Evaluation of relational algebra queries on probabilistic databases: Tractability and approximation

Robert Fink
St Cross College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity 2013

Abstract

Query processing is a core task in probabilistic databases: Given a query and a database that encodes uncertainty in data by means of probability distributions, the problem is to compute possible query answers together with their respective probabilities of being correct. This thesis advances the state of the art in two aspects of query processing in probabilistic databases: complexity analysis and query evaluation techniques. A dichotomy is established for non-repeating, conjunctive relational algebra queries with negation that separates #P-hard queries from those with PTIME data complexity. A framework for computing probabilities of relational algebra queries is presented; the probability computation algorithm is based on decomposition methods and provides exact answers in the case of exhaustive decompositions, or anytime approximate answers with absolute or relative error guarantees in the case of partial decompositions. The framework is extended to queries with aggregation operators. An experimental evaluation of the proposed algorithms' implementations within the SPROUT query engine complements the theoretical results. The SPROUT^2 system uses this query engine to compute answers to queries on uncertain, tabular Web data.

Acknowledgements

First of all, I owe a heartfelt thank-you to my supervisor, Dan Olteanu, who has not only provided more support than any well-supervised student could possibly hope for, but who also easily proved to be a reliable daily lunch-buddy and friend.

I would further like to acknowledge my examiners, Michael Benedikt and Graham Cormode, for their very detailed evaluation of this thesis and for their valuable feedback. Thank you to my collaborators and friends who have helped shape ideas, algorithms, implementations, and grammar, or convinced me that mine were wrong: Serge Abiteboul, Jamie Berezin, Larisa Han, Jiewen Huang, Swaroop Rath, Cristian Riveros, and Pierre Senellart.

Last but not least, thank you, Jakub Závodný, for making my days so much more enjoyable (mrkvička ...) and my proofs so much more correct. Thank you to all my wonderful friends that I have made over the last years; you were most certainly the coolest and best thing about Oxford. Caitlin, you are the sexiest. And thank you for continually providing me with some of your excess energy.

My research was funded by the ERC FP7 grant agreement FOX number FP7-ICT-233599.

Statement of Originality

The material of Chapter 3 is Robert Fink's original work and is so far unpublished. A variation of the dichotomy result was first announced by Robert Fink, Dan Olteanu, and Swaroop Rath [34].

The material from Section 4.1 has been published by Robert Fink and Dan Olteanu [33]. The idea underlying the decomposition technique presented in Section 4.2 and the experiments of Section 4.3 have previously been published by Dan Olteanu, Jiewen Huang, and Christoph Koch [63]. The ideas of the two papers were combined and published by Robert Fink, Dan Olteanu, and Jiewen Huang [32]; Chapter 4 is based on the latter publication.

Chapter 5 is based on joint work of Robert Fink, Larisa Han, and Dan Olteanu [30].

SPROUT^2 has been developed by Robert Fink, Andrew Hogue, Dan Olteanu and Swaroop Rath [31].

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction and overview

Standard relational databases contain certain data: Every tuple in the database is assumed to be correct. Probabilistic databases generalise deterministic databases by allowing data to be uncertain: Tuples may be in the database only with a given probability, and data values can be specified by means of probability distributions over possible values. Prime sources of uncertain information are information extraction systems such as Google Squared [18]; a Square is the answer to a user-supplied keyword query and comprises two types of columns. Firstly, a key column holding entities that match the query, and secondly a number of attribute columns holding values that describe various properties of these entities. As a result of the information extraction process, properties may take multiple alternative values, and each possible value is annotated with a likelihood measure such as "Low confidence" or "High confidence". The confidence annotation of a value may reflect the trustworthiness of the Web page from which is was learned, as well as the confidence of the extraction algorithm that it is a correct value for the respective property.

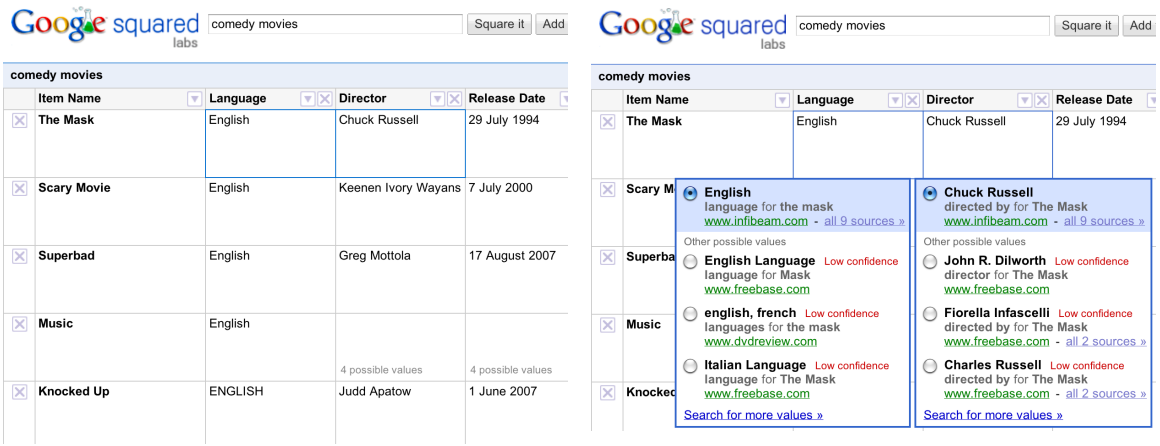


Figure 1.1: Google Square for the query *comedy movies*; the right screenshot highlights the different possible values for the properties *Language* and *Director* of the film *The Mask*. The values are annotated with a confidence label that lends itself to a probabilistic interpretation for the likelihood of each value being correct.

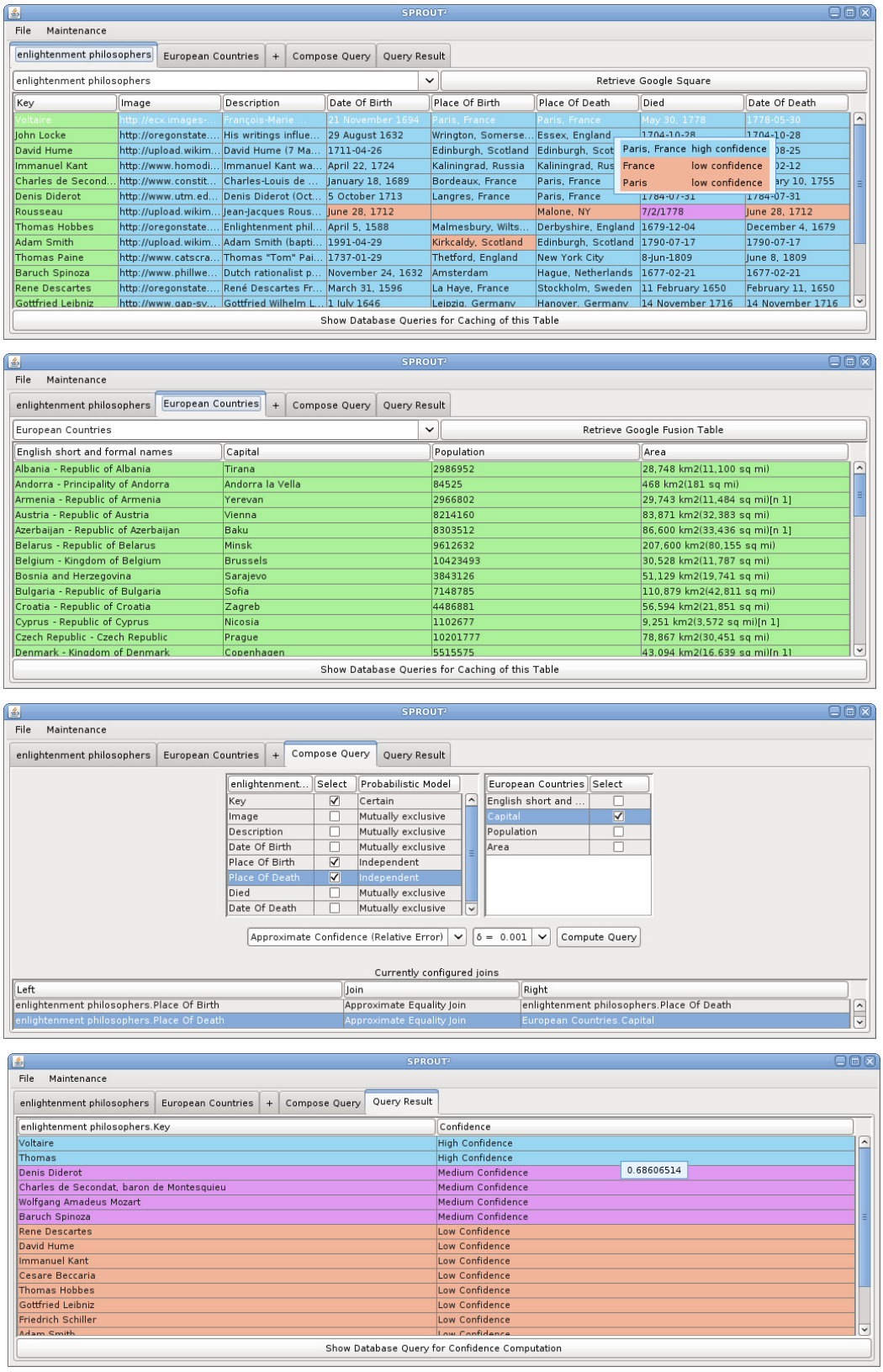


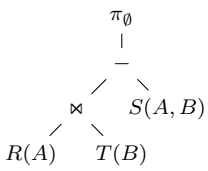
Figure 1.2: Screenshots of a SPROUT^2 query asking for philosophers that were born and died in the same European capital city; the top two screenshots depict the input tables, the third and fourth screenshots show the query and its results, respectively.

Example 1. Figure 1.1 shows the Square for the query *comedy movies*. The left screenshot shows a list of movie titles along with their languages, directors, and release dates. By clicking on a cell (cf. right screenshot), a user can reveal alternative values for a property, each annotated with a label indicating its likelihood according to the employed information extraction algorithms. For example, the most likely director of *The Mask* is *Chuck Russell* and less likely alternatives include *John R. Dilworth*, *Fiorella Infascelli*, and *Charles Russell*. Uncertainty in the data — as represented by the confidence annotations of different possible values — can be modelled as a probabilistic database which assigns higher probabilities to values annotated with "High confidence" than to values with "Low confidence". Moreover, such probability distributions can capture correlations in the data. For example, alternative values for each property may be considered mutually exclusive, or the probability of values may be positively correlated if they were extracted from the same Web page. □

A core task in a probabilistic database is query evaluation: Given a database and a query, return each possible answer to the query along with the probability that it is correct. Queries shield users from the complications of the underlying probabilistic semantics of the data. They issue standard, deterministic queries (for example in SQL), and it is the task of the probabilistic database management system to compute the probabilities for query answers with respect to the uncertainty of the data as encoded in the database. A ranking of the possible answers by their probability suggests to the users the most likely answers to their query. Let us illustrate such probabilistic queries with a query in the SPROUT^2 ("SPROUT Squared") system [31]. SPROUT^2 allows a users to import uncertain tabular Web data (such as Google Squares or NELL uncertain relations [16]) and provides a graphical interface that assists them in authoring SQL-like queries on the imported data. The thus composed queries are then evaluated to yield a list of possible query answers ranked by their likelihood. SPROUT^2 demonstrates a core strength of probabilistic databases: disparate data from different sources can be consolidated into one coherent database by modelling the relationship of data items probabilistically, and queries against such data are oblivious to the heterogeneous nature of the underlying data.

Example 2. Figure 1.2 shows a SPROUT^2-query on the Google Square *enlightenment philosophers* and a table on *European countries* and their capitals (first and second screenshot). The query (third screenshot) asks for those philosophers who were born and died in the same European capital city by enforcing uncertain joins between *Place of Birth*, *Place of Death*, and *Capital*. The bottom screenshot shows the query results; for example, *Voltaire* is a very likely answer because the Google Square suggests with high confidence that France's capital Paris is both his place of birth and his place of death. □

This thesis addresses two fundamental computational aspects of relational algebra queries on probabilistic databases. Firstly, different classes of relational algebra queries are analysed with respect to their data complexity (i.e., queries are considered fixed); secondly, efficient algorithms are proposed for the evaluation of such queries.



R

A	$\mathbf{\Phi}$
1	$\mathbf{x_1}$
2	$\mathbf{x_2}$

T

B	$\mathbf{\Phi}$
1	$\mathbf{y_1}$
2	$\mathbf{y_2}$

S

A	B	$\mathbf{\Phi}$
1	1	$\bot$
1	2	$\bot$
2	1	$\bot$
2	2	$\top$

$R \bowtie T$

A	B	$\mathbf{\Phi}$
1	1	$\mathbf{x_1y_1}$
1	2	$\mathbf{x_1y_2}$
2	1	$\mathbf{x_2y_1}$
2	2	$\mathbf{x_2y_2}$

$(R \bowtie T) - S$

A	B	$\mathbf{\Phi}$
1	1	$\mathbf{x_1y_1}\neg\bot$
1	2	$\mathbf{x_1y_2}\neg\bot$
2	1	$\mathbf{x_2y_1}\neg\bot$
2	2	$\mathbf{x_2y_2}\neg\top$

$\pi_\emptyset[(R \bowtie T) - S]$

$\mathbf{\Phi}$
$\mathbf{x_1y_1} \vee \mathbf{x_1y_2} \vee \mathbf{x_2y_1}$

Figure 1.3: Query and database used in Example 3.

Complexity analysis of queries. Charting the tractability frontier of query evaluation lies at the foundation of probabilistic databases. The investigation of the data complexity of queries is in line with the corresponding analysis in standard deterministic databases in which, for example, *acyclic* queries fall into a lower complexity class than *cyclic* queries.

Arguably, understanding the data complexity in the probabilistic case is even more important than in the deterministic case, because the separation between "easy" and "hard" queries is vast: Some queries are *tractable* because they have polynomial-time data complexity, while others are considered *intractable* because they are provably hard for #P. Existing probabilistic database management systems, such as MystiQ, MayBMS and SPROUT, rely fundamentally on query tractability results as they provide exact evaluation techniques for tractable queries and approximate techniques for intractable queries.

Example 3. Let us discuss an example of an intractable query and sketch a hardness proof. Figure 1.3 depicts a relational algebra query Q and a probabilistic database with relations R, S, T. The query comprises a join ($\bowtie$) between relations R and T; in this particular query, the join degenerates to a Cartesian product between the relations because it does not enforce any join condition. Then, the query expresses the set difference ($-$) between the result of $R \bowtie T$ and the relation S. Finally, the query projects on the empty list of attributes ($\pi_\emptyset$) to yield a Boolean query answer.

The database is represented in the so-called *pc-table* formalism, in which each tuple is annotated with a propositional formula over a set of Boolean random variables and constants $\top$ (Boolean true) and $\bot$ (Boolean false); this annotation is depicted in Φ-column of each relation. An annotation captures the uncertainty regarding the correctness of a tuple: Intuitively, a tuple with annotation $\top$ is in the database with certainty, and a tuple with annotation $\bot$ is not in the database with certainty; a tuple with annotation x is in the database with a given probability, i.e., a number P_x with $0 \leq P_x \leq 1$, and is not in the database with probability $1 - P_x$. For example, in a Google Square the different possible values for a property and their respective confidence annotations can be modelled as a pc-table by adding a tuple for each of the values, and by annotating each tuple with a variable symbol whose probability reflects the confidence annotation. For instance, in the Square from Figure 1.1, we would add the tuples *Chuck Russell*, *John R. Dilworth*, *Fiorella Infascelli*, and *Charles Russell* and annotate them with variables v_2, v_2, v_3, v_4, respectively, and could define their probabilities by $P_{v_1} = 0.8$ — indicating that Chuck Russell is a

likely value — and $P_{v_2} = P_{v_3} = P_{v_4} = 0.3$ — indicating that Dilworth, Infascelli, and Charles Russell are less likely values. Note that these annotations model the four values as being independent; if we instead consider the values Chuck Russell and Dilworth mutually exclusive, we could introduce another variable v_0 and annotate the two tuples with $v_1 v_0$ and $v_2 \neg v_0$, respectively. There is no valuation of the variables that satisfies both formulas, and hence the two tuples are mutually exclusive.

Query evaluation can be understood as a two-step process. Firstly, tuple annotations are propagated by query operators to yield an annotation for each query answer (or just one annotation in case of a Boolean query). This formula traces the contribution of each input tuple to the query answer:

- The annotation of the result of a join is the conjunction of the annotations of the participating tuples. For example, tuple $(1, 2)$ in relation $R \bowtie T$ has annotation $x_1 y_2$ because tuples $R(1)$ and $T(2)$ have annotations x_1 and y_2, respectively.

- The difference operator yields the conjunction of the annotation of a "left" tuple with the negation of the annotation of a "right" tuple. For example, the annotation of tuple $(2, 1)$ in $(R \bowtie T) - S$ is $x_2 y_1 \neg \bot$ because the "left" tuple $(2, 1)$ in $R \bowtie T$ has annotation $x_2 y_1$ and the "right" tuple $(2, 1)$ in S has annotation $\bot$. Annotations may optionally be simplified according to the standard Boolean equivalences; this would yield the annotation $x_2 y_1$ for tuple $(2, 1)$ and annotation $\bot$ for tuple $(2, 2)$.

- The projection operator yields the disjunction of all annotations of tuples that are duplicates as a result of the projection. For example, the empty projection $\pi_\emptyset$ yields the disjunction $x_1 y_1 \vee x_1 y_2 \vee x_2 y_1$ of all annotations of $(R \bowtie T) - S$; note that the unsatisfiable clause $x_2 y_2 \neg \top \equiv x_2 y_2 \bot \equiv \bot$ is omitted.

The annotation $x_1 y_1 \vee x_1 y_2 \vee x_2 y_1$ of the query result (cf. right-most table) reads as follows. The query is true if at least one of the following combinations of tuples is in the database: $R(1)$ and $T(1)$, or $R(1)$ and $T(2)$, or $R(2)$ and $T(1)$. Note that S is a deterministic relation since its annotations are constants $\top$ and $\bot$. The combination of $R(2)$ and $T(2)$ can never contribute to satisfying the query because tuple $(2, 2)$ is annotated with $\top$ in relation S and is thus in the database for any valuation of the variables; in other words, tuple $(2, 2)$ in relation S always cancels tuple $(2, 2)$ in relation $R \bowtie T$. Consequently, the annotation of the query result does not contain the clause $x_2 y_2$.

Secondly, the probability of a query result is computed by evaluating the probability of its annotation formula Φ. Intuitively, this is the probability that Φ is satisfied by a randomly chosen valuation of the annotation variables of the input tuples. More formally, there is an intimate connection between the model counting and the probability computation problem: The former asks for the number $\#\Phi = \sum_{\nu \models \Phi} 1$ of total valuations ν of the variables that satisfy Φ, the latter asks for the sum $P_\Phi = \sum_{\nu \models \Phi} \Pr(\nu)$ of the probabilities of the satisfying valuations. It is easy to see that if each of the n variables is true with probability 0.5, then

all valuations are equally likely — their probability is $\Pr(\nu) = 2^{-n}$ — and the two quantities are related via

$$\#\Phi = 2^n P_\Phi \tag{1.1}$$

In the light of the above insights, it is straightforward to sketch a reduction from the #P-complete model-counting problem for the class 2DNF of positive, bipartite propositional formulas in disjunctive normal form (DNF) to the query evaluation problem of the above query Q. The underlying idea is to encode a given 2DNF formula Φ as a probabilistic database $\mathcal{D}$ such that the annotation of Q evaluated on $\mathcal{D}$ is exactly Φ.

