_1 x_2$ and $x_1(x_2 + 1)$. The factors containing x_1 are identical (hence x_1 is not distinguished) while the factors containing x_2 differ (hence x_2 is distinguished).

PROOF. The "only if" direction is trivial: if two factors G_i, G_j are equivalent, then for any substitution of the variables other than x, the two polynomials F_i and

F_j are also equivalent. For the "if" direction, write $G_i(x) = a_i * x - b_i$ where a_i, b_i are polynomials that depend on all variables other than x: denote these other variables $\bar{y}$. We have $a_i \not\equiv 0$ because G_i depends on x. We prove that for any distinct i, k, the polynomials $a_i b_k$ and $a_k b_i$ are not identical. Suppose otherwise, i.e. $a_i b_k \equiv a_k b_i$. Then we can factorize them like this:

$$a_i = u \cdot v$$
$$b_k = w \cdot z$$
$$a_k = u \cdot z$$
$$b_i = v \cdot w$$

Then $G_i(x) = u \cdot v \cdot x + v \cdot w = v \cdot (u \cdot x + w)$ and $G_k(x) = z \cdot (u \cdot x + w)$. We know that $u \not\equiv 0$, because $a_i \not\equiv 0$ (and $a_k \not\equiv 0$). Since both G_i and G_k are irreducible, both v and z must be constants. But then G_i and G_k are equivalent, which is a contradiction. This proves the claim that $a_i b_k \not\equiv a_k b_i$.

Consider the following polynomial:

$$H(\bar{y}) = (\prod_i a_i) \times (\prod_{ik} (a_i * b_k - a_k * b_i))$$

Here $H(\bar{y})$ is a multivariate polynomial in variables $\bar{y}$ (that is, all variables except x), which is not identically zero. Hence there are values $\bar{y} = \bar{v} \in (0,1)^n$ s.t. $H[\bar{v}/\bar{y}] \neq 0$. (Otherwise, if H is zero on an open set, then it is identically zero). We check that these values satisfy the two conditions in the theorem. Indeed, we have $a_i[\bar{v}/\bar{y}] \neq 0$, hence $a_i[\bar{v}/\bar{y}] \cdot x - b_i[\bar{v}/\bar{y}]$ is not degenerate. Next, note that, forall $i \neq k$, $a_i[\bar{v}/\bar{y}] b_k[\bar{v}/\bar{y}] \neq a_i[\bar{v}/\bar{y}] b_i[\bar{v}/\bar{y}]$, which means that the two polynomials i, k are inequivalent. □

Thus, in order to prove Prop. 8.23, it remains to show that we can choose in the sub-block $D_1(a, b)$ a distinguish tuple s.t., denoting $x = P(\neg t)$ (we drop the index i from x_i from now on), the variable x is distinguished in the set of polynomials f_{uv}, $u \in [m_1], v \in [m_2]$. Before we do this, we remark that, if d is not a forbidden query, then we cannot satisfy the condition in this proposition. For example, consider the query $d = R(x), S_1(x, y) \vee S_2(x, y), T(y) = d^L \vee dR$. Since d^L and d^R do not share any common symbols, one can see that f_{uv} is expressed as $g_u \cdot h_v$, where g_u depends only on d^L and h_v depends only on d^R. Any irreducible factor of g_u will occur in both f_{uv_1} and f_{uv_2}; any irreducible factor of h_u will occur in both $f_{u_1 v}$ and $f_{u_2 v}$.

8.6.4 *Positive Boolean Expressions and Irreducible Polynomials.* Here we make a short detour to establish some background on the connection between positive Boolean expressions and irreducible polynomials. Let Φ be a positive Boolean expression in CNF, over variables X_i, $i = 1, n$. Thus, Φ is a conjunction of clauses C_i, and each clause is a disjunction of variables $X_{i_1} \vee X_{i_2} \vee \ldots$. We assume no clause is redundant. The *primal* graph, $G(\Phi)$ has a node for each variable X_i, and an edge (X_i, X_j) whenever the two variables co-occur in a clause. The primal graph is sometimes also called the co-occurrence graph, but we prefer to reserve the latter term for DNF expressions. A connected component C of $G(\Phi)$ corresponds to the Boolean expression Φ^C, namely the conjunction of all clauses whose variables are in C. If each X_i is set independently to `true` with probability $x_i \in [0, 1]$, then $P(\Phi)$

is given by a multilinear polynomial denoted $F_\Phi(x_1, \ldots, x_n)$.

PROPOSITION 8.26. *F is irreducible iff $G(\Phi)$ is connected. More generally, for each variable x_i, let C be the connected component in $G(\Phi)$ that contains X_i. Then the irreducible factor of F that contains x_i is F_{Φ^C}.*

PROOF. If $G(\Phi)$ is disconnected, then we can write $\Phi = \Phi_1 \wedge \Phi_2$ where Φ_1, Φ_2 do not share any variables. Hence, $F_\Phi = F_1 \cdot F_2$, where $F_i = F_{\Phi_i}$, $i = 1, 2$. Conversely, if $F_\Phi = F_1 \cdot F_2$, then define Φ_1 to be the obtained from Φ by substituting all variables in F_2 with `true`; similarly Φ_2 is obtained from Φ by substituting all variables in F_1 with `true`. We claim $\Phi \Rightarrow \Phi_1 \wedge \Phi_2$. For the converse, consider a truth assignment s.t. $\Phi_1 =$ `true` and $\Phi_2 =$ `true`, and suppose $\Phi =$ `false`. Assign to the real variables x_i the values 0 or 1 according to this truth assignment. Hence $F_\Phi = F_1 \cdot F_2 = 0$. Assume wlog $F_1 = 0$. Then set all variables occurring in F_2 to `true` (and 1 respectively): in the new assignment we sill have $F_\Phi = F_1 F_2 = 0$, hence $\Phi =$ `false`, and now by definition $\Phi = \Phi_1$, contradiction. This proves the first part. For the second part, let $C_1, \ldots, C_m$ be all connected components of the graph, then $\Phi = \Phi^{C_1} \wedge \ldots \wedge \Phi^{C_m}$. Then $F_\Phi = \prod_i F_{\Phi^{C_i}}$ and each $F_{\Phi^{C_i}}$ is irreducible, proving our claim. □