The figure from the beginning of this example illustrates this reduction for the depicted query Q and for the problem instance $x_1y_1 \vee x_1y_2 \vee x_2y_1$: Indeed, the propagation of the annotations of the database yields the annotation $\Phi = x_1y_1 \vee x_1y_2 \vee x_2y_1$ for the Boolean query result. Thus, any algorithm that evaluates $P_{Q(\mathcal{D})} = P_\Phi$ also determines the number of models, i.e., the number of satisfying valuations, of $x_1y_1 \vee x_1y_2 \vee x_2y_1$ via Equation (1.1). We show in Chapter 3 that query Q can indeed encode every 2DNF formula. □

Exact and approximate query evaluation. Since queries may be known a-priori to be either tractable or intractable, it is sensible to study the query evaluation problem from two different perspectives: *Exact* evaluation techniques compute query probabilities efficiently for tractable queries, while *approximate* evaluation techniques may be applied to intractable queries and yield a probability interval that satisfies given error guarantees.

Example 4. The underlying idea of our algorithms is to recursively decompose a given annotation formula Φ into simpler sub-formulas in such way that the probability of Φ can be efficiently computed given the probabilities of the sub-formulas.

Assume that a query result has the annotation

$$\Phi = x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \vee x_3y_6z_4.$$

The exact probability P_Φ can be computed by repeatedly applying two decomposition steps:

1. Since Φ is the disjunction of the two independent sub-formulas

$$\Phi_1 = x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2 \qquad \Phi_2 = x_3y_5z_3 \vee x_3y_6z_4,$$

   its probability is equal to $1 - (1 - P_{\Phi_1})(1 - P_{\Phi_2})$.

2. Φ_1 cannot be decomposed into independent sub-formulas, but it may be further simplified by applying a Shannon expansion step. For instance, the expansion for variable x_1 is

$$\Phi_1 \equiv x_1 \wedge \Phi_1|_{x_1 \leftarrow \top} \quad \vee \quad \neg x_1 \wedge \Phi_1|_{x_1 \leftarrow \bot},$$

   where $\Phi_1|_{x_1 \leftarrow \top}$ is the formula obtained from Φ_1 by replacing every occurrence of

variable x_1 with the constant $\top$, and $\Phi_1|_{x_1 \leftarrow \bot}$ is defined symmetrically. Since the two sub-formulas obtained by the expansion are mutually exclusive, the probability of their disjunction is simply the sum of their probabilities:

$$P_{\Phi_1} = P_{x_1} \cdot P_{\Phi_1|_{x_1 \leftarrow \top}} \quad + \quad (1 - P_{x_1}) \cdot P_{\Phi_1|_{x_1 \leftarrow \bot}},$$

The recursive application of these two rules gives rise to a *decomposition tree* for Φ that reflects the decomposition steps taken and allows to compute P_Φ in one bottom-up pass. The base case of the recursive decomposition is a trivial formula with one literal whose probability is given.

Alternatively, one may be interested only in the approximate probability of Φ. A probability interval $[L, U]$ such that $P_\Phi \in [L, U]$ can be obtained by noting that the two formulas

$$\Phi_L = x_1 y_1 z_1 \vee x_2 y_4 z_2 \vee x_3 y_5 z_3 \qquad\qquad \Phi_U = z_1 \vee x_1 y_2 \vee x_2 y_4 z_2 \vee x_3$$

are lower and upper model-bounds for Φ in the sense the Φ_L implies Φ which in turn implies Φ_U; consequently, $P_{\Phi_L} \leq P_\Phi \leq P_{\Phi_U}$, i.e., Φ_L and Φ_U give rise to the desired probability interval. The formulas have the special property that each variable symbol occurs only once and for such formulas it is straightforward to evaluate their probability. Moreover, Φ_L and Φ_U are optimal in the sense that there are no DNF formulas in which each variable symbol occurs only once that are better bounds for Φ.

We will show that the above decomposition-based technique for exact probability computation and the model-bounds-based technique for approximating formulas can be merged into a deterministic approximation algorithm that computes the probabilities of queries in probabilistic database up to user-defined error guarantees. □

Contributions

This thesis advances the state of the art in both of the above aspects — complexity analysis and query evaluation — of query processing in probabilistic databases.

Complexity analysis. Thus far, complexity dichotomies are known for non-repeating conjunctive queries and unions of conjunctive queries: When restricted to tuple-independent databases, the data complexity of any query in each of these languages is either #P-hard or in PTIME. Remarkably, the so-called *hierarchical* property provides a simple syntactic characterisation of all tractable queries in the language of non-repeating conjunctive queries.

The central complexity result shown in this thesis is a #P-hard/PTIME dichotomy for non-repeating conjunctive relational algebra queries with negation. The syntactic characterisation of tractable queries in this language is given by the same hierarchical property. The #P-hardness of non-hierarchical queries is shown by an elaborate reduction from the model-counting problem for positive bipartite DNF formulas. The tractability result for

hierarchical queries is obtained by compiling annotation formulas into Ordered Binary Decision Diagrams of size linear in the database.

The complete tractability frontier is described for so-called quantified queries, such as division, set-inclusion and set-equivalence queries, which are expressed using universal quantification and repeating relation symbols. Finally, known classes of tractable queries with only one level of aggregation are generalised to queries with nested aggregation.

Query evaluation. A complete framework is presented for evaluating arbitrary relational algebra queries over probabilistic databases expressed in the pc-tables formalism. The proposed algorithms evaluate queries by decomposing the associated annotation formulas and give rise to exact query probabilities when formulas are exhaustively decomposed, or to approximate query probabilities with error guarantees when formulas are partially decomposed. A core ingredient of the latter is a novel syntactic characterisation of optimal model-based bounds for propositional formulas. In contrast to sampling-based approximation techniques, the proposed approximation algorithm is deterministic and anytime, i.e., the approximation interval provably never worsens between successive iterations.

The exact evaluation technique is extended to queries with aggregation operators to yield the first system to provide support for queries with arbitrarily nested aggregation. To that end, the representation system is lifted from Boolean annotation formulas that correspond to set semantics to semiring and semimodule expressions that can express bag semantics. The decomposition-based evaluation technique is generalised to such expressions.

Extensive experiments complement the theoretical results and demonstrate the scalability of the algorithms and the trade-off between exact and approximate evaluation.

Structure of this thesis

The remainder of this document is organised as follows. Chapter 2 defines notations and recalls the concepts in probabilistic databases relevant to this work. Chapter 3 analyses the data complexity of queries with negation and is complemented by the description of a complete framework for evaluating the probability of relational algebra queries in Chapter 4. Chapter 5 discusses the extension of the latter framework to queries with aggregation operators. Each chapter is prefaced with a high-level introduction of the main challenges and the approach taken to solve them; these introductions comprise informal examples intended to convey an intuitive — rather than mathematically rigorous — understanding of the technical content in the remainder of the chapters. The connection between the results of this thesis and the context of prior and related work is discussed separately in each of the chapters. Listings and statistics of queries used in experimental evaluations are given in the appendix.

Chapter 2

Preliminaries

2.1 Propositional formulas

2.1.1 Syntax

Let $\mathbf{X}$ be a finite set of Boolean variables. A literal is a variable or its negation. A clause is a conjunction of literals. A formula can be constructed using variables and constants $\top$ (true) and $\bot$ (false) using the logical connectives $\vee$ ("or"), $\wedge$ ("and"), and $\neg$ ("not"). We denote by $\mathcal{B}(\mathbf{X})$ the set of thus constructed propositional formulas over variables $\mathbf{X}$ and by $\mathrm{vars}(\Phi)$ the set of variables of the formula Φ. Two formulas Φ and Ψ are independent if their variable sets are disjoint, i.e., $\mathrm{vars}(\Phi) \cap \mathrm{vars}(\Psi) = \emptyset$; otherwise, they are dependent.

We assume that formulas are in negation normal form (NNF), where negation can only appear at literals; any formula can be brought in NNF in linear time. A formula (in NNF) is positive if all of its literals occur positively; unate formulas, i.e., formulas where each variable occurs either only positively or only negatively, can trivially be converted into equivalent positive formulas.

A formula is in disjunctive normal form (DNF) if it is a disjunction of clauses; it is in one-occurrence form (1OF) or read-once, if each of its variables occurs once; it is in independent DNF (iDNF) if it is in 1OF and DNF.

We alternatively see clauses as sets of literals, DNF formulas as sets of clauses, and arbitrary (NNF) formulas as their parse trees: Inner nodes are logical connectives and leaves are variables. Given two DNF formulas (clauses) Φ and Ψ, $\Phi \subseteq \Psi$ means that Φ consists of a subset of the clauses (literals, respectively) of Ψ. The empty set $\{\}$ is the unsatisfiable DNF formula with no clause — corresponding to the constant *false*, the set $\{\{\}\}$ is the tautological DNF formula with one empty clause — corresponding to the constant *true*. By using the set perspective, we implicitly assume that a clause (a literal) only appears once in a DNF formula (clause, respectively); indeed, for all our purposes, we can ignore duplicate clauses and literals; if duplicates are created during processing, then they are trivially removed.

Example 5. The formula $x_1y_1 \vee x_1y_2$ is in DNF, but not in iDNF, since its clauses share variable x_1. Its equivalent formula $x_1(y_1 \vee y_2)$ is in 1OF. Figures 4.4 and 4.5 show the parse trees of two formulas. If the marked leaves are removed, we obtain formulas in 1OF. □

A DNF formula Φ has *arity* (*max-arity*) k if every clause in Φ has exactly (at most, respectively) k variables.

Given two disjoint sets of variables, $\mathbf{X}$ and $\mathbf{Y}$, a DNF formula is *bipartite* over $\mathbf{X}$ and $\mathbf{Y}$ if each clause has the form $x \wedge y$ with one variable $x \in \mathbf{X}$ and one variable $y \in \mathbf{Y}$. The set of positive bipartite DNF formulas is denoted by 2DNF.

A convenient way of representing a positive bipartite DNF formula is by first labelling the variables by natural numbers — $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ — and then representing each clause by a pair $(i, j) \in \mathbb{N} \times \mathbb{N}$. A set $E \subseteq \mathbb{N} \times \mathbb{N}$ of such pairs then defines the formula $\Psi = \bigvee_{(i,j)\in E} x_i y_j$.

2.1.2 Semantics

Given the set $\mathbf{X}$ of variables, we denote by $\Omega_{\mathbf{X}} = \{\nu : \mathbf{X} \to \{\top, \bot\}\}$ the set of possible assignments (or valuations) ν of all variables from $\mathbf{X}$ to constants true and false; when the context is clear, we write Ω for $\Omega_{\mathbf{X}}$. For a formula Φ, its set of assignments is denoted by $\Omega(\Phi) = \{\nu : \mathrm{vars}(\Phi) \to \{\top, \bot\}\}$. A *model* of Φ is a satisfying assignment, i.e., an assignment ν that maps Φ to true, also denoted by $\nu \models \Phi$. The set of models of Φ is denoted by $\mathcal{M}(\Phi)$. Counting the number of models (i.e., determining the number $\#\Psi = |M(\Psi)|$) is #P-complete already for the set of bipartite positive DNF formulas [67].

Given two formulas Φ and Ψ over the same variables, Φ is *semantically* contained in Ψ, denoted by $\Phi \models \Psi$, if $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Psi)$. We also say that Φ *implies* Ψ. Equivalence is two-way containment and denoted by $\equiv$. Given two clauses φ and ψ, it holds that $\varphi \models \psi$ if and only if $\varphi \supseteq \psi$. For DNF formulas, we use the following key result (formulated in terms of tableaux containment, Theorem 3 in [74]):

Lemma 1 ([74])**.** *Let Φ, Ψ be positive DNF formulas. Then,*

$$\Phi \models \Psi \quad \textit{iff} \quad \forall \varphi \in \Phi\ \exists \psi \in \Psi\ :\ \varphi \models \psi \tag{2.1}$$

$$\textit{iff} \quad \forall \varphi \in \Phi\ \exists \psi \in \Psi\ :\ \varphi \supseteq \psi. \tag{2.2}$$

A positive DNF formula Φ is *reducible* if there exist clauses $\varphi, \psi \in \Phi$ that satisfy $\varphi \models \psi$, in which case φ is redundant and can be removed; otherwise, it is *irreducible*.

Corollary 2 (Lemma 1)**.** *Two irreducible positive DNF formulas are equivalent if and only if they have the same clauses.*

This statement follows immediately from Lemma 1. Computing the irreducible equivalent of a positive DNF formula can be done in polynomial time.

2.1.3 Probabilistic interpretation

Let $\mathbf{X}$ be a set of independent Boolean random variables. For each variable $x \in \mathbf{X}$, let $P_x[\top]$ be the probability of x being true, and $P_x[\bot]$ the probability of x being false.

The probability mass function $\Pr(\nu) = \prod_{x \in \mathbf{X}} P_x[\nu(x)]$ for each assignment $\nu \in \Omega$ and the probability measure $\Pr(E) = \sum_{\nu \in E} \Pr(\nu)$ for all $E \subseteq \Omega$ define the probability space $(\Omega, 2^\Omega, \Pr)$ that we call the *probability space induced by* $\mathbf{X}$.

A formula Φ can be interpreted as a random variable $\Phi : \Omega \to \{\top, \bot\}$ over $(\Omega, 2^\Omega, \Pr)$ by letting $\Phi : \nu \mapsto \nu(\Phi)$ and with probability distribution defined as

$$\begin{aligned} P_\Phi[\top] &= \sum_{\nu \in \Omega, \nu(\Phi) = \top} \Pr(\nu) \\ P_\Phi[\bot] &= \sum_{\nu \in \Omega, \nu(\Phi) = \bot} \Pr(\nu) \end{aligned} \tag{2.3}$$

We also write P_Φ or $P(\Phi)$ for $P_\Phi[\top]$ and $P_{\neg\Phi}$ or $P(\neg\Phi)$ for $P_\Phi[\bot] = 1 - P_\Phi[\top]$.

In the special case where $P_x = 0.5$ for all variables, the model counting problem directly reduces to the probability computation problem via $P_\Phi = 2^{-|\mathrm{vars}(\Phi)|}\#\Phi$. It follows that the latter problem is $\#P$-hard for the set of positive bipartite DNF formulas. In contrast, formulas in 1OF and iDNF admit probability computation in linear time. Recall that a 1OF formula is a variable or a conjunction or disjunction of independent 1OF formulas. The probability of a 1OF formula Φ can be computed recursively as follows:

$$\text{If } \Phi = \Phi_1 \wedge \cdots \wedge \Phi_n, \text{ then } P(\Phi) = \prod_{i=1}^{n} P(\Phi_i) \tag{2.4}$$

$$\text{If } \Phi = \Phi_1 \vee \cdots \vee \Phi_n, \text{ then } P(\Phi) = 1 - \prod_{i=1}^{n} (1 - P(\Phi_i)) \tag{2.5}$$

$$\text{If } \Phi = x \text{ for a variable } x, \text{ then } P(\Phi) = P_x \tag{2.6}$$

$$\text{If } \Phi = \neg\Phi', \text{ then } P(\Phi) = 1 - P(\Phi') \tag{2.7}$$

2.1.4 Ordered binary decision diagrams

A binary decision diagram (BDD) is a representation formalism for Boolean propositional formulas. A BDD over a set $\mathbf{X}$ of variable symbols is a directed acyclic graph where inner nodes are labelled with literals over $\mathbf{X}$ and terminal nodes are $\top$ or $\bot$. Each inner node has two outgoing edges, one representing that the variable of this node is set to true, and one representing that it is set to false. A BDD is *ordered* (OBDD) if there is a total order Π $\mathbf{X}$ such that the variables on each directed path are visited in Π-order. Such an OBDD is denoted by Π-OBDD to emphasise its variable order. The Boolean formula f represented by an OBDD is defined by the (possibly partial) valuations of $\mathbf{X}$ encoded in its maximal paths: We take the path corresponding to a valuation $\nu : \mathbf{X} \to \{\text{true}, \text{false}\}$ by taking the corresponding edges (edge *true* from a node x with $\nu(x) = \text{true}$ and edge *false* from a node

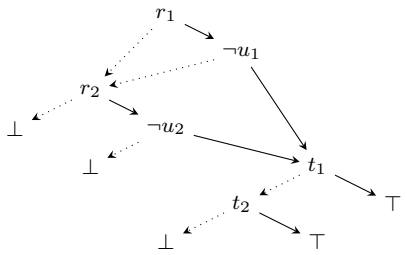


Figure 2.1: OBDD for formula $(r_1 \neg u_1 \vee r_2 \neg u_2) \wedge (t_1 \vee t_2)$; solid lines denote *true*-edges, and dotted lines denote *false*-edges.

x with $\nu(x) = \text{false}$); if we reach a terminal $\top$ then f evaluates to false under ν, else f is true under ν.

Example 6. An OBDD for the formula $(r_1 \neg u_1 \vee r_2 \neg u_2) \wedge (t_1 \vee t_2)$ is given by the diagram in Figure 2.1. In this OBDD, solid lines denote the *true*-edges, and dotted lines the *false*-edges. For example, the path $r_1 \xrightarrow{\text{true}} \neg u_1 \xrightarrow{\text{false}} r_2 \xrightarrow{\text{false}} \bot$ encodes that any valuation ν with $\nu(r_1) = \text{true}$ and $\nu(\neg u_1) = \nu(r_2) = \text{false}$ evaluates the expression $\Psi_1 = (r_1 \neg u_1 \vee r_2 \neg u_2) \wedge (t_1 \vee t_2)$ to false. □

We recall a result regarding Boolean combinations of OBDDs with the same variable order, and a result regarding the complexity of evaluating the probability of a formula over Boolean random variables, given its OBDD:

Lemma 3 (Boolean combination of OBDDs [90])**.** *Let Φ_1, Φ_2 be two propositional formulas and consider a fixed variable order Π on its variables. If there exist Π-OBDDs of width w_1, w_2 for Φ_1, Φ_2, respectively, then there exist Π-OBDDs of width $w_1 \cdot w_2$ for $\Phi_1 \wedge \Phi_2$ and for $\Phi_1 \vee \Phi_2$.*

Lemma 4 (Probability computation in OBDDs [60])**.** *Given an OBDD for a formula Ψ, the probability P_Ψ can be evaluated in time linear in the size of the OBDD.*

2.2 Queries

2.2.1 Relational algebra

We denote by RA the class of standard relational algebra queries under the named perspective [2]. In order to emphasise the schema of a database, relation symbols may carry their attributes in queries; for example, we often write $R(A) \bowtie S(A, B)$ instead of $R \bowtie S$ to indicate implicitly that the schemata of the relations R and S have attributes A and A, B, respectively. The set of attributes of a relation R is $\text{sch}(R)$. A query Q is *non-repeating* if each relation symbol occurs at most once in Q.

Given a query Q and an operator Op in Q, Op has *even polarity* if the number of $-$ operators between Op (exclusive) and the root of Q (inclusive) for which Op is a right

Symbol	Chapter	Definition	Description
RA	4	Section 2.2.1	Full relational algebra
$\text{RA}^{\not\cup}$	3	Definition 3	Non-repeating, union-free relational algebra with equi-joins
RA^{ϖ}	5	Definition 30	Positive relational algebra with aggregation

Figure 2.2: Fragments of relational algebra queries used in this work.

descendant is even, and has *odd polarity* otherwise. The pol function captures the notion of polarity:

$$\text{pol}(Q, Op) = \begin{cases} 1 & \text{if } Op \text{ has odd polarity in } Q \\ 0 & \text{if } Op \text{ has even polarity in } Q \end{cases} \tag{2.8}$$

Definition 1 (Exported variables)**.** *The variables* exported *by a query Q, denoted $\mathcal{E}(Q)$, are defined recursively by the query structure as follows:*

1. *If $Q = Q_1 \bowtie_\rho Q_2$, then $\mathcal{E}(Q) = \mathcal{E}(Q_1) \cup \mathcal{E}(Q_2)$*
2. *If $Q = Q_1 \cup Q_2$, then $\mathcal{E}(Q) = \mathcal{E}(Q_1)$*
3. *If $Q = Q_1 -_\rho Q_2$, then $\mathcal{E}(Q) = \mathcal{E}(Q_1)$*
4. *If $Q = \pi_{\bar{A}}(Q_1)$, then $\mathcal{E}(Q) = \bar{A}$*
5. *If $Q = \sigma_\rho(Q_1)$, then $\mathcal{E}(Q) = \mathcal{E}(Q_1)$*
6. *If $Q = R$, then $\mathcal{E}(Q) = sch(R)$*

The notation Q^A denotes a query Q such that $A \in \mathcal{E}(Q)$, and $Q^{\neg A}$ is a query Q with $A \notin \mathcal{E}(Q)$. Similarly, $Q^{A\neg B}$ denotes a query Q with $A \in \mathcal{E}(Q)$ and $B \notin \mathcal{E}(Q)$.

Queries are assumed to be well-formed in the canonical sense (e.g., in $Q_1 -_\rho Q_2$, the schemata of Q_1 and Q_2 are compatible; the attributes equated in the equi-join $Q_1 \bowtie_\rho Q_2$ and the attributes selected in a projection $\pi_{\bar{A}} Q_1$ must be exported by Q_1 and Q_2).

The fragments of relational algebra used in this work are shown in Figure 2.2.

2.2.2 Relational calculus

Relational calculus queries without universal quantifiers, denoted $\text{RC}^{\exists,\neg}$, are expressions of the form $\{\bar{H} \mid F\}$ where F is a formula defined by the grammar

$$F ::= R(\bar{X}) \mid \exists_X(F_1) \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg(F_1),$$

and $\bar{H}$ is a tuple of variable symbols that occur unquantified in F. The name $\mathrm{RC}^{\exists,\neg}$ emphasises that formulas may syntactically contain existential quantification and negation symbols, but not the universal quantification symbol $\forall$. When the context is clear, we represent a query by its formula F alone and omit the explicit reference to the head variables $\bar{H}$. We say that variable X *is root*[1] in a query $\exists_X(Q)$ if X occurs in every relation symbol in Q. An $\mathrm{RC}^{\exists,\neg}$ query in which no disjunction operator occurs is called *disjunction-free*. A query in which each distinct relation symbol appears only with the same variable identifiers is *canonicalised*, viz:

Definition 2 (Canonicalised $\mathrm{RC}^{\exists,\neg}$query)**.** *An $RC^{\exists,\neg}$ query Q is* canonicalised *if every occurrence of a relation symbol $R(\bar{X})$ in Q has the same variable symbols $\bar{X}$.*

The semantics of evaluating a query $Q = \{\bar{H} \mid F\}$ over a database $\mathcal{D}$ is standard: A tuple $\bar{A}$ is an answer to Q if $F[\bar{A}/\bar{H}]$ is satisfied by $\mathcal{D}$.

2.3 Probabilistic databases

The probabilistic database formalism considered in this work is that of pc-tables, where each input tuple is annotated by a propositional formula over independent Boolean random variables [79]. Such formulas define the probabilities of database tuples and can express arbitrary correlations between tuples, such as conditional independence and positive and negative correlations. Following the standard deterministic case where queries map between relational databases, in pc-tables queries map between pc-tables: The query results are pc-tables, whose tuples are computed as in the deterministic case and their associated formulas are constructed from the input formulas using conjunction, disjunction, and negation, and reflect the derivation of the result from the input [45]. The pc-tables formalism subsumes tuple-independent probabilistic databases, such as NELL tables learned from the web [16], and block-independent disjoint databases, such as Google Squared tables that aggregate Google Search results [31] and probabilistic tables obtained from information extraction models [43]. It can also capture more complex correlations that arise when conditioning probabilistic databases using constraints [55].

2.3.1 Syntax and semantics

Probabilistic c-multitables, or pc-tables for short, are relational databases where each tuple is annotated with a propositional formula over a set $\mathbf{X}$ of independent Boolean random variables [45, 79]. Under *possible worlds semantics*, a pc-table $\mathcal{D}$ represents a finite set of possible worlds: Each valuation ν of the variables in $\mathbf{X}$ defines a possible world D_ν representing a deterministic relational database consisting of exactly those tuples in $\mathcal{D}$ whose

[1]The term *root variable* was first coined in the context of conjunctive queries [24], where the class of hierarchical queries allows for an arrangement of the query variables in a specific tree, such that the variables that occur in all relation symbols under the scope of its quantifier correspond to the roots of the sub-trees.

$$
\begin{aligned}
[\![R]\!] &= \text{R} \\
[\![\delta_{A\to B}(Q)]\!] &= \text{select R.*, R.A as B, R.}\Phi \text{ from } ([\![Q]\!])\ \text{R} \\
[\![\sigma_{\rho}(Q)]\!] &= \text{select R.* from } ([\![Q]\!])\ \text{R where } \rho \\
[\![\pi_{A}(Q)]\!] &= \text{select R.A, } \textstyle\bigvee\text{(R.}\Phi\text{) as } \Phi \text{ from } ([\![Q]\!])\ \text{R group by R.A} \\
[\![Q_1 \bowtie_{\rho} Q_2]\!] &= \text{select R.*, S.*, R.}\Phi\wedge\text{S.}\Phi \text{ as } \Phi \text{ from } ([\![Q_1]\!])\ \text{R}, ([\![Q_2]\!])\ \text{S where } \rho \\
[\![Q_1 \cup Q_2]\!] &= \text{select R.*, } \textstyle\bigvee\text{(R.}\Phi\text{) as } \Phi \text{ from } \big(\text{select * from } ([\![Q_1]\!]) \text{ union all select * from } ([\![Q_2]\!])\big)\ \text{R group by R.*} \\
[\![Q_1 - Q_2]\!] &= \text{select R.*, R.}\Phi \wedge \neg\ \text{S.}\Phi \text{ as } \Phi \text{ from } ([\![Q_1]\!])\ \text{R left out join } ([\![Q_2]\!])\ \text{S on R.* = S.*}
\end{aligned}
$$

Algorithm 1: Query rewriting procedure for RA queries over pc-tables.

annotations are satisfied by ν. The probability of each world is the product of probabilities of the variable assignments in ν, i.e., $\Pr(\nu)$. This construction yields a probability space over the possible worlds of $\mathcal{D}$ that is isomorphic to the probability space induced by $\mathbf{X}$. The pc-tables formalism is complete in the sense that it can represent arbitrary probability distributions over any finite set of possible worlds. It is a strong representation system since query results can be represented as pc-tables.

A pc-tables database is called *tuple-independent* if the annotations of its tuples are pairwise independent formulas.

Remark 1. The original definition of c-tables [45] allows variables both as annotations and as the "data". In this thesis, we will use variables only in annotations; whenever variable symbols appear in tuples, they are treated as constants. For example, in Figure 3.2, the relation $T \bowtie S$ contains a tuple $(1, x_1)$ with annotation formula $\neg x_1$; under the possible world semantics, this annotated tuple represents the deterministic tuple $(1, x_1)$ — where x_1 is just a constant — in worlds ν with $\nu(\neg x_1) = \text{true}$. □

2.3.2 Query evaluation

Given a query Q and a pc-table $\mathcal{D}$ over variables $\mathbf{X}$, the query evaluation problem is to compute the distinct tuples in the results of Q in the worlds of $\mathcal{D}$ together with their probabilities. The probability $P(\bar{A} \in Q(\mathcal{D}))$ of a tuple $\bar{A}$ is the probability that $\bar{A}$ is in the result of Q in a world randomly drawn from $\mathcal{D}$:

$$P(\bar{A} \in Q(\mathcal{D})) = \sum_{\nu \in \Omega_{\mathbf{X}} : \bar{A} \in Q(D_\nu)} \Pr(\nu) \tag{2.9}$$

We adopt the *intensional approach* to query evaluation [79]: For each answer tuple $\bar{A}$, first construct a propositional formula $\Phi_{\bar{A}\in Q(\mathcal{D})}$ that annotates $\bar{A}$ such that $P(\bar{A} \in Q(\mathcal{D})) = P(\Phi_{\bar{A}\in Q(\mathcal{D})})$, and then compute $P(\Phi_{\bar{A}\in Q(\mathcal{D})})$ as per Equation (2.3). The annotation of a Boolean query Q is denoted by $\Phi_{Q(\mathcal{D})}$; when the context is clear, we often omit the explicit reference to the database $\mathcal{D}$ and simply write Φ_Q.

We next discuss how annotation formulas can be computed in the case of relational

R		S			T	
A	Φ	A	B	Φ	B	Φ
1	x_1	1	1	y_1	1	z_1
2	x_2	1	2	y_2	2	z_2
3	x_3	2	1	y_3	3	z_3
		2	2	y_4	4	z_4
		3	3	y_5		
		3	4	y_6		

$$Q_1 = \pi_\emptyset(R \bowtie_A S \bowtie_B T)$$
$$Q_2 = \pi_\emptyset(R \bowtie_{R.A \leq S.A} S \bowtie_B T)$$

$$\Phi_1 = x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \vee x_3y_6z_4$$
$$\Phi_2 = x_1y_1z_1 \vee x_1y_2z_2 \vee x_1y_3z_1 \vee x_1y_4z_2 \vee x_1y_5z_3 \vee x_1y_6z_4 \vee$$
$$x_2y_3z_1 \vee x_2y_4z_2 \vee x_2y_5z_3 \vee x_2y_6z_4 \vee x_3y_5z_3 \vee x_3y_6z_4$$

Figure 2.3: A pc-table with relations R, S, T annotated with variables from disjoint sets $V_1 = \{x_1, \ldots, x_3\}$, $V_2 = \{y_1, \ldots, y_6\}$, $V_3 = \{z_1, \ldots, z_4\}$, respectively. Formulas Φ_1 and Φ_2 are the annotations of the results of queries Q_1 and Q_2 evaluated on this database.

algebra and relational calculus queries.

Annotations of relational algebra queries. The tuples together with their annotation in the result of a relational algebra query Q_{RA} can be computed directly from the input pc-table $\mathcal{D}$. This is achieved by rewriting Q_{RA} into a query Q' such that standard relational evaluation of Q' on $\mathcal{D}$ yields a pc-table $\mathcal{D}'$ representing the results of Q_{RA} in the worlds of $\mathcal{D}$. Algorithm 1 specifies such a rewriting function $[\![\cdot]\!]$ for any relational algebra query. It assumes that the formulas annotating the tuples in the input pc-table are stored in a distinguished column called Φ; for a relation R, we consider that this column is not selected by the selector `R.*`. The rewriting is expressed here in SQL[2] and — besides a straightforward encoding of the relational operators in SQL — it constructs formulas annotating result tuples based on the formulas of input tuples as follows. In case of identity, selection, and renaming operators, the input annotations are just copied in the result. For projection and union, the formula of each distinct result tuple is constructed as the disjunction of all input tuples with the same restriction to the attributes in the projection list. For the join operator, the formula of a result tuple is the conjunction of the formulas of the contributing input tuples; to avoid cluttering, we slightly abuse notation in stating the attributes of the select clause: `R.*, S.*` means here the set-union of the attributes in R and S. A tuple $\bar{A}$ in the result of $Q_1 - Q_2$ has annotation Φ_1 if $\bar{A}$ is in Q_1 with annotation Φ_1 and $\bar{A}$ is not in Q_2, and has annotation $\Phi_1 \wedge \neg\Phi_2$ if $\bar{A}$ is in Q_1 with annotation Φ_1 and $\bar{A}$ is in Q_2 with annotation Φ_2. These two cases are implemented in $[\![\cdot]\!]$ by a left outer join.

Example 7. Figure 2.3 shows pc-tables, where each tuple is annotated by a distinct Boolean random variable stored in column Φ, and the formulas Φ_1 and Φ_2 annotating the results of two Boolean queries Q_1 and Q_2, respectively. The answers to the queries in the figure are the sets consisting of the empty tuple whose probability is $P(\Phi_1)$ and $P(\Phi_2)$, respectively.

[2]Note that the choice of SQL for expressing the rewritten query is by no means exclusive, but merely convenient: First, the language necessarily has to support bag semantics (as per SQL's `union all` and `group by`), and second, the SQL rewriting is both concise and very close to the original RA query and hence easy to understand.

Note that the relations in this example are tuple-independent; the propagation rules for tuple annotations are however the same for tuple-independent and general pc-tables. In this thesis, tuple-independent databases are used in sections on complexity analysis (Chapter 3 and Section 5.4), while the decomposition-based query evaluation techniques presented in Chapter 4 and Chapter 5 are defined for general, i.e., not necessarily tuple-independent, databases. □

Annotations of relational calculus queries. Given a query $Q_{RC} = \{\bar{H} \mid F\}$, a tuple $\bar{A}$ has annotation $\Phi_{\bar{A} \in Q_{RC}(\mathcal{D})} = \Phi_{F[\bar{A}/\bar{H}]}$ which is inductively defined as follows:

$$\begin{aligned}
\Phi_{R(\bar{A})} &= \begin{cases} \bot & \text{if relation } R \text{ has no tuple } \bar{A} \text{ in } \mathcal{D} \\ \Phi & \text{if relation } R \text{ has tuple } \bar{A} \text{ in } \mathcal{D} \text{ with annotation } \Phi \end{cases} \\
\Phi_{\exists_X(Q)} &= \bigvee_{A \in \mathrm{ADom}(\mathcal{D})} \Phi_{Q[A/X]} \\
\Phi_{Q_1 \wedge Q_2} &= \Phi_{Q_1} \wedge \Phi_{Q_2} \\
\Phi_{Q_1 \vee Q_2} &= \Phi_{Q_1} \vee \Phi_{Q_2} \\
\Phi_{\neg(Q)} &= \neg(\Phi_Q)
\end{aligned} \tag{2.10}$$

2.4 Computational complexity

#P is the class of counting problems corresponding to decision problems in NP. More specifically, it is the class of problems of the form "compute $f(x)$", where $f(x)$ is the number of accepting paths of a standard, non-deterministic, polynomial-time Turing machine [85, 7]. A problem is #P-hard if every problem in #P can be reduced to it by a polynomial-time Turing reduction that relates their number of solutions. The canonical #P-complete problem is #SAT, i.e., the problem of determining the number of satisfying assignments of a Boolean formula. #SAT remains #P-complete when restricted to bipartite, positive 2DNF formulas [67].

Encoding and resource model. We assume an encoding of probabilistic databases that is based on the standard encoding of deterministic database instances [2], in which values are mapped to their binary representation, and tuples and relations are encoded as the concatenation of their values and tuples, respectively. Tuple annotations can be incorporated into this encoding by treating them as distinct, separate attribute Φ of type `string` in each relation. The probability distributions of the annotation variables are considered part of the input database and can be encoded as a distinct relation mapping variables to their probabilities. The complexity results in this work are with respect to the standard RAM model of computation, and, in particular, fixed-cost arithmetic, i.e., assuming that addition and multiplication of probability values can be performed in constant time.

Remark 2. All complexity results in this work can be lifted from fixed-cost arithmetic to bit-cost arithmetic with only polynomial overhead. To see this, assume that the input probabilities of the random variables $x_1, \ldots, x_n$ annotating the input database can be represented in $b_1, \ldots, b_n$ bits; note that the probabilities are considered part of the input, i.e., $b_1 + \cdots + b_n = \mathcal{O}(|D|)$. The probability of a query result $\bar{A}$ is the sum of the probabilities of the possible worlds in which $\bar{A}$ is a result, cf. Equation (2.9); there are 2^n possible worlds and the probability of each world is the product of the corresponding n probabilities and can be represented in $\mathcal{O}(b_1 + \ldots b_n) = \mathcal{O}(|D|)$ bits. The sum of $2^n = \mathcal{O}(2^{|D|})$ numbers of bit-size $\mathcal{O}(|D|)$ can be represented in $\mathcal{O}(|D|)$ bits.

The equivalence of fixed-cost and bit-cost arithmetic is a consequence of the restricted expressiveness of the query languages considered in this work. More powerful query languages, such as recursive Markov chains, can succinctly represent doubly-exponentially many possible worlds. In such cases, the representation of the probabilities of possible worlds can require a number of bits exponential in the size of the input; this gives rise to an exponential gap between the complexity in terms of fixed-cost and bit-cost arithmetic [9].

Data complexity considers a fixed query and examines the complexity of the query evaluation problem in terms of the size of the database. Conversely, query complexity assumes a fixed database and focuses on the complexity in terms of the size of the query. Combined complexity measures the complexity in terms of both data and query [86, 2].

Complexity of relational algebra on deterministic databases. With respect to data complexity and standard, deterministic databases, RA and $\text{RA}^{\not\emptyset}$ are in LOGSPACE and RA^{ϖ} is in PTIME; with respect to query complexity, RA is PSPACE-complete [86, 2].

Upper bounds for the complexity of relational algebra on probabilistic databases. For any query language, the probabilistic query evaluation problem reduces to the standard, deterministic query evaluation problem via Equation 2.9: In terms of data complexity, the naïve algorithm that enumerates all $\mathcal{O}(2^{|D|})$ possible worlds, evaluates the given query in each world, and sums up the probabilities of those worlds that satisfy the query has an EXPTIME overhead over deterministic query evaluation. All query languages considered in this paper are in PTIME data complexity for the deterministic case; this yields EXPTIME data complexity as the upper bound for probabilistic query evaluation.

By the same argument, the combined complexity of any query language that is hard for EXPTIME (or any class containing EXPTIME) in terms of combined complexity of deterministic evaluation carries over to the probabilistic case. For example, the combined complexity of RA is PSPACE-complete in both the deterministic and the probabilistic case. The combined complexity of any language that is in EXPTIME in the deterministic case is in EXPTIME in the probabilistic case.