Next, consider Φ to be a positive formula in DNF, over variables X_i. We call its DNF co-occurrence graph to be the graph where the variables are nodes, and edges a pair of variables that co-occur in a minterm. The *dual* of Φ is the following formula, over new variables X_i^d: $\Phi^d = \neg\Phi[\neg X_i^d / X_i; i = 1, n]$. That is, we negate each variable, and negate the expression too. Clearly, the dual is also positive, and the primal graph of Φ^d is isomorphic to the co-occurrence graph of Φ (because the CNF of Φ^d can be obtained from the DNF of Φ by replacing $\wedge \to \vee$ and $\vee \to \wedge$). In our problem, the probabilities y_i given by Eq. 16 and Eq. 20 are the probability of dual expressions of the DNF lineage formulas. Thus, to reason about the irreducibles of the polynomials y_i, it suffices to examine the co-occurrence graphs of lineage expressions, which are naturally given in DNF.

8.6.5 *Constructing the Sub-block $D_1(a, b)$.* Finally, we show how to construct the sub-block $D_1(a, b)$ (recall that all other sub-blocks $D_2(a, b), \ldots, D_m(a, b)$ are isomorphic, where $m = m_1 m_2$). We take $D_1(a, b)$ to be the zig-zag-zig block defined in Sect. 8.2. Denoting n_x, n_y the maximum number of x-, and y-variables in every component, the set of constants in zig-zag-zig block is $\{a\} \cup C \cup D \cup \{b\}$, where $C = \{c_1, \ldots, c_{n_x}\}$ and $D = \{d_1, \ldots, d_{n_y}\}$, and there are three kinds of binary tuples: *Zig-tuples* $S(a, d_j)$; *Zag-tuples* $S(c_i, d_j)$; and *Zig-tuples* $S(c_i, b)$, for $i = 1, n_x$ and $j = 1, n_y$. Furthermore, if d^L has type 1, then there exists also the unary tuple $R(c)$ (in this case $n_x = 1$, and we denote $c_1 = c$). Similarly, if d^R has type 1, then the zig-zag-zig block contains $T(d)$. We choose a Zag-tuple $t_0 = S(c_i, d_j)$ arbitrarily, and designated it as *distinguished tuple* in $D_1(a, b)$. Let $x_0 = P(\neg t_0)$ be its probability.

Let Y_{uv} be the lineage of $d_{uv}(a, b)$ on $D_1(a, b)$ (Eq. 4). The lineage is defined over a set of Boolean variables X_t, one for each tuple $t \in D_1(a, b)$. To simplify the notation, we will denote the Boolean variable X_t with t; thus, we will refer to tuples in $D(a, b)$ and Boolean variables interchangeably. Then $y_{uv} = P_{D_1(a,b)}(\neg d_{uv}) = F_{Y_{uv}^d}$. In other words, y_{uv} is the multi-linear polynomial associated to the dual of Y_{uv}. For

each u, v, denote C_{uv} the connected component of the co-occurrence graph of the DNF expression Y_{uv} that contains the distinguished tuple t_0, and let $Y_{uv}^{C_{uv}}$ denote the DNF expression Y_{uv} restricted to tuples in C_{uv}. To prove that the variable x_0 is distinguished for the polynomials y_{uv}, $u \in [m_1], v \in [m_2]$ (Prop. 8.25) we need to show that the Boolean expressions $Y_{uv}^{C_{uv}}$, for $u \in [m_1], v \in [m_2]$ are inequivalent: then, by Prop. 8.26, the irreducible factors in F_{uv} that contain x_0 are inequivalent, and therefore we can apply Prop. 8.25.

We examine the linage Y_{uv} closer. Recall that $d_{uv}(a, b) = d_u^L(a) \vee d^{-a,-b} \vee d_v^R(b)$, for both Type 1,1 queries (Eq. 16) and Type 2,2 queries (Eq. 20). All components in $d_u^L(a)$ have a single variable, y; $d^{-a,-b}$ is the same as d, except that all left components have the additional predicate $x \neq a$ and all right components have the additional predicate $y \neq b$; and the components in $d_v^R(b)$ have a single variable, x. For any component c in $d_{uv}(a, b)$, let X denote the set of its x-variables and Y the set of its y variables: define $c(C, D) = \bigvee_{\theta_1,\theta_2} c[\theta_1, \theta_2]$, where $\theta_1 : X \rightarrow C$ maps all X variables to constants in C, and $\theta_2 : Y \rightarrow D$ maps all Y variables to constants in D. We extend this notation to unions of components. Then the lineage is:

$$\begin{aligned} Y_{uv} = \; & d_u^L(a, D) & \vee d^C(a, D) \quad & \vee d^R(\{a\} \cup C, D) \\ & & \vee d^C(C, D) \quad & \\ \vee \; & d^L(C, D \cup \{b\}) & \vee d^C(C, b) \quad & \vee d_v^R(C, b) \end{aligned}$$

The reader can verify that this is indeed the correct lineage expression, by tracing where the variables in $d_{uv}(a, b)$ can be mapped to the zig-zag-zig block $D(a, b)$.