Chapter 3

The dichotomy of non-repeating, union-free relational algebra queries

The main result proven in this chapter is a dichotomy for the language $\mathrm{RA}^{\not\cup}$ of non-repeating, union-free relational algebra queries with equi-joins[1] with respect to their data complexity: So-called *hierarchical* queries on tuple-independent databases have PTIME data complexity, and all other queries are hard for #P. The significance of this dichotomy is that it provides a PTIME algorithm for a class of queries for which the naïve evaluation is in EXPTIME for data complexity; the data complexity of $\mathrm{RA}^{\not\cup}$ is in LOGSPACE in the deterministic case, and the combined complexity of $\mathrm{RA}^{\not\cup}$ is in PSPACE for both the deterministic and the probabilistic case, cf. Section 2.4.

The notion of hierarchical queries was first coined in the context of conjunctive queries [24]: The class of hierarchical, non-repeating conjunctive queries allows for a hierarchical arrangement of the variables and relation symbols of the query that corresponds to the factorisation and nesting order of the read-once annotation of the query evaluated over a tuple-independent database [60].

Intuitively, a Boolean query is hierarchical if it does not contain three relation symbols $R(A)$, $S(A, B)$, $T(B)$ and its joins or difference operations enforce equality constraints between the A-attributes of R and S, and between the B-attributes of S and T, but not between any A-attributes and B-attributes. Such a constraint can be established by means of a join condition (e.g., in the query $R(A) \bowtie S(A, B) \bowtie T(B)$ with equi-joins between R, S, T), or via a difference operator (e.g., in the query $S(A, B) - [R(A) \bowtie T(B)]$ the join operator enforces no constraints on the attributes of R and T, but the difference operator encodes constraints between the A-attributes of S and R and between the B-attributes of S and T).

[1] The formal definition of $\mathrm{RA}^{\not\cup}$ queries is given in Definition 3 below.

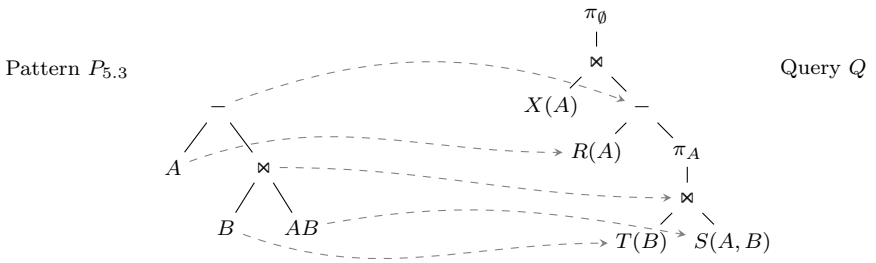


Figure 3.1: Illustration of a match between a pattern (left) and a query (right).

The syntactic characterisations of hierarchical non-repeating conjunctive queries [22, 25] on the one hand, and of $\text{RA}^{\not\emptyset}$ queries on the other hand, are equivalent when each difference operator in a $\text{RA}^{\not\emptyset}$ query is replaced by an equi-join on all pairs of attributes exported by its two sub-queries. Consequently, our result extends the dichotomy *#P-hard vs. PTIME* of query evaluation on tuple-independent database from non-repeating conjunctive queries to the case of non-repeating conjunctive relational algebra queries with negation:

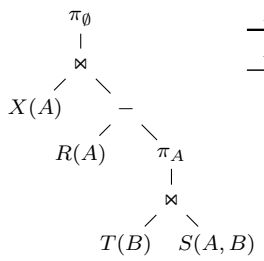
Theorem 5. *For any $RA^{\not\emptyset}$ query on tuple-independent databases, the probabilistic query evaluation problem has #P-hard data complexity if the query is non-hierarchical, and PTIME data complexity if it is hierarchical.*

Proof. The statement follows from Lemmata 7 and 13. □

Furthermore, we specify a LOGSPACE decision procedure that classifies queries as hierarchical or non-hierarchical. The existence of an efficiently decidable criterion to distinguish PTIME from #P-hard queries is crucial, since the complexity gap for the query evaluation problem is large: We will see in Chapter 4 that the evaluation of tractable queries on probabilistic data incurs virtually no overhead when compared to the deterministic case, while #P-hard queries are substantially harder [6].

Hardness proof for non-hierarchical queries

We prove that every non-hierarchical query has #P-hard data complexity by reduction from the #P-hard model-counting problem for positive bipartite DNF formulas. The starting point for the analysis of non-hierarchical queries is to show that the hierarchical property has an equivalent characterisation by means of an embedding into one of 48 basic *patterns* which will help to craft a specific hardness reduction for every query, based on which pattern is embeddable. A *pattern* is a concise graphical representation of an infinite class of queries that satisfy certain structural properties as specified by the pattern. The connection between patterns and their represented queries is formalised by the notion of a *match*. For example, the query Q in the right diagram of Figure 3.1 is non-hierarchical as witnessed by the three relations R, S, T, and it matches the pattern shown in the left diagram of the



R, X		T		S			$T \bowtie S$			$\pi_A(T \bowtie S)$		$R - \pi_A(T \bowtie S)$	
A	$\mathbf{\Phi}$	B	$\mathbf{\Phi}$	A	B	$\mathbf{\Phi}$	A	B	$\mathbf{\Phi}$	A	$\mathbf{\Phi}$	A	$\mathbf{\Phi}$
1	$\top$	x_1	$\neg\mathbf{x_1}$	1	x_1	$\top$	1	x_1	$\neg\mathbf{x_1}$	1	$\neg\mathbf{x_1} \vee \neg\mathbf{y_1}$	1	$\mathbf{x_1 y_1}$
2	$\top$	y_1	$\neg\mathbf{y_1}$	1	y_1	$\top$	1	y_1	$\neg\mathbf{y_1}$	2	$\neg\mathbf{x_1} \vee \neg\mathbf{y_2}$	2	$\mathbf{x_1 y_2}$
		y_2	$\neg\mathbf{y_2}$	1	y_2	$\bot$	1	y_2	$\bot$				
				2	x_1	$\top$	2	x_1	$\neg\mathbf{x_1}$				
				2	y_1	$\bot$	2	y_1	$\bot$				
				2	y_2	$\top$	2	y_2	$\neg\mathbf{y_2}$				

Figure 3.2: Sketch of a hardness reduction for a non-hierarchical query Q (left-most diagram) matching pattern $P_{5.3}$. Note, that to avoid clutter — and in contrast to the naming convention employed in Section 3.4 — the depicted query uses the same attribute names across multiple relations and assumes an equi-join between T and S; the difference operator is unambiguous because its child queries export exactly one attribute A.

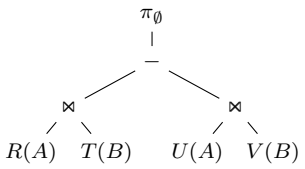
figure. Intuitively, the query matches the pattern, because the arrangement of the three relation symbols $R(A)$, $S(A, B)$, $T(B)$ and the operators connecting them in the query correspond to the structure of the attributes A and B and the operators in the pattern; this is indicated by the arrows in the illustration.

Example 8. Let us consider the query Q from above and exemplify our reduction from the $\#P$-complete model-counting problem for positive bipartite DNF formulas to probability computation for Q. The query and the relations and query results involved in the reduction are shown in Figure 3.2, and the reduction for Q is exemplified for the problem instance $\Psi = x_1y_1 \vee x_1y_2$. The first insight is that the reduction used is determined solely by the pattern that Q matches, i.e., there is a specific reduction for each pattern. The core challenge is to specify a database for the relation symbols that establish the match (R, S, T, in this example) such they give rise to annotation formula Ψ when Q is evaluated over this database. The remaining relation symbols (X, in this example) are populated such that they leave the annotations introduced by R, S, T unaltered.

The tables in Figure 3.2 detail such a database $\mathcal{D}$ in which the relations R, S, T are populated[2] such that the annotation of the query result yields exactly Ψ. Relation X is filled in such way that it leaves these annotations unaltered. The tables $T \bowtie S$, $\pi_A(T \bowtie S)$ and $R - \pi_A(T \bowtie S)$ in the figure illustrate how the relations and their annotations are propagated through the sub-queries of Q.

The final projection operator $\pi_\emptyset$ in Q yields as annotation of the (Boolean) query answer the disjunction of the annotations x_1y_1 and x_1y_2 of tuples (1) and (2) in the relation $R - \pi_a(T \bowtie S)$, i.e., exactly the formula Ψ. If we choose probabilities 0.5 for all three annotation variables x_1, y_1, y_2, then the model-counting problem $\#\Psi$ is reduced to the problem of computing the probability $P_{Q(\mathcal{D})}$ via $\#\Psi = 2^3 P_\Psi = 2^3 P_{Q(\mathcal{D})}$. □

[2]We consider variable symbols that occur in tuples (as opposed to in annotations of tuples) as constants, cf. Remark 1.



R		T		U		V		$R \bowtie T$			$R \bowtie T - U \bowtie V$		
A	$\mathbf{\Phi}$	B	$\mathbf{\Phi}$	A	$\mathbf{\Phi}$	B	$\mathbf{\Phi}$	A	B	$\mathbf{\Phi}$	A	B	$\mathbf{\Phi}$
1	$\mathbf{r_1}$	1	$\mathbf{t_1}$	1	$\mathbf{u_1}$	1	$\mathbf{v_1}$	1	1	$\mathbf{r_1 t_1}$	1	1	$\mathbf{r_1 t_1 \neg(u_1 v_1)}$
2	$\mathbf{r_2}$	2	$\mathbf{t_2}$	2	$\mathbf{u_2}$	2	$\mathbf{v_2}$	1	2	$\mathbf{r_1 t_2}$	1	2	$\mathbf{r_1 t_2 \neg(u_1 v_2)}$
								2	1	$\mathbf{r_2 t_1}$	2	1	$\mathbf{r_2 t_1 \neg(u_2 v_1)}$
								2	2	$\mathbf{r_2 t_2}$	2	2	$\mathbf{r_2 t_2 \neg(u_2 v_2)}$

Figure 3.3: Hierarchical query Q and a tuple-independent database $\mathcal{D}$ for its relations. The tables $R \bowtie T$ and $R \bowtie T - U \bowtie V$ show how the annotations of R, T, Y, V are propagated by Q.

PTIME algorithm for hierarchical queries

The annotation of a hierarchical[3] query on tuple-independent databases is representable by an ordered binary decision diagram (OBDD) of asymptotic size linear in the database; it is well-known that the probability of such a diagram can be evaluated in linear time. In contrast to conjunctive queries [60] — and similar to the case of unions of conjunctive queries [48] — the class of PTIME queries strictly contains the class of queries that give rise to read-once annotations, i.e., there exist PTIME queries with non-read-once annotations.

Example 9. Consider the query Q with respect to the database $\mathcal{D}$ depicted in Figure 3.3. The query is hierarchical because it does not contain a relation symbol with both attributes A and B. Q evaluated on $\mathcal{D}$ yields the annotations depicted in the tables. After the final projection $\pi_\emptyset$, the Boolean result of Q is annotated with the formula

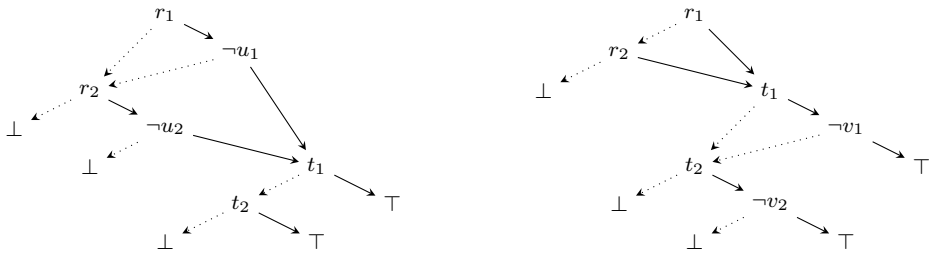
$$\Psi = r_1\big[t_1(\neg u_1 \vee \neg v_1) \vee t_2(\neg u_1 \vee \neg v_2)\big] \vee r_2\big[t_1(\neg u_2 \vee \neg v_1) \vee t_2(\neg u_2 \vee \neg v_2)\big]$$

Note that Ψ is not equivalent to a read-once formula. In contrast to hierarchical non-repeating conjunctive queries, the difference operator in $\mathrm{RA}^{\not\emptyset}$ queries may entangle the annotations of the participating relations such that the resulting annotation is not a read-once formula; this entanglement is the pivotal intricacy introduced by the difference operator. We will show in Section 3.2 that for every tuple-independent database $\mathcal{D}$, the annotation of the result of Q on $\mathcal{D}$ has an OBDD of size $\mathcal{O}(|\mathcal{D}|)$. The underlying idea is to translate Q into a union of disjunction-free relational calculus queries such that each of the disjuncts gives rise to a linear-sized OBDD and all OBDDs have compatible variable orders. We denote the language of such queries by $\mathrm{RC}^{\exists,\neg}$. For Q, this expansion yields the $\mathrm{RC}^{\exists,\neg}$ query

$$Q_{RC} = \underbrace{\exists_A\big(R(A) \wedge \neg U(A)\big) \wedge \exists_B T(B)}_{Q_1} \quad \vee \quad \underbrace{\exists_A R(A) \wedge \exists_B\big(T(B) \wedge \neg V(B)\big)}_{Q_2}.$$

When evaluated over database $\mathcal{D}$ from Figure 3.3, the annotation formulas represented by

[3]The formal definition of hierarchical queries relies on the notation introduced in Section 3.1 and is therefore deferred to Definition 6 on page 26. The intuition for the hierarchical property is given in the beginning of this chapter.



OBDD for the annotation Ψ_1 of query Q_1 OBDD for the annotation Ψ_2 of query Q_2

Figure 3.4: OBDDs for the annotations Ψ_1 (left) and Ψ_2 (right) of queries Q_1 and Q_2 in Example 9.

the two queries Q_1 and Q_2 are

$$\Psi_1 = (r_1 \neg u_1 \vee r_2 \neg u_2) \wedge (t_1 \vee t_2) \qquad \Psi_2 = (r_1 \vee r_2) \wedge (t_1 \neg v_1 \vee t_2 \neg v_2)$$

and clearly $\Psi_1 \vee \Psi_2 \equiv \Psi$. We will show that the $\mathrm{RC}^{\exists,\neg}$ expressions Q_1 and Q_2 can be written such that (i) for each quantifier $\exists_X(Q')$ every relation symbol in Q' contains variable X, and (ii) the nesting order of the quantifiers is the same in Q_1 and Q_2. Property (1) entails that the annotations Ψ_1 and Ψ_2 represented by Q_1 and Q_2 allow for an OBDD of size $\mathcal{O}(|\mathcal{D}|)$, as exemplified in the diagrams of Figure 3.4.

Property (ii) implies that the above linear-sized OBDDs for each disjunction-free part in Q_{RC} can be constructed under the same variable order, and it follows from classic results regarding Boolean combinations of OBDDs [90] that the disjunction $\Psi_1 \vee \Psi_2$ of the OBDDs has asymptotic size $\mathcal{O}(|\mathcal{D}|)$. Thus, the probability of this OBDD — and hence the probability $P_{Q(\mathcal{D})}$ — can be computed in time $\mathcal{O}(|\mathcal{D}|)$. □

Universal quantification and reasoning on sets

The second contribution of the chapter is an analysis of the data complexity of queries with repeating relation symbols and nested negations that can express universal quantification and complex binary relationships amongst sets of entities, such as relational division and set inclusion, equality, or incomparability. We show that while some of these are #P-hard, others admit polynomial-time evaluation for tuple-independent databases. Such queries are used for instance in decision support applications, such as insurance and health-care [69] or data mining applications [68] and it is important to understand to their computational complexity when issued over probabilistic data.

Example 10. Consider a tuple-independent TPCH-like database consisting of a supplier relation S with two columns *sid* for supplier key and *pid* for product key, and a product relation P with only a product key *pid* that contains keys of all products of a given brand (cf. Figure 3.5). Then the relational division query $S \div P$ returns suppliers that supply all products of this brand.

Supplier S

sid	pid	Φ
1	1	$\mathbf{x_1}$
1	2	$\mathbf{x_2}$
1	3	$\mathbf{x_3}$
1	7	$\mathbf{x_4}$
2	1	$\mathbf{x_5}$
2	2	$\mathbf{x_6}$
2	5	$\mathbf{x_7}$

Product P

pid	Φ
1	$\mathbf{y_1}$
2	$\mathbf{y_2}$
3	$\mathbf{y_3}$

Relational division $S \div P$

sid	Φ
1	$(\mathbf{x_1}+\mathbf{x_2}+\mathbf{x_3}+\mathbf{x_4})\neg[(\mathbf{x_1}+\mathbf{x_2}+\mathbf{x_3}+\mathbf{x_4})(\mathbf{y_1}\neg\mathbf{x_1}+\mathbf{y_2}\neg\mathbf{x_2}+\mathbf{y_3}\neg\mathbf{x_3})]$ $=(\mathbf{x_1}+\mathbf{x_2}+\mathbf{x_3}+\mathbf{x_4})(\mathbf{y_1}\rightarrow\mathbf{x_1})(\mathbf{y_2}\rightarrow\mathbf{x_2})(\mathbf{y_3}\rightarrow\mathbf{x_3})$
2	$(\mathbf{x_5}+\mathbf{x_6}+\mathbf{x_7})\neg[(\mathbf{x_5}+\mathbf{x_6}+\mathbf{x_7})(\mathbf{y_1}\neg\mathbf{x_5}+\mathbf{y_2}\neg\mathbf{x_6}+\mathbf{y_3})]$ $=(\mathbf{x_5}+\mathbf{x_6}+\mathbf{x_7})(\mathbf{y_1}\rightarrow\mathbf{x_5})(\mathbf{y_2}\rightarrow\mathbf{x_6})(\neg\mathbf{y_3})$

Figure 3.5: Supplier S, Product P, Division $S \div P = \pi_{\text{sid}}(S) - \pi_{\text{sid}}(\pi_{\text{sid}}(S) \bowtie P - S)$.

The query can be expressed as stated in Figure 3.5. In a deterministic setting, the answer would only consist of the tuple with sid 1, since this is the only one paired in S with all pids in P. In a probabilistic setting, the third tuple from P can be absent with probability $1 - P_{y_3}$, where P_{y_3} is the probability that this tuple is present. In that case, sid 2 can be an answer to $S \div P$ although sid 2 is not paired with pid 3 in S. The set of instances that witness sid $\in \{1, 2\}$ in the answer is defined by the annotation expressions over the input random variables given in column Φ.

We explain the annotation for sid 1. The projection $\pi_{\text{sid}}(S)$ corresponds to the formula $x_1 \vee \cdots \vee x_4$: Indeed, if at least one of the variables is true, then sid 1 is part of the answer. The subsequent cross product with P generates the tuple $(1, i)$ with annotation $(x_1 \vee \cdots \vee x_4)y_i$ for $1 \leq i \leq 3$. The difference with S keeps all tuples in the product and their annotations are extended by the negation of the annotation of the corresponding tuples in S: In case of tuple $(1, i)$, its annotation becomes $(x_1 \vee \cdots \vee x_4)y_i\neg x_i$. The next projection yields the disjunction of all annotations for tuples $(1, 1)$, $(1, 2)$, and $(1, 3)$: $(x_1 \vee \cdots \vee x_4)[y_1\neg x_1 \vee y_2\neg x_2 \vee y_3\neg x_3]$. The outermost difference produces the annotation as shown in Figure 3.5.

The final annotation reads as follows: sid 1 is in the answer provided at least one of the tuples with sid 1 is present in S, and if pid $= i$ is present in P, then the tuple $(1, i)$ must be present in S. We show in Section 3.5 that the probability of this annotation can be computed in time linear in the size of the input database. □

Organisation of this chapter

The remainder of this chapter is organised as follows. Section 3.1 gives the formal definition of the hierarchical property and presents a LOGSPACE algorithm for deciding it. Section 3.2 presents the PTIME procedure for computing probabilities of hierarchical queries. Section 3.3 introduces the techniques used in the hardness proofs for non-hierarchical queries, and Section 3.4 details the #P-hardness reductions themselves. The chapter closes with tractability results for quantified queries in Section 3.5 and a discussion of related work and open research directions in Section 3.6.

3.1 The hierarchical property for relational algebra queries

This section defines the hierarchical property for $\mathrm{RA}^{\not\cup}$ queries formally, and analyses the complexity of the decision problem *"Given a query Q, is Q hierarchical?"* by means of an explicit algorithm.

3.1.1 Non-repeating conjunctive relational algebra queries with equi-joins and difference

We start by formalising the subset $\mathrm{RA}^{\not\cup}$ of relational algebra queries treated in this chapter. We assume a database schema in which attribute names are unique throughout all relations.

Definition 3 ($\mathrm{RA}^{\not\cup}$ queries)**.** *$RA^{\not\cup}$ is the class of non-repeating, union-free relational algebra queries composed from the following operators:*

- *Relation symbols*
- *Equi-join: $Q_1 \bowtie_\rho Q_2$, where ρ is a conjunction of equality conditions $\rho = (A_1{=}B_1) \wedge \cdots \wedge (A_n{=}B_n)$ such that all A_i are attributes of Q_1 and all B_i are attributes of Q_2; note that the equality conditions are 1-to-1, i.e., each attribute of Q_1 and Q_2 can participate in at most one equality conditions.*
- *Projection: $\pi_{A_1,\ldots,A_n}$ for attributes $A_1, \ldots, A_n$, or $\pi_{\bar{A}}$ for a set $\bar{A}$ of attributes.*
- *Difference: $Q_1 - _\rho Q_2$, where $\mathcal{E}(Q_1) = \{A_1, \ldots, A_n\}$, $\mathcal{E}(Q_2) = \{B_1, \ldots, B_n\}$, and ρ is the following conjunction of correspondence conditions: $\rho = (A_1{\leftrightarrow}B_1) \wedge \cdots \wedge (A_n{\leftrightarrow}B_n)$; note that each attribute of Q_1 and Q_2 participates in exactly one $\leftrightarrow$ condition.*

In $Q_1 \bowtie_\rho Q_2$ or $Q_1 -_\rho Q_2$, we write $A \in \rho$ to express that ρ contains an equality condition on A, and $(A{=}A') \in \rho$ or $(A{\leftrightarrow}A') \in \rho$ to express that ρ contains the equality condition $A{=}A'$ or $A{\leftrightarrow}A'$, respectively. When no confusion arises, we choose a schema with suggestive unique attribute names like $R(A_r), S(A_s, B_s), T(B_t)$ and then write the queries $R \bowtie_{A_r=A_s} S$ and $(R \bowtie T) -_{A_r\leftrightarrow A_s \wedge B_t \leftrightarrow B_s} S$ more concisely as $R \bowtie S$ and $(R \bowtie T) - S$.

Transitively joined attributes. For every attribute A that appears in a $\mathrm{RA}^{\not\cup}$ query Q, we define the transitive join closure $[A]$ of A: $[A]$ contains all attributes that are transitively joined on A in Q. In computing $[A]$, we consider the difference operators as joins on all attributes of its operands, as introduced above. The transitive closures induces an equivalence relation on the set of attributes of a query, i.e., for any two attributes A, B we have $[A] = [B]$ or $[A] \cap [B] = \emptyset$. More formally:

Definition 4 (Transitively joined attributes)**.** *Let attr*(Q) *be the set of attributes of a query* Q*, and joins*$(Q) \subseteq$ *attr*$(Q) \times$ *attr*(Q) *the symmetric binary relation over attr*(Q) *such that* $(A, A) \in$ *joins*(Q) *for all attributes* A*, and* $(A, B) \in$ *joins*(Q) *if and only if* Q *contains an operator* $\bowtie_\rho$ *or* $-_\rho$ *with* $(A{=}B) \in \rho$*, or* $(B{=}A) \in \rho$*, or* $(A{\leftrightarrow}B) \in \rho$*, or* $(B{\leftrightarrow}A) \in \rho$*. Let* $joins^+(Q)$ *be the transitive closure of* $joins(Q)$*. Then* $A' \in [A]$ *if and only* $(A, A') \in joins^+(Q)$*.*

The notation of exported variables (cf. Definition 1) is generalised to variables exported modulo their transitive closure as follows:

Definition 5 (Exported variables)**.** *A query* Q exports $[A]$ *if there exists* $A' \in [A]$ *such that* $A' \in \mathcal{E}(Q)$*. Conversely,* Q does not export $[A]$ *if for all* $A' \in [A]$ *it holds that* $A' \notin \mathcal{E}(Q)$*.*

The notation Q^A denoting a query Q such that $A \in \mathcal{E}(Q)$ is extended to the case of transitively joined attributes as follows: $Q^{[A]}$ is a query that exports $[A]$, and $Q^{[\neg A]}$ is a query that does not export $[A]$; $Q^{[A][\neg B]}$ is a query that exports $[A]$, but not $[B]$.

We use the notation $\pi_{-A_1,\dots-A_n}(Q)$ as syntactic sugar for "projecting away $A_1, \dots, A_n$", i.e., $\pi_{-A_1,\dots,-A_n}(Q) = \pi_{\mathcal{E}(Q)-\{A_1,\dots,A_n\}}(Q)$. Similarly, for an attribute A, the operator $\pi_{[A]}$ is a shortcut for $\pi_{A'}$ for any $A' \in [A]$, and the operator $\pi_{-[A]}$ denotes $\pi_{-A_1,\dots,-A_n}$ where $[A] = \{A_1, \dots, A_n\}$.

Remark 3 (Pushing down equality conditions). Let Q be query and a sub-query $Q_1^{A'} \bowtie_\rho Q_2^{A'}$ of Q such that attributes A and A' are transitively joined in Q, i.e., $[A] = [A']$. Then even if $(A{=}A') \notin \rho$, the query Q is invariant under adding this condition to ρ. In other words, join conditions that are inferred by means of the transitive closure can be added the query and pushed down as far as possible without altering the query. □

3.1.2 The hierarchical property

We are now in the position to give the formal definition of the classes of hierarchical and non-hierarchical queries:

Definition 6 (Hierarchical $\text{RA}^{\not\emptyset}$ query)**.** *A* $RA^{\not\emptyset}$ *query* Q *is* hierarchical *if for every pair* A*,* B *of attributes of* Q *with* $A, B \notin \mathcal{E}(Q)$*, it holds that* Q *does not contain three distinct relation symbols* $R^{[A][\neg B]}$*,* $S^{[A][B]}$*,* $T^{[\neg A][B]}$*. Else,* Q *is called* non-hierarchical*.*

The membership problem

Problem IsHierarchical: Input: $\text{RA}^{\not\emptyset}$ query Q
Output: "Yes" if Q is hierarchical, "No" else

is equivalent to the corresponding problem for conjunctive queries [23], with the exception that IsHierarchical requires the computation of transitively joined attributes, while

```
IsHierarchical(Query Q)
    foreach pair of attributes A,B occurring in Q do
        HasA, HasB, HasAB ← false
        foreach relation symbol X in Q do
            if X exports [A] and not [B] then
                HasA ← true
            if X exports [A] and [B] then
                HasAB ← true
            if X exports [B] and not [A] then
                HasB ← true
        if HasA and HasB and HasAB then return "No"
    return "Yes"
```

Algorithm 2: The LOGSPACE-algorithm IsHierarchical decides the membership problem for the class of hierarchical $\mathrm{RA}^{\not\emptyset}$ queries.

they are explicitly given in the case of conjunctive queries. More specifically, the hierarchical property for $\mathrm{RA}^{\not\emptyset}$ queries needs to additionally account for the constraints imposed by difference operators. The hierarchical property for conjunctive queries can be decided in LOGSPACE [23] and deciding whether two attributes are transitively joined is in LOGSPACE (see below). This suggests that IsHierarchical is in LOGSPACE; we show that this is indeed the case by giving an explicit algorithm, cf. Algorithm 2. For each pair of variables A, B, the algorithm iterates over the relation symbols in Q and indicates by three Boolean flags whenever one of the relations $R^{[A][\neg B]}$, $S^{[A][B]}$, or $T^{[\neg A][B]}$ has been found. This amounts to checking whether two attributes are transitively joined in Q, i.e. whether $A' \in [A]$.

Proposition 6. *The decision problem IsHierarchical is in LOGSPACE.*

Proof. Algorithm IsHierarchical decides the problem IsHierarchical and is a LOGSPACE-algorithm: It uses a constant number of Boolean flags and a constant number of iterators over Q; the transitive join condition $A' \in [A]$ can be cast as the LOGSPACE-problem [73] of checking whether A and A' are connected in the undirected graph whose vertices are the attributes of Q and which has an edge between X and Y if and only if Q contains an operator $\bowtie_\rho$ with $(X{=}Y) \in \rho$ or an operator $-_\rho$ with $(X{\leftrightarrow}Y) \in \rho$. □

3.2 PTIME evaluation of hierarchical queries

The main result shown in this section is:

Lemma 7. *For any tuple-independent probabilistic database $\mathcal{D}$, the query evaluation problem for any hierarchical $RA^{\not\emptyset}$ query has PTIME data complexity.*

Proof. The lemma follows from the formal statements derived in this section as follows:

Q_{RA} is a hierarchical $\text{RA}^{\not\cup}$ query

$\Rightarrow$ Lemma 8 — Q_{RA} is equivalent to a special relational calculus query $Q_{RC} = [\![Q_{RA}]\!]$

$\Rightarrow$ Lemma 9 — The annotation $\Phi_{Q_{RC}(\mathcal{D})}$ has an OBDD of size $\mathcal{O}(|\mathcal{D}|)$

$\Rightarrow$ Corollary 10 — $P_{Q_{RC}(\mathcal{D})}$ can be computed in time $\mathcal{O}(|\mathcal{D}|)$ □

The fundamental technique in our PTIME algorithm is the conversion of a given hierarchical relational algebra query Q_{RA} into an equivalent disjunction Q_{RC} of disjunction-free relational calculus queries such that Q_{RA} and Q_{RC} give rise to the same annotation expression Φ when evaluated over a database $\mathcal{D}$. We show that for any tuple-independent database $\mathcal{D}$, Φ can be represented by an OBDD of size $\mathcal{O}(|\mathcal{D}| \cdot 2^{|Q_{RC}|})$ which in turn allows to compute $P_{Q_{RC}(\mathcal{D})}$ in time $\mathcal{O}(|\mathcal{D}| \cdot 2^{|Q_{RC}|})$; the asymptotic complexity of query evaluation is thus linear in the size of the database and in a factor that depends on the query only.

The reason for translating the relational algebra query into a relational calculus query is, that in contrast to the former the relational calculus expression allows for the unfolding of negation as per $\neg(Q_1 \wedge Q_2) \equiv \neg Q_1 \vee \neg Q_2$. Under the premise that Q_{RA} is hierarchical, we will show that the resulting $\text{RC}^{\exists,\neg}$ query Q_{RC} is *hierarchical*, viz:

Definition 7 (Hierarchical $\text{RC}^{\exists,\neg}$ query)**.** *A relational calculus query Q is* hierarchical *if for every sub-query $\exists_X(Q')$ in Q it holds that X is root in Q'.*

Recall that variable X is *root* in Q' if it appears in every relation symbol in Q', cf. Section 2.2.2, and that a query is *canonicalised* if each relation symbol occurs only with the same variable symbols, cf. Definition 2. In addition to being hierarchical, we show that quantifiers appear in only one nesting order in Q_{RC}:

Definition 8 ($\exists$-consistent $\text{RC}^{\exists,\neg}$ query)**.** *Let Q be a canonicalised, hierarchical $RC^{\exists,\neg}$-query. Q is* $\exists$-consistent *if there exists a total order $>_\exists$ of the variable symbols in Q such that $X >_\exists Y$ implies that Q does not contain a sub-query of the form $\exists_Y Q'(\exists_X)$.*

Intuitively, hierarchical and $\exists$-consistent queries have the same join order and their annotations can be compiled into OBDDs with the same variable order. Let us first illustrate these concepts with an example and then prove formally (1) that a hierarchical $\text{RA}^{\not\cup}$ query can be translated to a hierarchical, $\exists$-consistent $\text{RC}^{\exists,\neg}$ query (Section 3.2.1), and (2) the annotation of such an $\text{RC}^{\exists,\neg}$ query Q_{RC} can be represented in an OBDD of size $\mathcal{O}(|\mathcal{D}| \cdot 2^{|Q_{RC}|})$ for any tuple-independent database $\mathcal{D}$ (Section 3.2.2).

Example 11. Consider the following three Boolean $\text{RC}^{\exists,\neg}$ queries:

$$Q_1 = \exists_X R(X)\exists_Y S(X,Y) \qquad Q_2 = \exists_Y T(Y)\exists_X S(X,Y) \qquad Q_3 = \exists_X U(X)\exists_Y S(X,Y)$$

Note that all three queries are hierarchical since in each occurrence of $\exists_X$ and $\exists_Y$, X and Y, respectively, are root variables. Let us evaluate the queries over the database $\mathcal{D}$, viz:

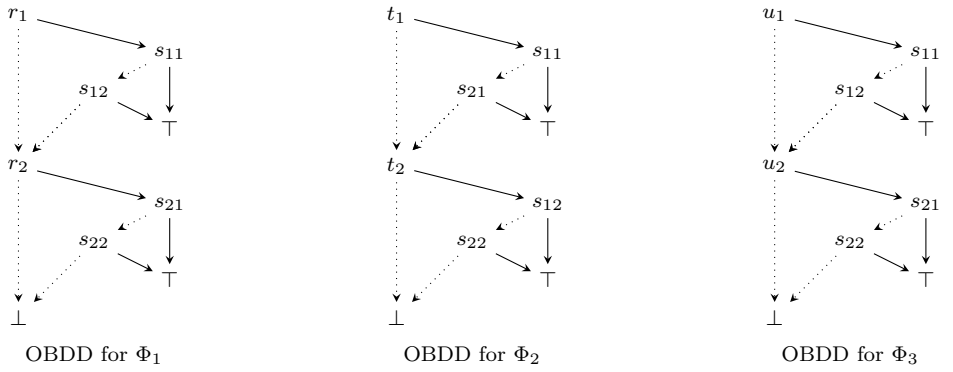


Figure 3.6: Three OBDDs with compatible variable orders, cf. Example 11.

R		S			T		U	
X	$\mathbf{\Phi}$	X	Y	$\mathbf{\Phi}$	Y	$\mathbf{\Phi}$	X	$\mathbf{\Phi}$
1	$\mathbf{r_1}$	1	1	$\mathbf{s_{11}}$	1	$\mathbf{t_1}$	1	$\mathbf{u_1}$
2	$\mathbf{r_2}$	1	2	$\mathbf{s_{12}}$	2	$\mathbf{t_2}$	2	$\mathbf{u_2}$
		2	1	$\mathbf{s_{21}}$				
		2	2	$\mathbf{s_{22}}$				

The annotations Φ_1, Φ_2, Φ_3 of Q_1, Q_2, Q_3 evaluated over $\mathcal{D}$ are

$$\Phi_1 = r_1(s_{11} \vee s_{12}) \vee r_2(s_{21} \vee s_{22})$$
$$\Phi_2 = t_1(s_{11} \vee s_{21}) \vee t_2(s_{12} \vee s_{22})$$
$$\Phi_3 = u_1(s_{11} \vee s_{12}) \vee u_2(s_{21} \vee s_{22})$$

and can be represented by the three OBDDs shown in Figure 3.6. The variable orders Π_1, Π_2, Π_3 in the three OBDDs are:

$$\Pi_1 : r_1, s_{11}, s_{12}, r_2, s_{21}, s_{22}$$
$$\Pi_2 : t_1, s_{11}, s_{21}, t_2, s_{12}, s_{22}$$
$$\Pi_3 : u_1, s_{11}, s_{12}, u_2, s_{21}, s_{22}$$

Now consider the query $Q_{13} = Q_1 \vee Q_3$; this query is canonicalised and hierarchical. Moreover, Q_{13} is $\exists$-consistent because the quantifiers only occur in order $\exists_X Q'(\exists_Y)$. Note that the variable orders Π_1 and Π_3 are compatible in the sense that they can be combined to an order Π_{13} over all variables in Φ_1 and Φ_3:

$$\Pi_{13} : r_1, u_1, s_{11}, s_{12}, r_2, u_2, s_{21}, s_{22}$$

In the light of Lemma 3, the OBDDs of Φ_1 and Φ_3 can be combined to yield an OBDD of width 4 for the annotation $\Phi_1 \vee \Phi_3$ of query Q_{13}.

Conversely, the query $Q_{12} = Q_1 \vee Q_2$ is hierarchical and canonicalised, but not $\exists$-

consistent. Consequently, the variable orders Π_1 and Π_2 are not compatible. In fact, query Q_{12} is #P-hard [25] and there is in general no OBDD of asymptotic size polynomial in $|\mathcal{D}|$ for the annotation of Q_{12} on $\mathcal{D}$ [48]. □

The following two sub-sections formalise the intuition illustrated by the example.

3.2.1 Translation from relational algebra to relational calculus

At the core of the PTIME algorithm for hierarchical queries is a recursive compilation procedure that transforms a given relational algebra query Q_{RA} into an equivalent relational calculus query Q_{RC}; this translation function $[\![\cdot]\!]$ is given in Algorithm 3.

The translation is equivalent to the standard translation from relational algebra to relational calculus [2], with the addition that after each recursive translation step, we "flatten" the resulting $\mathrm{RC}^{\exists,\neg}$query by (i) pushing $\exists$-operators as deep as possible ($\textsc{Push}^{\exists_X}$), by (ii) pushing $\neg$ operators as deep as possible ($\textsc{Push}^{\neg}$), and by (iii) expanding conjunctions of disjunctions into disjunctions of conjunctions (Expand). These transformations are standard, equivalence-preserving rewritings of the original $\mathrm{RC}^{\exists,\neg}$ expression. Moreover, for any database, the annotations of the query results of a relational algebra query Q_{RA} and of its relational calculus translation $[\![Q_{RA}]\!]$ are equivalent propositional formulas. Due to the eager pushing of negation and expansion of products to sums, the translation incurs a non-elementary blow-up of the query size; Section 3.2.3 discusses a lower bound for the worst-case blow-up.

Example 12. Example 9 from the introduction to this chapter demonstrates how a relational algebra query is translated to a relational calculus query, and that this query gives rise to the same result annotations as the original one. □

A very useful property of queries obtained from $[\![\cdot]\!]$-translating a non-repeating $\mathrm{RA}^{\not\emptyset}$ query is that by construction each occurrence of any relation symbol appears with the same variable identifiers, i.e. any such translated query is *canonicalised*, cf. Definition 2.