We start by proving that any two Boolean expressions in the set $\{Y_{uv} \mid u \in [m_1], v \in [m_2]\}$ are in-equivalent.

Proposition 8.27. *Let $u_1, u_2 \in [m_1]$ and $v_1, v_2 \in [m_2]$. If the logical implication $Y_{u_1v_1} \Rightarrow Y_{u_2v_2}$ holds, then the following implications hold between Boolean expressions: $d_{u_1}^L(a, D) \Rightarrow d_{u_2}^L(a, D)$ and $d_{v_1}^R(C, b) \Rightarrow d_{v_2}^R(C, b)$. Furthermore, the following implications hold between queries: $d_{u_1}^L(a) \Rightarrow d_{u_2}^L(a)$ and $d_{v_1}^R(b) \Rightarrow d_{v_2}^R(b)$.*

In particular, if $Y_{u_1v_1} \equiv Y_{u_2v_2}$ then the following queries are equivalent: $d_{u_1}^L(a) \equiv d_{v_1}^L(a)$. Hence $u_1 = u_2$. Similarly, $v_1 = v_2$.

Proof. We keep unchanged all tuples of the form $S(a, c_i)$ where S is a left symbol, set all other tuples $t = \texttt{false}$ in both $Y_{u_1v_1}$ and $Y_{u_2v_2}$. Recall that d^C is the union of all central components of d, i.e. where both x and y are root variables. Denote d^{LC} the union of all central components consisting only of left symbols[8]. We have:

$$Y_{u_1v_1} \equiv d_{u_1}^L(a, D) \vee d^{LC}(a, D) \Rightarrow Y_{u_2v_2} \equiv d_{u_2}^L(a, D) \vee d^{LC}(a, D)$$

Every minterm on the left must imply a minterm on the right. However, a minterm $s_i(a, d_j)$ in $d_{u_1}^L$ is some X-subcomponent $s_i(x, y)$ of d^L, with x, y substituted with a and d_j respectively. This minterm cannot imply any minterm $c_k(a, c_j)$ in $d^{LC}(a, D)$ (otherwise we have a homomorphism $c_k \rightarrow s_i$ contradicting the fact that d is

[8]The query d^{LC} is not necessarily empty. For example in $U(x, y_1), S_1(x, y_1), U(x, y_2), S_2(x, y_2), U(x, y_3), S_3(x, y_3) \vee U(x, y), S_1(x, y), S_2(x, y) \vee S_1(x, y), S_2(x, y), S_3(x, y), C(x, y) \vee \ldots$ where C is a central symbol, we have $d^{LC} = U(x, y), S_1(x, y), S_2(x, y)$.

minimized), hence it must imply some minterm in $d^L_{u_2}(a, D)$. This proves that $d^L_{u_1}(a, D) \Rightarrow d^L_{u_2}(a, D)$. To prove that the implication $d^L_{u_1}(a) \Rightarrow d^L_{u_2}(a)$ holds between queries, we use minterms in $d^L_{u_1}(a, D)$ that correspond to injective mappings of the y-variables to the set D; we leave the details to the reader. □

We need to prove something stronger: that the Boolean expressions $Y^{C_{uv}}_{uv}$ are inequivalent, where C_{uv} is the connected component that contains the distinguished tuple t_0. We start with a technical lemma:

LEMMA 8.28. *If $c_i[\theta_1, \theta_2]$ belongs to one of $d^R(\{a\} \cup C, D)$ or $d^C(C, D)$ or $d^L(C, D \cup \{b\})$, and θ_1, θ_2 are injective functions, then $c_i[\theta_1, \theta_2]$ is a minterm, i.e. it is not redundant.*

PROOF. Suppose all tuples of some other minterm $c_j[\theta'_1, \theta'_2]$ are contained in $c_i[\theta_1, \theta_2]$ (making the latter redundant). Then $c_j[\theta'_1, \theta'_2]$ cannot belong to $d^L_u(a, D)$, because all tuples in the latter are of the form $S(a, d_i)$ where S is a left symbol, and $c_i[\theta_1, \theta_2]$ does not contain any such tuples. Similarly, it cannot belong to $d^R_v(C, b)$. This means that c_j is a component of d, and we obtain a homomorphism $c_j \to c_i$ (since θ_1, θ_2 are injective) contradicting the fact that d was minimized. □

Next, we show:

LEMMA 8.29. *C_{uv} contains all the following tuples:*

—*$R(c)$ (if d^L is of Type 1).*
—*$S(c_i, b)$ and $S(c_i, d_j)$, for every left symbol S, and forall $c_i \in C$, $d_j \in D$.*
—*$S(c_i, d_j)$, for every central symbol S, and forall $c_i \in C$, $d_j \in D$.*
—*$S(a, d_j)$ and $S(c_i, d_j)$, for every right symbol S, and forall $c_i \in C$, $d_j \in D$.*
—*$T(c)$ (if d^R is of Type 2)*

PROOF. Let $c_0, c_1, \ldots, c_k$ be a strict path in the co-occurrence graph of d. That is, c_0 is a left component, c_k is a right component, all the others are central components. We have seen Prop. 8.11 that all relational symbols appear on this path. In particular, $c_1, \ldots, c_{k-1}$ contain all central symbols: call it the "central part" of the path. Recall that the distinguished tuple is $t_0 = S(c_i, d_j)$, for two constants c_i, d_j. Instantiate the central part of the path by setting $\theta_1(x) = c_i$, and $\theta_2(y) = d_j$; abbreviated $x = c_i$ and $y = d_j$. We obtain that all tuples of the form $S(c_i, d_j)$ where S is any central symbol belong to C_{uv}. Now consider the left component c_0, and let S be a left binary symbol common to both c_0 and c_1; clearly $S(c_i, d_j)$ belongs to C_{uv}, because this tuple occurs in the instantiation $c_1[c_i/x, d_j/y]$. We consider two cases, depending on the type of the query d.

Case 1 d is of Type 1-1. Then c_0 has only two variables x, y. In this case there is a single constant c_i and a single constant d_j, we denote them c, d, and consider the instantiation $c_0[c/x, d/y]$: it contains the tuple $S(c, d)$, hence all its tuples belong to C_{uv}. This proves that for all left symbols S, the tuple $S(c, d)$ belongs to C_{uv}. Also, $R(c)$ is in C_{uv}, where R is the unique left unary symbol. Finally, for every left symbol S, the tuple $S(c, b)$ belongs to some minterm of the form $c'[c/x, b/y]$, where c' is a left component. Since the latter contains the tuple $R(c)$, all its tuples, including $S(c, b)$, belong to C_{uv}. Similarly we prove that for all right symbols S, the tuples $S(c, d)$, $T(d)$, and $S(a, d)$ belong to C_{uv}.

Case 2 d is of Type 2-2. Consider an instantiation $c_0[\theta_1, \theta_2]$ of c_0 s.t. θ_1 maps x to c_i, θ_2 is injective and maps some atom with the symbol S to $S(c_i, d_j)$. By Lemma 8.28 we know that $c_0[\theta_1, \theta_2]$ is a minterm, and therefore all its tuples belong to C_{uv}. So far θ_2 mapped the y-variables in c_0 to the constants $d_1, \ldots, d_{n_y}$. Next, we consider all substitutions θ_2' that map the y-variables to $b, d_1, \ldots, d_{n_y}$. Since we have one more constants than variables, by changing the mapping for only one y-variable at a time, we can construct a sequence of θ_2' and prove that for all left binary symbols S occurring in c_0, all tuples of the form $S(c_i, d_p)$ and $S(c_i, b)$ belong to the connected component C_{uv}, for every $p = 1, \ldots, n_y$. For a fixed p, we can instantiate again the central part of the strict path, to $x = c_i$, $y = d_p$, and obtain that for all central symbols S, the tuples of the form $S(c_i, d_p)$ belong to C_{uv}. Finally, we repeat the same argument at the right end, and conclude that for every right symbol S, all tuples $S(c_q, d_p)$ and $S(a, d_p)$ belong to C_{uv}, and for every central symbol S the tuples $S(c_q, d_p)$ belong to C_{uv}. □

We now prove our claim, that all Boolean expressions $Y_{uv}^{C_{uv}}$ are in-equivalent. We do this separately for queries of type 1-1 and of type 2-2. We start with the latter.

PROPOSITION 8.30. *If the query d is of type 2-2, then for every u, v, the co-occurrence graph of Y_{uv} is connected.*

The proposition implies that $Y_{uv} = Y_{uv}^{C_{uv}}$, and our claim follows from Prop. 8.27.

PROOF. We need to prove that all symbols occurring in $d_u^L(a, D) \vee d^C(a, D)$ belong to the connected component C_{uv}, and similarly for $d^C(C, b) \vee d_v^R(C, b)$. Consider any strict path $c_0, c_1, \ldots, c_{k-1}, c_k$ in d; its central part, $c_1, \ldots, c_{k-1}$, contains all central symbols. Instantiate the central path to $x = a$, $y = d_j$, for some fixed constant d_j. First, we need to show that all resulting sets of tuples (one for each of $c_1, c_2, \ldots, c_{k-1}$) is a minterms in Y_{uv}. Suppose otherwise, that $c_i[a/x, d_j/y]$ is not a minterm, i.e. it is implied by another minterm. The latter can only come from $d_u^L(a, D)$ (otherwise, if it is of the form $c'[\theta_1', \theta_2']$ where c' is a component in d, then we obtain a homomorphism $c_i \to c'$ contradicting the fact that d was minimized), but each such minterm corresponds to an X-subcomponent in d^L, which has a ubiquitous left binary symbol U (Prop. 8.13), which (by definition) does not occur in c_1, and does not occur in any of $c_2, \ldots, c_{k-1}$ either because the latter do not contain any left symbols. Hence, all sets of tuples obtained from by instantiating the central path are minterms (i.e. non-redundant) and, therefore, all their tuples belong the connected component C_{uv}. Any central symbol S occurs in the central part of the path, hence $S(a, d_j)$ belongs to C_{uv}. It remains to show that for a left symbol S, the tuple $S(a, d_j)$ is in C_{uv}. There are two cases. Either S occurs in c_1, in which case it follows immediately that $S(a, d_j)$ belongs to C_{uv}. Or it does not occur in c_1: in that case it is a ubiquitous symbol, and we denote it U. Then we trace U in $d_u^L(a, D)$. Recall that every component of $d_u^L(a)$ corresponds to an X-subcomponent of d^L of some left component c. By Corollary 8.14, $c, c_1, \ldots, c_k$ is also a strict path. Consider the subcomponent s of c that occurs in $d_u^L(a)$: in other words, $d_u^L(a) = s[a/x] \vee \ldots$ Clearly, s contains the symbol U, because U is ubiquitous. We claim that it contains at least one left binary symbol S in common with c_1; otherwise, s consists only of ubiquitous symbols, and, if s' is any other

subcomponent of c, then there exists a homomorphism $s \to s'$ (since s' contains all ubiquitous symbols by definition), contradicting the fact that c is minimized. Thus, there exists a left binary symbol S that is common to s and c_1. Instantiate s by setting $x = a$ and $y = d_j$. This is a minterm $s[a, d_j]$ in Y_{uv}, containing both tuples $U(a, d_j)$ and $S(a, d_j)$: since the latter belongs to the connected component C_{uv}, so does the former. This completes the proof that Y_{uv} is connected. □

Next, we prove the claim for queries of type 1-1.