Remark 4. For a $\mathrm{RA}^{\not\emptyset}$ query Q_{RA} and its relational calculus (canonicalised) translation $Q_{RC} = [\![Q_{RA}]\!]$, we have the following obvious relationship between attribute names in Q_{RA} and variable names in Q_{RC}; let R be a relational symbol and X a variable in Q_{RA}. Then there exists a variable $X' \in [X]$ such that R occurs with X' in Q_{RC} if and only if R exports $[X]$ in Q_{RA}. We also say that Q_{RC} is *canonicalised with respect to* Q_{RA} to emphasise this relationship. □

The $\mathrm{RC}^{\exists,\neg}$ query obtained from the translation satisfies a set of properties as detailed in the following lemma:

Lemma 8. *Let Q_{RA} be a hierarchical $RA^{\not\emptyset}$ query. There exists an equivalent $RC^{\exists,\neg}$-query Q_{RC} with the following properties:*

$$
\begin{aligned}
[\![R]\!] &= \{sch(R) \mid R(sch(R))\} \\
[\![\pi_{-X}(Q)]\!] &= \{H_{[\![Q]\!]} \setminus \{x\} \mid \textsc{Push}^{\exists_X}(F_{[\![Q]\!]})\} \\
[\![Q_1 \bowtie_{\wedge_i(X_i=Y_i)} Q_2]\!] &= \{H_{[\![Q_1]\!]} \mid \textsc{Expand}(F_{[\![Q_1]\!]} \wedge F_{[\![Q_2]\!]}[\forall i : X_i/Y_i])\} \\
[\![Q_1 -_{\wedge_i(X_i \leftrightarrow Y_i)} Q_2]\!] &= \{H_{[\![Q_1]\!]} \mid \textsc{Expand}(F_{[\![Q_1]\!]} \wedge \textsc{Push}^{\neg}(F_{[\![Q_2]\!]}[\forall i : X_i/Y_i]))\}
\end{aligned}
$$

$\textsc{Push}^{\exists_X}$(Query Q)
- **if** X *does not occur in* Q **then**
  - **return** Q
- **if** $Q = \exists_Y(Q')$ **then**
  - **return** $\exists_Y(\textsc{Push}^{\exists_X}(Q'))$
- **else if** $Q = \bigvee_i Q_i$ **then**
  - **return** $\bigvee_i \textsc{Push}^{\exists_X}(Q_i)$
- **else if** $Q = \neg Q'$ *such that* X *is a root variable in* Q' **then**
  - **return** $\exists_X(\neg Q')$
- **else if** $Q = (\bigwedge_i Q_i) \wedge (\bigwedge_j Q'_j)$ *such that* X *is a root variable in all* Q_i *and* X *does not appear in any* Q'_j **then**
  - ▷ Note that the second conjunct may be empty.
  - **return** $\exists_X(\bigwedge_i Q_i) \wedge (\bigwedge_j Q'_j)$
- **else**
  - **fail**

$\textsc{Push}^{\neg}$(Query Q)
- **if** $Q = \exists_X(Q')$ **then**
  - **return** $\neg\exists_X(Q')$
- **else if** $Q = \neg Q'$ **then**
  - **return** Q'
- **else if** $Q = \bigvee_i Q_i$ **then**
  - **return** $\bigwedge_i \textsc{Push}^{\neg}(Q_i)$
- **else if** $Q = \bigwedge_i Q_i$ **then**
  - **return** $\bigvee_i \textsc{Push}^{\neg}(Q_i)$
- **else if** $Q = R(\dots)$ **then**
  - **return** $\neg R(\dots)$

$\textsc{Expand}$(Query Q)
- **if** $Q = \exists_X Q'$ *or* $Q = \neg Q'$ *or* $Q = R(\dots)$ **then return** Q
- **else if** $Q = (\bigvee_i Q_i) \wedge (\bigvee_j Q'_j)$ **then return** $\bigvee_{i,j} \textsc{Expand}(Q_i) \wedge \textsc{Expand}(Q'_j)$
- **else if** $Q = (\bigwedge_i Q_i \vee \bigwedge_j Q'_j)$ **then return** $(\bigwedge_i \textsc{Expand}(Q_i)) \vee (\bigwedge_j \textsc{Expand}(Q'_j))$

Algorithm 3: Translation function $[\![\cdot]\!]$ from relational algebra to relational calculus.

(a) Q_{RC} *is canonicalised with respect to* Q_{RA}*. Furthermore, if a variable* X *occurs in the scope of a quantifier* $\exists_X$*, then every occurrence of* X *is in the scope of a quantifier* $\exists_X$*.*

(b) Q_{RC} *is a disjunction of disjunction-free* $RC^{\exists,\neg}$*-queries.*

(c) For every variable X *in* Q_{RC}*,* Q_{RC} *has no sub-query of the form*[4] $\exists_X(Q) \wedge Q'(\exists_X)$*.*

(d) Q_{RC} *is hierarchical.*

(e) Q_{RC} *is* $\exists$*-consistent.*

Note that condition (c) permits sub-queries of the form $\neg\exists_X(Q) \wedge \neg\exists_X(Q')$ or of the form $\exists_X(Q) \vee \exists_X(Q')$ and disallows $\exists_X(Q) \wedge \exists_X(Q')$, $\exists_X(Q) \wedge \neg\exists_X(Q')$, $\exists_X(Q) \wedge \neg\exists_Y(Q'' \wedge \neg\exists_X(Q'''))$, etc.

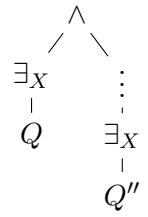
Proof. We show that the translation $[\![Q_{RA}]\!]$ returns an $\mathrm{RC}^{\exists,\neg}$query Q_{RC} that satisfies the properties of the lemma. The proof is direct for properties (a), (b), (c), (e), and inductive on the structure of the query Q_{RA} for property (d).

[4] We write $Q(\exists_X)$ to denote a query Q in which $\exists_X$ occurs.

Property (a)**.** According to Definition 2, attributes that are transitively joined in Q_{RA} must translate to the same variable name in Q_{RC}. This is indeed achieved in $[\![Q_1 \bowtie Q_2]\!]$ and $[\![Q_1 - Q_2]\!]$ by renaming all attributes in Q_2 to the corresponding variable name in Q_1. When a quantifier $\exists_X$ is introduced via $[\![\pi_{-X}(Q)]\!]$, then clearly all variables X in Q are in the scope of $\exists_X$; furthermore, the $\textsc{Push}^{\exists_X}$ procedure never pushes $\exists_X$ past a relation symbol containing X and hence every occurrence of X is in the scope of a quantifier $\exists_X$.

Property (b)**.** The combination of pushing down negation ($\textsc{Push}^{\neg}$) and expanding conjunctions of disjunctions (Expand) is standard and yields an expression with disjunctions only at the top level.

Property (c)**.** Let us assume that Q_{RC} contains a sub-query $\exists_X(Q) \wedge Q'(\exists_X)$. Then the parse-tree of Q_{RC} contains a sub-tree of the following form:



Let us consider what sequence of transformations could have led to this sub-query. Firstly, since Q_{RA} is non-repeating and since Q_{RC} is canonicalised, it is clear that the two $\exists_X$ operators originate from a single $\exists_X$ operator that was applied to the entire sub-query as a result of the translation of $[\![\pi_{-X}(Q)]\!]$. The $\exists_X$ operator must subsequently have been "duplicated" as a result of distributing it through disjunctions in $\textsc{Push}^{\exists_X}$. Since $\textsc{Push}^{\exists_X}$ does not distribute $\exists_X$ over a conjunction, the conjunction operator on top of this sub-query must have been flipped by a negation operator into a disjunction through which $\exists_X$ was distributed; in order to get back to a conjunction operator, there must have been another odd number of $\neg$ operators following $\exists_X$. This is a contradiction to the assumption that the query has the form $\exists_X(Q) \wedge Q'$ with $\exists_X$ appearing positively.

Property (d)**.** We show this property by induction on the different cases in $[\![\cdot]\!]$. The base case is a query $Q_{RA} = R$ for a single relation symbol. Since $[\![R]\!] = \{sch(R) \mid R(sch(R))\}$ does not contain $\exists$ operators, $[\![R]\!]$ is vacuously hierarchical. For the inductive step, the induction hypothesis is that the $\mathrm{RC}^{\exists,\neg}$queries obtained recursively by the algorithm are hierarchical; we show that any step taken by the translation preserves the hierarchical property. Clearly, under this assumption the only case in algorithm $[\![\cdot]\!]$ that may introduce a non-root $\exists_X$ operator is the rule for $[\![\pi_{-X}(Q_{RA})]\!]$; by the induction hypothesis, let $Q_{RC} = [\![Q_{RA}]\!]$ be the hierarchical $\mathrm{RC}^{\exists,\neg}$query resulting from the translation of Q_{RA}; Q_{RC} is canonicalised with respect to Q_{RA} and a disjunction of disjunction-free $\mathrm{RC}^{\exists,\neg}$queries, and does not contain a sub-query of the form $\exists(Q) \wedge Q'(\exists_X)$. We show that if $\textsc{Push}^{\exists_X}(Q_{RC})$ fails

then Q_{RA} is non-hierarchical; conversely, for any hierarchical query Q_{RA}, $\textsc{Push}^{\exists_X}$ always succeeds in pushing $\exists_X$ so that X becomes a root variable in the scope of $\exists_X$.

First, the rule for $Q = \bigvee_i Q_i$ will push $\exists_X$ through the top-level disjunction and apply $\exists_X$ to each of the disjunction-free queries Q_i. Each of the Q_i is a conjunction $\bigwedge_i q_i$ where each q_i has the form $\exists_Y(q)$, $\neg\exists_Y(q)$, R, or $\neg R$. We analyse different cases separately:

1. Variable X occurs in exactly one of the q_i:

   (i) If $q_i = \exists_Y(q)$, then we recursively apply this argument to $\exists_Y(\textsc{Push}^{\exists_X}(q))$. Note that by induction hypothesis, Y is root in q and this remains true after commuting $\exists_X$ and $\exists_Y$.

   (ii) If $q_i = R$ or $q_i = \neg R$, then $\textsc{Push}^{\exists_X}$ terminates by returning $\exists_X R$ or $\exists_X \neg R$, respectively; in both cases, X is root in the scope of $\exists_X$.

   (iii) If $q_i = \neg\exists_Y(q)$ and X is root in q, then the rule for $Q = \neg Q'$ applies and we return $\exists_X \neg\exists_Y(q)$, in which X is root. If X is not root in q, then q must contain two relations symbols such that by induction hypothesis Y occurs in both relation symbols and X occurs in one but not the other. With loss of generality, let us assume that these relations are $T(Y)$ and $S(X, Y)$. Since $[\![\cdot]\!]$ never commutes $\neg$ and $\exists$, this implies that Q_{RA} contains a sub-query of the form $\pi_{-X}(Q_1 - \pi_{-Y} Q_2(S(X, Y), T(Y)))$; in order for this to be a valid query, Q_1 must export $[X]$; furthermore, since Q_{RC} is canonicalised, no variable in $[Y]$ can occur in any relation symbol in Q_1. This means that Q_{RA} contains[5] three relation symbols $R^{[X][\neg Y]}$, $S^{[X][Y]}$, $T^{[\neg X][Y]}$ and is thus non-hierarchical.

2. Variable X occurs in more than one of the q_i:

   (i) If X is root in the conjuncts in which it appears, then case $Q = \left(\bigwedge_i Q_i\right) \wedge \left(\bigwedge_j Q'_j\right)$ applies in $\textsc{Push}^{\exists_X}$ and X is root in the scope of $\exists_X$.

   (ii) Otherwise, there is a conjunct q_j in which X occurs and a conjunct q_i that contains two relation symbols S, T such that X occurs in one of them but not in the other. Thus q_i has the form $\exists_Y Q(S(X, Y), T(Y))$ or $\neg\exists_Y Q(S(X, Y), T(Y))$ where by induction hypothesis Y occurs in both S and T. The latter case is equivalent to the above case (1iii). In the former case, if q_j is a single relation in which X occurs (or the negation of a single relation in which X occurs) without a quantifier, then this relation cannot have variable Y because all occurrences of Y are in the scope of a quantifier, cf. Property (a). Thus R, S, T render Q_{RA} non-hierarchical. On the other hand, q_j may have the form $\exists_Z(Q)$ where Z is a variable different from Y and Q does not contain a quantifier $\exists_Y$, cf. Property (a). Then the relation in q_j in which X occurs together with S and T render Q_{RA} non-hierarchical.

[5] Compare also Remark 4.

Property (e)**.** Assume Q_{RC} contains two sub-queries $Q_a = \exists_X Q(\exists_Y))$ and $Q_b = \exists_Y Q(\exists_X)$ in which the order of the quantifiers $\exists_X$ and $\exists_Y$ is inconsistent. If one of the two sub-queries has the form $\exists_X \exists_Y (Q')$, then the order of the quantifiers may be switched to obtain $\exists$-consistent queries; conversely, there are two structurally different cases in which the order of the quantifiers can not be switched: (i) $\exists_X Q(\neg\exists_Y)$, and (ii) $\exists_X[Q(X) \wedge Q'(\exists_Y)]$. We consider the four combinations of these cases for the sub-queries Q_a and Q_b:

1. Q_a and Q_b are of type (i), i.e., $Q_a = \exists_X Q_1(\neg\exists_Y)$ and $Q_b = \exists_Y Q_2(\neg\exists_X)$. Then, since Q_{RC} is canonicalised and since the order of $\exists$ and $\neg$ is never swapped by $[\![\cdot]\!]$, the structure of Q_a implies that Q_{RC} has a sub-query of the form $\pi_{-X}(Q' - \pi_{-Y}(Q''))$ and the structure of Q_b implies that Q_{RC} has a sub-query of the form $\pi_{-Y}(Q' - \pi_{-X}(Q''))$; this is a contradiction.

2. Q_a and Q_b are of type (ii), i.e., $Q_a = \exists_X[Q_1(X) \wedge Q_1'(\exists_Y)]$ and $Q_b = \exists_Y[Q_2(Y) \wedge Q_2'(\exists_X)]$; then Q_1 contains a relation $R^{[X][\neg Y]}$, Q_2 contains a relation $T^{[\neg X][Y]}$, and Q_1' contains a relation $S^{[X][Y]}$; thus Q_{RA} is non-hierarchical and this is a contradiction to the initial assumption.

3. q_a is of type (i) and q_b is of type (ii), i.e., $Q_a = \exists_X Q_1(\neg\exists_Y)$ and $Q_b = \exists_Y[Q_2(Y) \wedge Q_2'(\exists_X)]$. Since in Q_b there is no negation operator between $\exists_Y$ and $\exists_X$, it follows that in Q there is no difference operator between π_{-X} and π_{-Y}. Conversely, the negation operator between $\exists_X$ and $\exists_Y$ in Q_a requires a difference operator between π_{-X} and π_{-Y} in Q_{RA}; this is a contradiction.

4. q_a is of type (ii) and q_b is of type (i); this is symmetric to the previous case. □

3.2.2 Construction of polynomial-size OBDDs for relational calculus queries

The last step in the PTIME algorithm is to show that the annotation $\Phi_{Q_{RC}(\mathcal{D})}$ defined by the $\mathrm{RC}^{\exists,\neg}$ query Q_{RC} obtained from Q_{RA} as per Lemma 8 allows for an OBDD with size linear in $|\mathcal{D}|$ and thus for PTIME probability computation. The proof is for Boolean queries in order to avoid syntactic clutter; the general case is analogous.

The OBDD construction is based on a total order Π on the tuples in $\mathcal{D}$ such that $\Phi_{Q_{RC}(\mathcal{D})}$ has a Π-OBDD of size linear in $|\mathcal{D}|$; the order Π can be derived from the structure of Q_{RC}. Let us first exemplify the construction of this order and then define it formally in the proof of Lemma 9.

Example 13. Consider the query $Q = \exists_X \big[R(X) \wedge \exists_Y S(X,Y)T(X,Y)\big]$. Since X is a root variable, the OBDDs for different values of X are independent and can simply be concatenated. For each fixed value $A \in \mathrm{ADom}(X)$, we have to construct the OBDD of the query $R(A) \wedge \exists_Y S(A,Y)T(A,Y)$; a good variable order for this OBDD is to first decide on the annotation of $R(A)$ and then on all annotations of $S(A,B)$ and of $T(A,B)$ for all values

$B \in \mathrm{ADom}(Y)$. If we write $R(1)$ for the annotation of tuple (1) in R, and similarly for S and T, then the overall variable order obtained is

$$
\begin{aligned}
&R(1), S(1,1), T(1,1), S(1,2), T(1,2), S(1,3), T(1,3) \ldots, && \text{(tuples with } X = 1) \\
&R(2), S(2,1), T(2,1), S(2,2), T(2,2), S(2,3), T(2,3) \ldots, && \text{(tuples with } X = 2) \\
&R(3), S(3,1), T(3,1), S(3,2), T(3,2), S(3,3), T(3,3) \ldots, && \text{(tuples with } X = 3) \\
&\ldots
\end{aligned}
$$

Note that the annotations are ordered in lexicographically ascending order: We first decide on all annotations with $X = 1$, then on all annotations with $X = 2$, etc. For all annotations with $X = 1$, we first decide on those with $Y = 1$, then those with $Y = 2$, etc. This variable order leads to a compact OBDD because the variable symbols X, Y in the relations R, S, T are ordered compatibly to the nesting order of the quantifiers $\exists_X$ and $\exists_Y$. □

The central result regarding the OBDD construction for hierarchical $\mathrm{RA}^{\not\emptyset}$ queries is:

Lemma 9. *Let Q_{RC} be a $RC^{\exists,\neg}$ query satisfying the properties of Lemma 8. The annotation $\Phi_{Q_{RC}(\mathcal{D})}$ of Q_{RC} on a tuple-independent database $\mathcal{D}$ can be represented by an OBDD of size $\mathcal{O}(|\mathcal{D}| \cdot 2^{|Q_{RC}|})$, where $|Q_{RC}|$ is the number of relation symbols in Q_{RC}.*

Proof. The proof uses the following conventions. The relation symbols in Q_{RC} are $R_1, \ldots, R_n$ and the variables are $X_1, \ldots, X_m$. In the context of a query Q_{RC} and a database $\mathcal{D}$, the active domain of variable X_i is denoted by $\mathrm{ADom}(X_i)$ and is — intuitively — the set of values that X_i takes in $\mathcal{D}$; more formally,

$$\mathrm{ADom}(X_i) = \bigcup_{R_j(\bar{X}) \text{ is relation symbol in } Q_{RC}} \{c \mid R_j(\bar{X})[c/X_i] \text{ is satisfiable by } \mathcal{D}\}$$

Furthermore, we assume without loss of generality that the active domain of the database is totally ordered. The annotation of tuple $\bar{A}$ of relation R_i in $\mathcal{D}$ is denoted by $R_i(\bar{A})$; for example, the annotation of tuple (a, b) in relation R_1 is $R_1(a, b)$. Without loss of generality, let the query variables in Q_{RC} be labelled $X_1, \ldots, X_m$ such that $X_i >_\exists X_j \Leftrightarrow i < j$ with respect to the nesting order $>_\exists$ defined by the $\exists$-consistency of Q_{RC}. I.e., $i < j$ allows for the quantifier nesting $\exists_{X_i} Q(\exists_{X_j})$, but not $\exists_{X_j} Q(\exists_{X_i})$. Since by Lemma 8 Q_{RC} is canonicalised and $\exists$-consistent, we can assume without loss of generality[6] that in each relation symbol R the query variables occur in $>_\exists$ order. For example, Q_{RC} may contain $R(X_1, X_5, X_7)$, but not $R(X_7, X_1, X_5)$.