PROPOSITION 8.31. *If d is of type 1-1, then all four Boolean expressions $Y_{uv}^{C_{uv}}$, $u \in [2], v \in [2]$ are in-equivalent.*

PROOF. Recall that $u = 1$ means `false` and $u = 2$ means `true`. Furthermore, $d_1^L(a) \equiv$ `false` and $d_1^R(b) \equiv$ `false`. This implies that the co-occurrence graph of Y_{11} is connected: indeed, all sets of tuples in $d^C(a, c)$ are minterms (i.e. they are not redundant), and for any strict path $c_0, c_1, \ldots, c_k$, the component c_1 contains all left symbols (Prop. 8.13) and therefore, all tuples occurring in $d^C(a, c)$ belong to the same large connected component C_{uv}; similarly for $d^C(d, b)$. However, when $u = 2$, then in the expression $d_u^L(a, c) \vee d^C(a, c)$ some of the components in d^C may become redundant, and as a consequence some tuples of the form $S(a, c)$ for a left symbol S may become disconnected from the large connected component C_{uv}. Let C_u^L denote the set of tuples that become disconnected (this set may consists of several connected components), and similarly C_v^R the set of tuples of the form $S(c, b)$ where S is a right binary symbol, that become disconnected from the big component C_{uv}. We have $C_1^L = C_1^R = \emptyset$, while C_2^L and C_2^R may be empty or not. Thus:

$$Y_{uv} = Y_{uv}^{C_u^L} \vee Y_{uv}^{C_{uv}} \vee Y_{uv}^{C_v^R}$$

where $Y_{uv}^{C_u^L}$, $Y_{uv}^{C_{uv}}$, and $Y_{uv}^{C_v^R}$ do not share any common tuples (Boolean variables), and $Y_{1v}^{C_1^L} \equiv$ `false`, $Y_{u1}^{C_1^R} \equiv$ `false`.

To prove our claim, suppose the contrary, that $Y_{u_1 v_1}^{C_{u_1 v_1}} \equiv Y_{u_2 v_2}^{C_{u_2 v_1}}$. The pair of labels u_1, v_1 is distinct from u_2, v_2, so assume w.l.o.g. that $u_1 = 1$ and $u_2 = 2$. There are two cases. First, $C_2^L \neq \emptyset$. Then $Y_{1v_1}^{C_{1v_1}}$ depends on all Boolean variables in C_2^L, while $Y_{2v_2}^{C_{2v_2}}$ does not depend on these variables, hence they cannot be equivalent. Second, $C_2^L = \emptyset$. Then

$$Y_{1v_1} = Y_{1v_1}^{C_{1v_1}} \vee Y_{1v_1}^{C_{v_1}^R}$$

$$Y_{2v_2} = Y_{2v_2}^{C_{2v_2}} \vee Y_{2v_2}^{C_{v_2}^R}$$

Since $Y_{1v_1} \not\equiv Y_{2v_2}$, we must have $v_1 \neq v_2$, so assume w.l.o.g. that $v_1 = 1$ and $v_2 = 2$. Thus, $Y_{1v_1}^{C_{v_1}^R} \equiv$ `false`. Here there are also two cases. Either $C_2^R \neq \emptyset$: in this case $Y_{11}^{C_{11}}$ depends on strictly more Boolean variables than $Y_{22}^{C_{22}}$. Or $C_2^R = \emptyset$. In that case we obtain that $Y_{11} \equiv Y_{22}$, which is a contradiction. □

8.7 Solving for $det(M) \neq 0$

Finally, we show here that, if the Jacobian of $\bar{x} \mapsto \bar{y}$ is non-zero, then M is non-singular and, moreover, once can choose in polynomial time values for $\bar{x}$ s.t. the re-

sulting matrix M is non-singular. For the first part we apply the Vandermonde/Kronecker-product principle that we alluded to. For the second part, we note that the matrix M consists of powers of the polynomials y_{uv} (see Eq. 18 and Eq. 21), and therefore, its determinant is a polynomial too. Given that this polynomial is not identically zero (which follows from the first part), we need to find in polynomial time some values for its variables such that the polynomial at those values is not zero. Here, we apply a simple principle. Given a polynomial in a single variable x, $f(x) = \sum_k a_k x^k$, one can find in PTIME in n a value v s.t. $f(v) \neq 0$: indeed, simply choose $n+1$ distinct real values $v_0, v_1, \ldots, v_n$ and compute $f(v_j)$ for $j = 0, \ldots, n$. At least one $\neq 0$, otherwise the polynomial is identical zero.

PROPOSITION 8.32. *Let $\bar{x} = (x_1, \ldots, x_m)$ be m variables, and $\bar{G}(\bar{x}) = (G_1(\bar{x}), \ldots, G_n(\bar{x}))$ be n multivariate polynomials in $\bar{x}$. Consider n distinct copies of $\bar{x}$, denoted $\bar{x}_i$, $i = 1, n$, and let $\bar{X} = (\bar{x}_i)_{i=1,n}$ be the set of $m \cdot n$ distinct variables. Define the matrix of polynomials:*

$$M(\bar{G}) \;=\; (G_j(\bar{x}_i))_{i,j=1,n} \tag{23}$$

Then there exists an algorithm that runs in time $(n+1)^{O(m)}$ and either determines that $det(M) \equiv 0$ (as a multivariate polynomial in $\bar{X}$) or finds values $\bar{V}$ s.t. $det(M(\bar{G})[\bar{V}/\bar{X}]) \neq 0$.

The second proposition is:

PROPOSITION 8.33. *Let $\bar{F}(\bar{x}) = (F_1(\bar{x}), \ldots, F_m(\bar{x}))$ be m multivariate polynomials, each in m variables $\bar{x}$. For each $n > 0$, define $(n+1)^m$ polynomials $\bar{G}(\bar{x})$ as follows:*

$$\bar{G} \;=\; (\prod_{i=1,m} F_i^{j_i}(\bar{x}))_{0 \leq j_1, \ldots, j_m \leq n} \tag{24}$$

Thus, for each m-tuple $(j_1, j_2, \ldots, j_m) \in \{0, \ldots, n\}^m$ there is one polynomial in $\bar{G}$. Suppose that the Jacobian of the function $\bar{x} \to \bar{F}(\bar{x})$ is not identically zero. Then $det(M(\bar{G})) \not\equiv 0$, where $M(\bar{G})$ is the matrix of polynomials defined by Eq. 23.

Together, these two propositions prove the following. Fix $m = O(1)$, and suppose we are given m polynomials $F_1(\bar{x}), \ldots, F_m(\bar{x})$ in m variables $\bar{x}$ with a non-zero Jacobian. Define $\bar{G}$ as in Eq. 24 and $M = M(G)$ as in Eq. 23. Then the determinant of polynomials $det(M)$ is not zero, and one can find in PTIME $(n+1)^m$ values $\bar{V}$ for $\bar{x}$ s.t.the determinant of numbers $det(M)[\bar{V}] \neq 0$.

In the remainder of this section we prove the two propositions.

To prove Prop. 8.32 we need the following two lemmas.

LEMMA 8.34. *Let $P(x_1, \ldots, x_m)$ be a multivariate polynomial of degree n, with $m = O(1)$ variables. Suppose we have an Oracle that, given values $v_1, \ldots, v_m$, computes $P(v_1, \ldots, v_m)$ in time T. Then there exists an algorithm that runs in time $O((n+1)^m T)$ and either determines that $P \equiv 0$ or returns a set of values $\bar{v} = (v_1, \ldots, v_m)$ s.t. $P(\bar{v}) \neq 0$.*

PROOF. By induction on m. Choose $n+1$ distinct values for the last variable: $x_m = v_m^0$, $x_m = v_m^1$, …, $x_m = v_m^n$. For each value v_m^i we substitute $x_m = v_m^i$ in P. We obtain a new polynomial, $Q = P[x_m/v_m^i]$, with $m-1$ variables. By

induction, we can find in time $O((n+1)^{m-1}T)$ a set of values $v_1, \ldots, v_{m-1}$ s.t. $Q[x_1/v_1, \ldots, x_{m-1}/v_{m-1}] \neq 0$, or determine that none exists. If for some i we find such values, then augment them with v_m^i and return $(v_1, \ldots, v_{m-1}, v_m^i)$. If $Q = P[v_m^i/x_m] \equiv 0$ for all $i = 0, n$, then P is divisible by $\prod_i (x_m - v_m^i)$, by Hilbert's Nullstellensatz. Since the degree of x_m in P is at most n, it follows $P \equiv 0$. □

Recall that n multivariate polynomials $G_1, \ldots, G_n$ are called *linearly independent* if, for any constants $c_1, \ldots, c_n$, if $c_1 G_1 + \ldots + c_n G_n \equiv 0$ then $c_1 = \ldots = c_n = 0$.

LEMMA 8.35. *The polynomials $G_1, \ldots, G_n$ are linearly dependent iff $det(M(\bar{G})) \equiv 0$, where M is given by Eq. 23.*

PROOF. The "if" direction follows immediately from the fact that the columns in M are linearly dependent. For the "only if" direction, we prove by induction on n that, if n polynomials $G_1, \ldots, G_n$ are linearly independent, then $det(M) \not\equiv 0$. Let M' be the $(n-1) \times (n-1)$ upper-left minor of M. Since $G_1, \ldots, G_{n-1}$ are also linearly independent, $det(M') \not\equiv 0$. Hence, there exists values $\bar{V}' = (\bar{v}_1, \ldots, \bar{v}_{n-1})$ s.t. $det(M')[\bar{V}'] \not\equiv 0$. Consider now $det(M)[\bar{V}']$: that is, we substitute $\bar{x}_1, \ldots, \bar{x}_{n-1}$ with the values $\bar{V}'$, and keep only the variables $\bar{x}_N$. Then $det(M(\bar{V}'))$ has in the last row the polynomials $G_1(\bar{x}_n), \ldots, G_n(\bar{x}_n)$, and has constants in all other rows. Its value is a linear combination of the polynomials in the last row: $det(M) = c_1 \cdot G_1 + \ldots + c_n \cdot G_n$. The last coefficient, $c_n = det(M')[\bar{V}']$, is non-zero, hence $det(M) \not\equiv 0$, because the polynomials are linearly independent. □

We can now prove Prop. 8.32.