We define a total order Π on the annotations of the tuples in $\mathcal{D}$ as follows. First, associate with every annotation $R(\bar{A})$ the string $\mathrm{string}(R(\bar{A})) = \bar{A}R$; for example, annotation $R_2(A_7, B_2, C_7)$ where A_7, B_2, and C_7 are constants in $\mathcal{D}$ is associated with string $A_7 B_2 C_7 R_2$.

[6] More precisely, we can always relabel the query and database schema such that the query variables occur only in $>_\exists$-order.

The order Π is then defined as

$$R(\bar{A}) <_\Pi R'(\bar{A}') \quad \Leftrightarrow \quad \text{string}(R(\bar{A})) <_{\text{lex}} \text{string}(R(\bar{A}')) \tag{3.1}$$

where $<_{\text{lex}}$ is the lexicographic order on strings as defined by the total order of the active domain of the database and the order $R_1 <_{\text{lex}} R_2 <_{\text{lex}} \cdots <_{\text{lex}} R_n$. Note that Π is uniquely defined by the order of the relation symbols and the order on the active domain of $\mathcal{D}$. However, different orders on the former and the latter give rise to different orders Π.

We show by structural induction over the construction of $\Phi_{Q_{RC}}$ via Equation (2.10) that $\Phi_{Q_{RC}}$ has a Π-OBDD of width $2^{|Q_{RC}|}$ where $|Q_{RC}|$ denotes the number of relation symbols in Q_{RC}:

- The base case is a relation symbol $R(\bar{A})$ which corresponds to a trivial Π-OBDD with one variable $R(\bar{A})$ and width 2.
- If $Q_{RC} = Q_1 \wedge Q_2$ or $Q_{RC} = Q_1 \vee Q_2$, then by induction hypothesis Φ_{Q_1} and Φ_{Q_2} have Π-OBDDs of width $2^{|Q_1|}$ and $2^{|Q_2|}$, respectively. Then by Lemma 3, $\Phi_{Q_{RC}}$ has a Π-OBDD of size $2^{|Q_1|} \cdot 2^{|Q_2|} = 2^{|Q_{RC}|}$.
- If $Q_{RC} = \neg Q$, then by induction hypothesis Q has a Π-OBDD of width $2^{|Q|}$. Swapping the $\top$ and $\bot$ nodes in this OBDD yields the required Π-OBDD for Q_{RC}.
- If $Q_{RC} = \exists_{X_i} Q$, then for every $A_l \in \text{ADom}(X_i)$ the formulas $\Phi_l = \Phi_{Q[A_l/X_i]}$ are over disjoints sets of variables because Q_{RC} is hierarchical by Lemma 8 and hence X_i is root in Q. Moreover, each Φ_l has a Π-OBDD of width $2^{|Q|}$ by induction hypothesis. Without loss of generality, let $\text{ADom}(X_i) = \{A_1, \ldots, A_h\}$ such that $A_k <_{\text{lex}} A_l$ if and only if $k < l$. By Equation (2.10), the annotation $\Phi_{Q_{RC}}$ is the disjunction $\Phi_{Q_{RC}} = \bigvee_{A_l \in \text{ADom}(X_i)} \Phi_l$. Since the formulas Φ_l are over disjoints sets of variables for distinct values of l, an OBDD for their disjunction is obtained by their concatenation in which the $\bot$ nodes of the OBDD for Φ_l are removed and their incoming edges are connected to the root node of the OBDD for Φ_{l+1}.

  It remains to be shown that this construction yields an OBDD with respect to order Π. First, note that the OBDD for each Φ_l is over order Π by induction hypothesis; it remains to be shown that for any two annotations $R(\bar{A}_k)$ in Φ_k and $R'(\bar{A}_l)$ in Φ_l where $k < l$ it holds that $R'(\bar{A}_k) <_\Pi R'(\bar{A}_l)$; by the definition of $<_\Pi$, this is equivalent to showing $\bar{A}_k R <_{\text{lex}} \bar{A}_l R'$. By construction (cf. Equation (2.10)), the variables X_j with $j < i$ are set to the same constants. Thus the strings $\bar{A}_k$ and $\bar{A}_l$ are identical in the first $i - 1$ places. The lexicographic order $\bar{A}_k$ and $\bar{A}_l$ — and hence the Π-order of $R(\bar{A}_k)$ in Φ_k and of $R'(\bar{A}_l)$ in Φ_l — is determined by the values of X_i in $\bar{A}_k$ and in $\bar{A}_l$; this value is A_l in $\bar{A}_l$ and A_k in $\bar{A}_k$. Since we concatenate the OBDDs in the order $\Phi_1 \to \Phi_2 \to \cdots \to \Phi_h$ and since $A_1 <_{\text{lex}} \cdots <_{\text{lex}} A_h$ it follows that $\bar{A}_k <_{\text{lex}} \bar{A}_l$ and thus $R(\bar{A}_k) <_\Pi R'(\bar{A}_l)$.

Finally, the constructed OBDD has width $2^{|Q_{RC}|} = 2^{|Q|}$, because the concatenation leaves the width unchanged. □

Given the OBDD construction of Lemma 9, we can apply Lemma 4 and find that the probability of the thus constructed OBDD can be evaluated in time linear in the size of the database and a factor that depends only on the query:

Corollary 10 (Lemmata 4, 9)**.** *Let Q_{RC} be a $RC^{\exists,\neg}$ query satisfying the properties of Lemma 8. For any tuple-independent database $\mathcal{D}$, the probability $P_{Q_{RC}(\mathcal{D})}$ can be evaluated in time $\mathcal{O}(|\mathcal{D}| \cdot 2^{|Q_{RC}|})$, where $|Q_{RC}|$ is the number of relation symbols in Q_{RC}.*

3.2.3 The complexity of the PTIME algorithm in terms of the query size

The combined complexity of $\mathrm{RA}^{\not\emptyset}$ queries is PSPACE-complete (cf. Section 2.4), and Corollary 10 shows that the data complexity of hierarchical $\mathrm{RA}^{\not\emptyset}$ queries is in PTIME. The discussion in this subsection shows that the PTIME algorithm presented above is not in PSPACE for combined complexity: We show that there is a class of queries for which the width of the OBBD constructed in Lemma 9 is a non-elementary function of the size of the query. The reason for this blow-up is the eager pushing of negation and expansion of conjunctions in the translation $[\![\cdot]\!]$ (cf. Algorithm 3) from the input relational algebra query to a relational calculus query.

Let us start by analysing this algorithm. As discussed before, the translation eagerly expands the $\mathrm{RA}^{\not\emptyset}$ query into a disjunction of disjunction-free $\mathrm{RC}^{\exists,\neg}$ queries; let $c(Q)$ denote the number of such conjuncts in Q and $a(Q)$ the maximum arity of such conjuncts; they can be expressed as follows:

$$
\begin{aligned}
Q = Q_1 \bowtie Q_2 : \quad & a(Q) = a(Q_1) + a(Q_2) \\
& c(Q) = c(Q_1) \cdot c(Q_2) \\
Q = Q_1 - Q_2 : \quad & a(Q) = a(Q_1) + c(Q_2) \\
& c(Q) = c(Q_1) \cdot a(Q_2)^{c(Q_2)}
\end{aligned}
$$

The behaviour of c and a is dominated by the exponentiation in the rule for $Q = Q_1 - Q_2$. The following recursively defined family of $\mathrm{RA}^{\not\emptyset}$ queries over the schema $R_0(A)$, $R'_0(A)$, $R_1(A)$, …, $R_n(A)$ maximises the number of difference operators for a given number of relation symbols and thus constitutes a worst-case query for the analysis of the asymptotic behaviour of c and a:

$$
\begin{aligned}
Q_1 &= R_1 - (R_0 \bowtie R'_0) \\
Q_n &= R_n - R_{n-1}
\end{aligned}
$$

Using the above recursions, we obtain for the size of the $\mathrm{RC}^{\exists,\neg}$ expansion of Q_n:

$$c(Q_1) = 2$$
$$c(Q_n) = (1 + c(Q_{n-2}))^{c(Q_{n-1})}$$

This function has a lower bound

$$c(Q_n) \leq 2^{\cdot^{\cdot^{\cdot^{2}}}}$$

where the number of stacked exponentials is n. Recall that $c(Q_n)$ denotes the number of conjuncts in the translated $\mathrm{RC}^{\exists,\neg}$ query and is a lower bound on the number of its relation symbols. $2^{c(Q_n)}$ is thus a lower bound on the width $2^{|Q_{RC}|}$ of the OBDD for Φ_{Q_n} in Lemma 9.

The above discussion suggests that our PTIME algorithm may not be optimal in terms of combined complexity; the exact trade-off between data and query complexity is left open by the results obtained in this chapter. The goal of this chapter — to separate tractable from intractable queries with respect to data complexity — is nevertheless achieved by Theorem 5.

3.3 Data-preserving fillings

The goal of this section is to develop a uniform formalism for determining a database $\mathcal{D}$ for the relations occurring in a $\mathrm{RA}^{\cup\!\!\!\!\cdot}$ query Q, such that the evaluation of Q on $\mathcal{D}$ preserves the tuples and annotations of a distinguished relation of the query; Lemmata 11 and 12 below formalise this intuition. The formalism is used in Section 3.4 to construct hardness reductions for non-hierarchical queries.

In order to avoid syntactic clutter in this section, it will be advantageous to blur the distinction between a query Q and the relation obtained by evaluating Q on a database $\mathcal{D}$; the relation symbols in Q directly represent relations, and the query operators combine the relations represented by their sub-queries. We say that we can *fill a relation (symbol) X of Q with a relation R* to denote the fact that there exists a database $\mathcal{D}$ such that $X(\mathcal{D}) = R$. In this sense, the notions of a *query* and a *relation* are the same; moreover, we will introduce classes of queries by means of properties of the relations they represent when evaluated on the database defined implicitly by the filling of their relations.

In the following, we assume two finite sets of constants, $\mathbf{A}$ and $\mathbf{B}$, and denote by ■ a constant distinct from those in $\mathbf{A}$ and $\mathbf{B}$. In this section, the projection operator π_A^Φ is used to symbolise the projection on attribute A and the annotation column Φ; in contrast, π_A selects only column A, neglecting the annotations of tuples. The notation $(a_1, \ldots, a_n | \Phi(a_1, \ldots, a_n))$ denotes a tuple $(a_1, \ldots, a_n)$ annotated with formula $\Phi(a_1, \ldots, a_n)$.

Preserving the data of one attribute

We commence by analysing queries with one distinct attribute A. Let Φ be a total function on $\mathbf{A}$. A relation Q is *A-reducible to* $(\mathbf{A}, \Phi)$, if the $[A]$-attributes of Q are filled with all values from $\mathbf{A}$, all non-$[A]$-attributes are filled with ■, and the annotation of a tuple identified by $a \in \mathbf{A}$ is $\Phi(a)$:

$$\pi^{\Phi}_{[A]}(Q) = \{(a|\Phi(a)) \mid a \in \mathbf{A}\}$$
$$\pi_C(Q) = \{(■)\} \qquad \text{for any attribute } C \text{ with } C \notin [A].$$

We write $\mathrm{red}_A(Q) = \mathbf{A}|\Phi$ to denote that Q is A-reducible to $(\mathbf{A}, \Phi)$. Queries that do not export $[A]$ are called $\emptyset$-reducible to a nullary[7] function Φ (denoted $\mathrm{red}_\emptyset(Q) = ■|\Phi$) if

$$\pi^{\Phi}_{\emptyset}(Q) = \{(\Phi)\}$$
$$\pi_C(Q) = \{(■)\} \qquad \text{for any attribute } C.$$

Next we define three classes of relations that are characterised by their A-reductions; let $\Phi_\top$ be the constant function $\Phi_\top(.) = \top$.

$$Q^{[A]} \in \mathcal{Q}^A \quad \text{if} \quad \mathrm{red}_A(Q) = \mathbf{A}|\Phi \text{ or } \mathrm{red}_A(Q) = \mathbf{A}|\neg\Phi \tag{3.2}$$
$$Q^{[A]} \in \mathcal{Q}_{\text{fill}} \quad \text{if} \quad \mathrm{red}_A(Q) = \mathbf{A}|\Phi_\top \tag{3.3}$$
$$Q^{[\neg A]} \in \mathcal{Q}_{\text{fill}} \quad \text{if} \quad \mathrm{red}_\emptyset(Q) = ■|\Phi_\top \tag{3.4}$$
$$Q \in \mathcal{Q}_\emptyset \quad \text{if} \quad Q = \emptyset \tag{3.5}$$

Queries $\mathcal{Q}^A$ are relations in which the values of $[A]$-attributes are populated with values from $\mathbf{A}$, and values for non-$[A]$-attributes are set to ■. There is a functional dependency $[A] \to \Phi$ such that every tuple is represented by its $[A]$-value a and has a corresponding annotation $\Phi(a)$ or $\neg\Phi(a)$. Queries $\mathcal{Q}_{\text{fill}}$ are similar to Q_A-queries, but every tuple is annotated with $\top$. Queries $\mathcal{Q}_\emptyset$ are simply empty relations.

Example 14. Given the domain $\mathbf{A} = \{x_1, x_2, x_3\}$, the following relation X over the distinguished attribute A and two attributes B, C with $B, C \notin [A]$ satisfies the properties of a $\mathcal{Q}^A$ query, and relation Y is a $\mathcal{Q}_{\text{fill}}$-query.

$\mathcal{Q}^A$-relation X

A_x	B_x	C_x	$\mathbf{\Phi}$
x_1	■	■	$\mathbf{x_1}$
x_2	■	■	$\mathbf{x_2}$
x_3	■	■	$\mathbf{x_3}$

$\mathcal{Q}_{\text{fill}}$-relation Y

A_y	B_y	C_y	$\mathbf{\Phi}$
x_1	■	■	$\top$
x_2	■	■	$\top$
x_3	■	■	$\top$

[7] I.e., a constant.

$\mathcal{Q}_1$	Op	$\mathcal{Q}_2$	$\mathcal{Q}_1\ Op\ \mathcal{Q}_2$
$\mathcal{Q}^A$	$\ltimes$	$\mathcal{Q}_{\text{fill}}$	$\mathcal{Q}^A$
	$-$	$\mathcal{Q}_\emptyset$	$\mathcal{Q}^A$
$\mathcal{Q}_{\text{fill}}$	$\ltimes$	$\mathcal{Q}^A$	$\mathcal{Q}^A$
		$\mathcal{Q}_{\text{fill}}$	$\mathcal{Q}_{\text{fill}}$
	$-$	$\mathcal{Q}^A$	$\mathcal{Q}^A$
		$\mathcal{Q}_\emptyset$	$\mathcal{Q}_{\text{fill}}$

Figure 3.7: Class membership of queries connecting classes Q^A, Q_{fill}, $Q_\emptyset$ with $\ltimes$ or $-$.

Note that in relation X we use the same symbols x_i both as data values for A, and for annotations. The functional dependency $A_x \to \Phi$ is satisfied by $\Phi(x_i) = x_i$. □

Figure 3.7 shows how $\mathcal{Q}^A$, $\mathcal{Q}_{\text{fill}}$, and $\mathcal{Q}_\emptyset$-queries are propagated through query operators. Given query classes $\mathcal{Q}_1$ and $\mathcal{Q}_2$, the right-most column ($\mathcal{Q}_1\ Op\ \mathcal{Q}_2$) in the table shows which class a query that combines two queries from those respective classes by operator Op belongs to.

Example 15. Continuing with the previous Example 14, the join $X \ltimes Y$ (with equality joins between the corresponding A, B, C attributes) of $\mathcal{Q}^A$-query X and $\mathcal{Q}_{\text{fill}}$-query Y yields the following relation:

$\mathcal{Q}^A$-query $X \ltimes Y$

A_x	A_y	B_x	B_y	C_x	C_y	$\mathbf{\Phi}$
x_1	x_1	■	■	■	■	$\mathbf{x_1}$
x_2	x_2	■	■	■	■	$\mathbf{x_2}$
x_3	x_3	■	■	■	■	$\mathbf{x_3}$

Clearly, $X \ltimes Y$ satisfies the conditions of a $\mathcal{Q}^A$-query as suggested by the rule $\mathcal{Q}^A \ltimes \mathcal{Q}_{\text{fill}} \to \mathcal{Q}^A$ in Figure 3.7. Similarly, the result of $Y - X$ is

$\mathcal{Q}^A$-query $Y - X$

A_y	B_y	C_y	$\mathbf{\Phi}$
x_1	■	■	$\neg\mathbf{x_1}$
x_2	■	■	$\neg\mathbf{x_2}$
x_3	■	■	$\neg\mathbf{x_3}$

This relation satisfies the conditions of a $\mathcal{Q}^A$-query as implied by the rule $\mathcal{Q}_{\text{fill}} - \mathcal{Q}^A \to \mathcal{Q}^A$ in Figure 3.7. □

Now let $Q^{[A]}$ be a query that contains a $\mathcal{Q}^A$-relation $X^{[A]}$. The goal is to populate the relations of Q such that Q is a $\mathcal{Q}^A$-query, i.e., that Q satisfies the above properties for $\mathcal{Q}^A$. More formally, we show the following:

Lemma 11 (Preserving one distinguished attribute)**.** *Given a query Q, a distinguished attribute A of Q, and a distinguished relation X^A of Q that satisfies Equation* (3.2)*, the remaining relations of Q can be filled such that Q satisfies Equation* (3.2)*.*

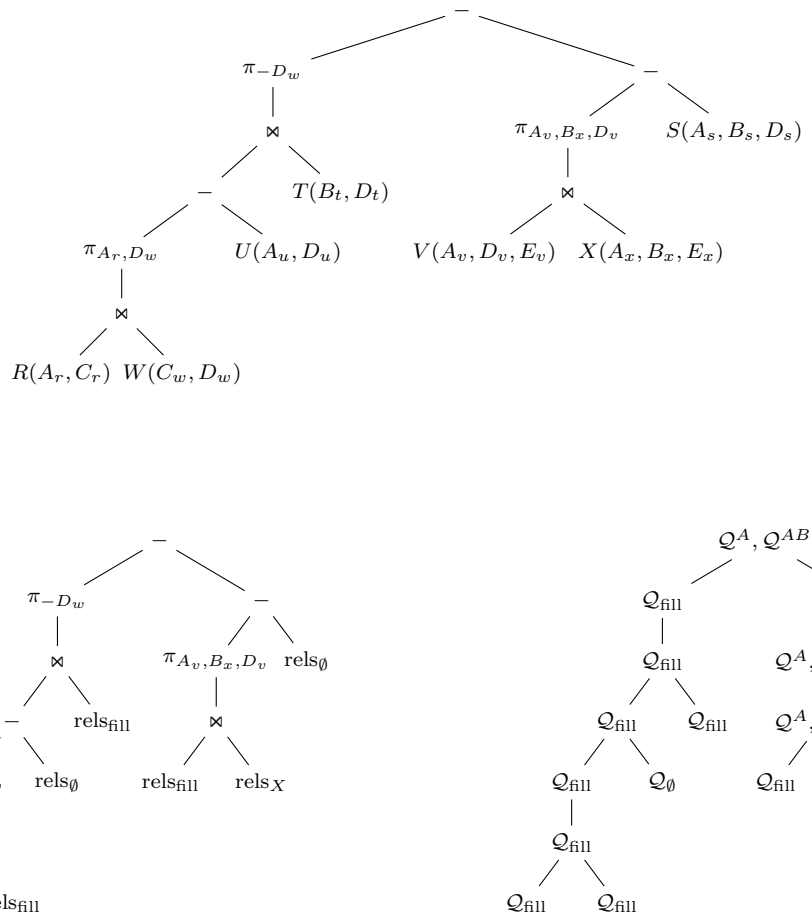


Figure 3.8: A query Q (top) and the corresponding partitioning of its relations into rels_X, $\text{rels}_\emptyset$, and $\text{rels}_{\text{fill}}$ relations (bottom-left), assuming that relation X and its annotations are to be preserved by the filling. The bottom-right figure shows how the query classes of the sub-queries of Q are propagated through query operators. If we fill for the one distinct attribute $[A]$ whose values are to be preserved, then all sub-queries rooted in operators on the path from X to the root of Q are $\mathcal{Q}^A$-queries; if the two attributes $[A]$ and $[B]$ are to be preserved, then these queries are $\mathcal{Q}^{AB}$-queries. Note that since X has odd polarity in Q, the annotations of tuples in Q are the negated annotations of the corresponding tuples in X.