PROOF. (of Prop. 8.32) We proceed by induction on n. Consider the upper left minor M' of dimensions $(n-1) \times (n-1)$. Apply induction to the matrix M'. If we determine that $det(M') \equiv 0$, then this implies that $G_1, \ldots, G_{n-1}$ are linearly dependent, and therefore so are $G_1, \ldots, G_n$, which implies $det(M) = 0$. Otherwise, we have computed a set of values $\bar{V}' = (\bar{v}_1, \ldots, \bar{v}_{n-1})$ that make the upper left minor M' non-singular. Then $det(M)$ is a polynomial $P(\bar{x}_n)$ in $m = O(1)$ variables $\bar{x}_n$. We also have an oracle for computing P in time $O(n^3)$: given values $\bar{v}$, substitute them in last row of M, then compute $det(M)$ using Gaussian elimination. Apply Lemma 8.34 to compute in time $O((n+1)^{m+3})$ a set of values $\bar{v}$ for which this polynomial is non-zero. Return the vector consisting of $\bar{V}'$ concatenated with $\bar{v}$. The total running time is $O(n^{O(m)}$ for the first $n-1$ steps (induction) on M', plus $O((n+1)^{m+3})$ for the n'th step. □

Next, we prove Prop. 8.33. Recall that a *Vandermonde matrix* of dimensions $n \times n$ is:

$$V(y_1, \ldots, y_n) = \left(y_i^{j-1}\right)_{i,j=1,n}$$

The *Kronecker product* of two matrices $A = (a_{ij})$, $B = (b_{kl})$ of dimensions $n_1 \times n_1$ and $n_2 \times n_2$ respecitvely, is the following matrix of dimensions $n_1 n_2 \times n_1 n_2$:

$$A \otimes B = (a_{ij} \cdot b_{kl})_{(i,k) \in [n_1] \times [n_2]; (j,l) \in [n_1] \times [n_2]}$$

It is known that $det(A) \neq 0$ and $det(B) \neq 0$ implies $det(A \otimes B) \neq 0$.

PROOF. (of Prop. 8.33) Note that here the matrix $M(\bar{G})$ has dimension $(1 + n)^m \times (1+n)^m$. Consider row i of the matrix: its entries are products of the form $F_1^{j_1} \cdots F_m^{j_m}$ for all possible values $j_1, \ldots, j_m \in \{0, \ldots, n\}$, and all over the same set of variables $\bar{x}_i$.

Since the Jacobian of $\bar{F}(\bar{x})$ is non-zero, the image of $\bar{F}$ includes a closed, m-dimensional cube $C = \prod_{i=1,m}[a_i, b_i]$. For each $i = 1, m$ choose a set $Y_i = \{y_{i,0}, y_{i,1}, \ldots, y_{i,n}\}$ s.t. $a_i < y_{i,0} < y_{i,1} < y_{i,2} < \ldots < y_{i,n} < b_i$. Then $\bar{Y} = Y_1 \times \ldots \times Y_m$ has $(n+1)^m$ points and $\bar{Y} \subseteq C$. Note that $\bar{Y}$ is in the image of $\bar{F}$, hence for every $\bar{y} \in \bar{Y}$ there exists values $\bar{x}_i$ s.t. $\bar{F}(\bar{x}) = \bar{y}$. By choosing these $(n+1)^m$ values for $\bar{x}_i$ in the definition of $M(\bar{G})$ in Eq. 24 we obtain the following matrix:

$$\begin{aligned} M(\bar{G}) &= \left(y_1^{j_1} \cdots y_m^{j_m}\right)_{(y_1,\ldots,y_m)\in\bar{Y},(j_1,\ldots,j_m)\in\{0,\ldots,n\}^m} \\ &= V(y_{1,0}, y_{1,1}, \ldots, y_{1,n}) \otimes \cdots \otimes V(y_{m,0}, y_{1,1}, \ldots, y_{m,n}) \end{aligned}$$

Each Vandermonde matrix is non-singular, implying that M is non-singular. □

8.8 Summary of the Proof of the Main Theorem

We summarize now proof of Theorem 8.1. Let d be a forbidden query. Given an oracle for computing $P(d)$, we show how to use it to solve an SC-problem (of the same type as the query). Let the SC problem be given by the bipartite graph $E \subseteq [n_1] \times [n_2]$. Start by constructing a generic sub-block $D_1(a, b)$ as a zig-zag-zig block. Choose any zag-tuple $t = S(c_i, d_j)$ arbitrarily and designate it as distinguished tuple. Its negated probability is a variable x. Each other tuple will be a separate variable. For each $u \in [m_1], v \in [m_2]$, consider the multi-linear polynomial $f_{uv} = P_{D_1(a,b)}(\neg d_{uv}(a, b))$. We have shown in Section 8.6.5 that x is a distinguished variable for this set of polynomials. Choose values of all variables other than x as per Prop. 8.25: now the polynomials f_{uv} are linear (they only depend on x), non-degenerate, and non-equivalent. Next, make copies $D_1(a, b), \ldots, D_{m_1 m_2}(a, b)$ of this block, keeping all probabilities equal (they are constants), except for the distinguished tuples, where the probabilities will be given by distinct variables $x_1, \ldots, x_{m_1 m_2}$. This completes the construction of the block $D(a, b)$. If the type of the query is 2-1 or 2-2, then also construct a generic block $D(a, \cdot)$ with m_1 distinct tuples whose probabilities are given by m_1 variables; if its type is 1-2 or 2-2 then also construct a block $D(\cdot, b)$ with m_2 distinct tuples, depending on m_2 variables. The entire database D is the union of blocks $D(a, b)$, $D(a, \cdot)$, $D(\cdot, b)$, for $(a, b) \in E$: it is a probabilistic database that depends on m variables[9] $\bar{x} = (x_1, \ldots, x_m)$. Choose $(n+1)^m$ values for $\bar{x}$ as explained Sect. 8.7 (essentially by trial and error) to ensure that the matrix M is non-singular. Call the oracle for $P(d)$ repeatedly, $(n+1)^m$ times. $P(d)$ is given by Eq. 18 or Eq. 21, and the matrix of the corresponding system of linear equations is M, which is non-singular. Solve the system using Gaussian elimination, and obtain the solution to the SC-problem. This proves that computing $P(d)$ is hard for $FP^{\#P}$

[9] m is either $m_1 m_2$ or $m_1 m_2 + m_1$ or $m_1 m_2 + m_2$ or $m_1 m_2 + m_1 + m_2$, depending on the type of the query d.

9. CONCLUSIONS

In this paper we have studied a fundamental computational problem connecting logic an probability theory: given a query and a probabilistic database, compute the probability of the query on that database. We have established a dichotomy for unions of conjunctive queries (also known as the positive, existential fragment of First Order logic): for every query Q, either $P(Q)$ can be computed in PTIME in the size of the database, or it hard for $FP^{\#P}$. We call the query safe in the first case, and unsafe in the second case. Safety/unsafety can be decided solely on the syntax of the query, where "syntax" includes, unexpectedly, the Mobius function of a certain lattice derived from the query. For safe queries we have given a simple, yet quite non-obvious algorithm, which alternates between an inclusion/exclusion step over the CNF representation of the query, and an elimination step of one existential variable. For the hardness proof, we have developed two distinct sets of techniques. One simplifies any unsafe query until it has only two types of attributes: this is called a *forbidden query*. The other set of techniques prove the hardness of every forbidden query directly, using a reduction from Provan and Ball's *Positive Partitioned 2 CNF* counting problem. The reduction is highly non-trivial, and involves arguments about irreducible multivariate polynomials.

As future work, we identify the following open problems. The first is a sharpening of the hardness result, to replace our current oracle construction with a parsimonious reduction. Second, an extension to full First Order Logic. Third, extensions to richer data models, such as disjoint-independent probabilistic databases, or functional dependencies.

Acknowledgments We thank Christoph Koch and Paul Beame for pointing us (independently) to incidence algebras. This work was partially supported by NSF IIS-0713576.

REFERENCES

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases.* Addison Wesley Publishing Co.

Chandra, A. and Merlin, P. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of 9th ACM Symposium on Thoery of Computing.* Boulder, Colorado, 77–90.

Creignou, N. and Hermann, M. 1996. Complexity of generalized satisfiability counting problems. *Inf. Comput 125,* 1, 1–12.

Dalvi, N., Schnaitter, K., and Suciu, D. 2010a. Computing query probability with incidence algebras. In *PODS*. 203–214.

Dalvi, N., Schnaitter, K., and Suciu, D. 2010b. Computing query probability with incidence algebras. Tehnical Report UW-CSE-10-03-02, University of Washington. March.

Dalvi, N. and Suciu, D. 2004. Efficient query evaluation on probabilistic databases. In *VLDB*. Toronto, Canada.

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

Dalvi, N. and Suciu, D. 2007b. Management of probabilistic data: Foundations and challenges. In *PODS*. Beijing, China, 1–12. (invited talk).

Darwiche, A. 2000. On the tractable counting of theory models and its application to belief revision and truth maintenance. *CoRR cs.AI/0003044*.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *J. Artif. Int. Res. 17,* 1, 229–264.

Golumbic, M. C., Mintz, A., and Rotics, U. 2006. Factoring and recognition of read-once functions using cographs and normality and the readability of functions associated with partial k-trees. *Discrete Applied Mathematics 154,* 10, 1465–1477.

Grädel, E., Gurevich, Y., and Hirsch, C. 1998. The complexity of query reliability. In *PODS*. 227–234.

Green, T., Karvounarakis, G., and Tannen, V. 2007. Provenance semirings. In *PODS*. 31–40.

Jha, A. and Suciu, D. 2010. Knowledge compilation meets database theory : Compiling queries to decision diagrams. (under review).

Knuth, K. H. 2005. Lattice duality: The origin of probability and entropy. *Neurocomputing 67*, 245–274.

Krattenthaler, C. 1999. Advanced determinant calculus. *Seminaire Lotharingien Combin 42 (The Andrews Festschrift)*, 1–66. Article B42q.

Olteanu, D. and Huang, J. 2009. Secondary-storage confidence computation for conjunctive queries with inequalities. In *SIGMOD*. 389–402.

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

Provan, J. S. and Ball, M. O. 1983. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput. 12,* 4, 777–788.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning 62,* 1-2, 107–136.

Sagiv, Y. and Yannakakis, M. 1980. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM 27*, 633–655.

Stanley, R. P. 1997. *Enumerative Combinatorics.* Cambridge University Press.

Valiant, L. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput. 8*, 410–421.

Wegener, I. 2000. *Branching programs and binary decision diagrams: theory and applications.* SIAM.

Wegener, I. 2004. BDDs–design, analysis, complexity, and applications. *Discrete Applied Mathematics 138,* 1-2, 229–251.