Proof. The first step is to identify the set $\mathcal{OP}_-$ of $-$ operators in Q that *do not have X as a right descendant*. Next we partition the relations of Q into three sets:

$$\begin{aligned}\text{rels}_X &= \{X\}\\ \text{rels}_\emptyset &= \text{set of relations that are right descendants of a } \mathcal{OP}_- \text{ operator}\\ \text{rels}_{\text{fill}} &= \text{all other relations}\end{aligned}$$

Finally, we populate every $\text{rels}_{\text{fill}}$ relation as a $\mathcal{Q}_{\text{fill}}$-query, and every $\text{rels}_\emptyset$ relation as a $\mathcal{Q}_\emptyset$-query. For the former, it suffices to populate each $[A]$ attribute of a $\text{rels}_{\text{fill}}$-relation with $\mathbf{A}$, and each non-$[A]$-attribute with ■. The following inductive argument shows that every operator on the path in Q between X and the root of Q is a $\mathcal{Q}^A$-query: First, this trivially holds at X itself. Now let Op be an operator on the path between X and the root of Q. There are the following cases:

- $Q_L \ltimes Q_R$ where without loss of generality Q_L contains X and hence Q_L is a $\mathcal{Q}^A$-query. Then Q_R contains a relation from $\text{rels}_{\text{fill}}$ and by the propagation rules in Figure 3.7, Q_R is a $\mathcal{Q}_{\text{fill}}$-query, and $Q_L \ltimes Q_R$ is a $\mathcal{Q}^A$-query.
- $Q_L - Q_R$ where Q_L contains X. Then the $-$ operator is in $\mathcal{OP}_-$ and Q_R is a $\mathcal{Q}_\emptyset$-query, Q_L is a $\mathcal{Q}^A$-query, and hence $Q_L - Q_R$ is a $\mathcal{Q}^A$-query.
- $Q_L - Q_R$ where Q_R contains X and hence Q_R is a $\mathcal{Q}^A$-query. Then Q_L contains a relation from $\text{rels}_{\text{fill}}$ and by the propagation rules in Figure 3.7 Q_L is a $\mathcal{Q}_{\text{fill}}$-query, and $Q_L - Q_R$ is a $\mathcal{Q}^A$-query. □

Remark 5. Note that if X has even polarity in Q, then the annotation $\Phi_Q(a)$ of a tuple (a) in $\pi_{[A]}(Q)$ is the same as the corresponding annotation $\Phi_X(a)$ of a tuple (a) in $\pi_{[A]}(X)$; conversely, if X has odd polarity in Q, then $\Phi_Q(a) = \neg\Phi_X(a)$. □

Example 16. Consider the query Q depicted in Figure 3.8 (top figure) and assume that we would like to preserve relation X with respect to its attribute A_x; we use the $[A]$-domain $\mathbf{A} = \{x_1, x_2, x_3\}$ and assume X's values to be $\mathbf{A}$ and its annotations to be $\Phi(x_i) = x_i$ as in Example 16. The bottom-left graph in Figure 3.8 shows the partition of Q's relations into $\text{rels}_X = \{X\}$, $\text{rels}_{\text{fill}} = \{R, W, T, V\}$ and $\text{rels}_\emptyset = \{U, S\}$. We hence set $U = S = \emptyset$, and populate in relations from $\text{rels}_{\text{fill}}$ all $[A]$ attributes (A_r, A_u, A_v) with $\mathbf{A}$ and all other attributes with ■. The bottom-right graph shows how the $\mathcal{Q}^A$, $\mathcal{Q}_{\text{fill}}$ and $\mathcal{Q}_\emptyset$ sub-queries are propagated by Q's operators. For example, the left and right sub-queries of the root operator are the following relations:

Left sub-query of root(Q)

A_r	B_t	D_t	$\mathbf{\Phi}$
x_1	■	■	$\top$
x_2	■	■	$\top$
x_3	■	■	$\top$

Right sub-query of root(Q)

A_v	B_x	D_v	$\mathbf{\Phi}$
x_1	■	■	$\mathbf{x_1}$
x_2	■	■	$\mathbf{x_2}$
x_3	■	■	$\mathbf{x_3}$

The relation represented by Q is hence:

Q			
A_r	B_t	D_t	$\mathbf{\Phi}$
x_1	■	■	$\neg\mathbf{x_1}$
x_2	■	■	$\neg\mathbf{x_2}$
x_3	■	■	$\neg\mathbf{x_3}$

Evidently, Q preserves the $[A]$ attribute and the annotations of relation X. Following Remark 5, the annotations of tuples in Q are exactly the negation of the corresponding annotations in X because X has odd polarity in Q. □

Preserving the data of two attributes

Let us now extend the above technique to queries that contain relations over two distinct attributes A and B whose values we would like to preserve; the first step is to generalise the notation of A-reducible relations. Let Φ^{AB} be a total function on $\mathbf{A} \times \mathbf{B}$, and let Φ^A be a total function on $\mathbf{A} \cup \mathbf{A} \times \mathbf{B}$ such that $\Phi^A(a) \equiv \bigvee_{b \in \mathbf{B}} \Phi^A(a, b)$ for all $a \in \mathbf{A}$; $\Phi_\top$ is the constant function $\Phi_\top(.) = \top$. As before, a relation Q is *A-reducible to* $(\mathbf{A}, \Phi^A)$, if

$$\pi^{\Phi}_{[A]}(Q) = \{(a|\Phi^A(a)) \mid a \in \mathbf{A}\}$$
$$\pi_C(Q) = \{(■)\} \qquad \text{for any attribute } C \text{ with } C \notin [A].$$

Similarly, Q is AB-reducible to $(\mathbf{A} \times \mathbf{B}, \Phi^{AB})$ if

$$\pi^{\Phi}_{[A][B]}(Q) = \{(a, b|\Phi^{AB}(a, b)) \mid a \in \mathbf{A}, b \in \mathbf{B}\}$$
$$\pi_C(Q) = \{(■)\} \qquad \text{for any attribute } C \text{ with } C \notin [A] \cup [B].$$

We write $\text{red}_{AB}(Q) = \mathbf{A} \times \mathbf{B}|\Phi^{AB}$ to denote that Q is AB-reducible to $(\mathbf{A} \times \mathbf{B}, \Phi^{AB})$. We next define the following classes of queries:

$$Q^{[A][\neg B]} \in \mathcal{Q}^A \text{ if } \text{red}_A(Q) = \mathbf{A}|\Phi^A \text{ or } \text{red}_A(Q) = \mathbf{A}|\neg\Phi^A \tag{3.6}$$
$$Q^{[A][B]} \in \mathcal{Q}^A \text{ if } \text{red}_{AB}(Q) = \mathbf{A} \times \mathbf{B}|\Phi^A \text{ or } \text{red}_{AB}(Q) = \mathbf{A} \times \mathbf{B}|\neg\Phi^A \tag{3.7}$$
$$Q^{[A][B]} \in \mathcal{Q}^{AB} \text{ if } \text{red}_{AB}(Q) = \mathbf{A} \times \mathbf{B}|\Phi^{AB} \text{ or } \text{red}_{AB}(Q) = \mathbf{A} \times \mathbf{B}|\neg\Phi^{AB} \tag{3.8}$$
$$Q^{[A][\neg B]} \in \mathcal{Q}_{\text{fill}} \text{ if } \text{red}_A(Q) = \mathbf{A}|\Phi_\top \tag{3.9}$$
$$Q^{[A][B]} \in \mathcal{Q}_{\text{fill}} \text{ if } \text{red}_{AB}(Q) = \mathbf{A} \times \mathbf{B}|\Phi_\top \tag{3.10}$$
$$Q^{[\neg A][\neg B]} \in \mathcal{Q}_{\text{fill}} \text{ if } \text{red}_\emptyset(Q) = ■|\Phi_\top \tag{3.11}$$
$$Q \in \mathcal{Q}_\emptyset \text{ if } Q = \emptyset \tag{3.12}$$

Queries from these classes are propagated by query operators as depicted in Figure 3.9. The extension of Lemma 11 to the case of one or two attributes is:

$\mathcal{Q}_1$	Op	$\mathcal{Q}_2$	$\mathcal{Q}_1\ Op\ \mathcal{Q}_2$
$\mathcal{Q}^A$	$\ltimes$	$\mathcal{Q}_{\text{fill}}$	$\mathcal{Q}^A$
	$-$	$\mathcal{Q}_\emptyset$	$\mathcal{Q}^A$
$\mathcal{Q}^{AB}$	$\ltimes$	$\mathcal{Q}_{\text{fill}}$	$\mathcal{Q}^{AB}$
	$-$	$\mathcal{Q}_\emptyset$	$\mathcal{Q}^{AB}$
$\mathcal{Q}_{\text{fill}}$	$\ltimes$	$\mathcal{Q}^A$	$\mathcal{Q}^A$
		$\mathcal{Q}^{AB}$	$\mathcal{Q}^{AB}$
		$\mathcal{Q}_{\text{fill}}$	$\mathcal{Q}_{\text{fill}}$
	$-$	$\mathcal{Q}^A$	$\mathcal{Q}^A$
		$\mathcal{Q}^{AB}$	$\mathcal{Q}^{AB}$
		$\mathcal{Q}_\emptyset$	$\mathcal{Q}_{\text{fill}}$

Figure 3.9: Class membership of queries connecting classes Q^A, Q_{fill}, $Q_\emptyset$ with $\ltimes$ or $-$.

Lemma 12 (Preserving distinguished attributes)**.** *Given a query Q, two distinguished attributes A of B of Q, and a distinguished relation $X^{A\neg B}$ of Q that satisfies Equation* (3.6)*, the remaining relations of Q can be filled such that Q satisfies Equation* (3.6) *if Q exports* $[A]$ *but not* $[B]$*, or Equation* (3.7) *if Q exports* $[A]$ *and* $[B]$*. Similarly, for a query Q with a distinguished relation X^{AB}, the remaining relations in Q can be filled such that Q satisfies Equation* (3.8) *if Q exports* $[A]$ *and* $[B]$*.*

Proof. Let $Q^{[A][B]}$ be a query that contains a $\mathcal{Q}^A$-relation $X^{[A]}$ by virtue of Equation (3.6). The goal is to show that Q is a $\mathcal{Q}^A$-query according to Equation (3.7).

As in the proof of Lemma 11, we identify the set of $-$ operators $\mathcal{OP}_-$ that do not have X as a right descendant and then partition the relations of Q into three sets:

$$\begin{aligned}\text{rels}_X &= \{X\}\\ \text{rels}_\emptyset &= \text{set of relations that are right descendants of a } \mathcal{OP}_- \text{ operator}\\ \text{rels}_{\text{fill}} &= \text{all other relations}\end{aligned}$$

The relations of Q are populated similar to before: Every $\text{rels}_{\text{fill}}$ relation as a $\mathcal{Q}_{\text{fill}}$-query, and every $\text{rels}_\emptyset$ relation as a $\mathcal{Q}_\emptyset$-query. For the former, it suffices to populate each $[A]$ attribute of a $\text{rels}_{\text{fill}}$-relation with **A**, every $[B]$ attribute with **B**, and each non-$[A]$-attribute with ■. Relations in $\text{rels}_{\text{fill}}$ that export both $[A]$ and $[B]$ (i.e., queries that need to satisfy Equation (3.10)) are populated with the cross-product of **A** and **B** for attributes $[A]$ and $[B]$ and with ■ for all other attributes. The inductive argument along the path from X to the root of Q is similar to before; note however, that the lowest $\ltimes$-operator on the path from X to the root of Q that introduces a $[B]$-attribute marks the transition from a $\mathcal{Q}^A$ sub-query of type $Q^{[A][\neg B]}$ (Equation (3.6)) to $Q^{[A][B]}$ (Equation (3.7)). That is, the tuples $(a|\Phi(a))$ in X are expanded to tuples $(a, b|\Phi(a))$ by means of the cross product between the $\mathcal{Q}^A$-relation that does not export $[B]$ and the $\mathcal{Q}_{\text{fill}}$-relation that does export $[B]$.

Finally, let us consider the case of a query $Q^{[A][B]}$ that contains a $\mathcal{Q}^{AB}$-relation $X^{[A][B]}$. It follows exactly as above that Q is a $\mathcal{Q}^{AB}$-query, i.e., Q is equivalent to X with respect to attributes $[A]$ and $[B]$ and the annotations of the (a, b)-tuples. □

X

A_x	B_x	E_x	$\mathbf{\Phi}$
x_1	y_1	■	$\mathbf{x_{11}}$
x_1	y_2	■	$\mathbf{x_{12}}$
x_2	y_1	■	$\mathbf{x_{21}}$
x_2	y_2	■	$\mathbf{x_{22}}$
x_3	y_1	■	$\mathbf{x_{31}}$
x_3	y_2	■	$\mathbf{x_{32}}$

Q_{RWU}

A_r	D_w	$\mathbf{\Phi}$
x_1	■	$\top$
x_2	■	$\top$
x_3	■	$\top$

Q_{RWUT}

A_r	B_t	D_t	$\mathbf{\Phi}$
x_1	y_1	■	$\top$
x_1	y_2	■	$\top$
x_2	y_1	■	$\top$
x_2	y_2	■	$\top$
x_3	y_1	■	$\top$
x_3	y_2	■	$\top$

Q_{VXS}

A_v	B_x	D_v	$\mathbf{\Phi}$
x_1	y_1	■	$\mathbf{x_{11}}$
x_1	y_2	■	$\mathbf{x_{12}}$
x_2	y_1	■	$\mathbf{x_{21}}$
x_2	y_2	■	$\mathbf{x_{22}}$
x_3	y_1	■	$\mathbf{x_{31}}$
x_3	y_2	■	$\mathbf{x_{32}}$

Q

A_r	B_t	D_t	$\mathbf{\Phi}$
x_1	y_1	■	$\neg\mathbf{x_{11}}$
x_1	y_2	■	$\neg\mathbf{x_{12}}$
x_2	y_1	■	$\neg\mathbf{x_{21}}$
x_2	y_2	■	$\neg\mathbf{x_{22}}$
x_3	y_1	■	$\neg\mathbf{x_{31}}$
x_3	y_2	■	$\neg\mathbf{x_{32}}$

Figure 3.10: Relations used in Example 17.

Remark 5 regarding the polarity and the sign of the annotations in X and Q carries over to both of the above cases.

Example 17. Referring again to the query in Figure 3.8, let us now assume that we would like to preserve relation X with respect to $[A]$ and $[B]$; we assume the domains $\mathbf{A} = \{x_1, x_2, x_3\}$ and $\mathbf{B} = \{y_1, y_2\}$ and take X to be the $\mathcal{Q}^{AB}$-relation X depicted in Figure 3.10.

Note that the sets of relations rels_X, $\text{rels}_{\text{fill}}$ and $\text{rels}_\emptyset$ are identical the case of Example 16 and are depicted in the bottom-left graph in Figure 3.8. Also the propagation of the $\mathcal{Q}^{AB}$ and $\mathcal{Q}_{\text{fill}}$ sub-queries through the query is as before and depicted in the bottom-right graph. However, the relations represented by the different sub-queries are now different.

Let us first consider the sub-query Q_{RWU} consisting of relations R, W, U. Attribute A_r is filled with $\mathbf{A}$, and attributes C_r, C_w, D_w are set to ■; relation U is set to $\emptyset$; all annotations are $\top$. Then Q_{RWU} is a $\mathcal{Q}_{\text{fill}}$-query by Equation (3.9); the represented relation Q_{RWU} is again show in Figure 3.10.

In relation T, attribute B_t is filled with $\mathbf{B}$ and D_t is filled with ■; its annotations are $\top$. The join between Q_{RWU} and T enforces an equality join $D_w = D_t$, but enforces no constraint between A_r and B_t; this yields the relation depicted in Figure 3.10 that is the result of query Q_{RWUT} that consists of relations R, W, U, T.

Q_{RWUT} is a $\mathcal{Q}_{\text{fill}}$-query by virtue of Equation (3.10). On the right side of Q, it follows similarly to above that the sub-query Q_{VXS} consisting of relations V, X, S is the $\mathcal{Q}^{AB}$-query Q_{VXS} depicted in Figure 3.10. This leads to the $\mathcal{Q}^{AB}$-query $Q = Q_{RWUT} - Q_{VXS}$ as depicted in the figure. □

3.4 #P-hardness of non-hierarchical queries

The main result shown in this section is that probabilistic query evaluation for non-hierarchical $\text{RA}^{\cup}$ queries has #P-hard data complexity:

Lemma 13. *The probabilistic query evaluation problem for any non-hierarchical $RA^{\not\cup}$ query has #P-hard data complexity.*

Given such a query Q and a 2DNF formula Ψ, we provide a reduction from the model-counting problem #2DNF by means of an explicit construction of a database $\mathcal{D}$ such that $P_{Q(\mathcal{D})} = P_\Phi$. The details of the construction of this reduction depend on structural properties of the respective query which are formalised by the notions of *patterns* and *matches*. We first define patterns and matches, and then show that *matching a pattern* is equivalent to the non-hierarchical property (Proposition 14). The notion of a match is then refined to that of a *lineage-preserving match* in which the three distinct relations R, S, T identified by the pattern satisfy additional structural properties that allow us to populate these relations such that their annotations are preserved in the query (Lemma 16). This puts us in the position to give an explicit reduction from #2DNF to probabilistic query evaluation by analysing a finite number of of classes of queries separately, segmented by the pattern with which they have a lineage-preserving match (Lemma 17). More formally:

Proof. Lemma 13 follows from the formal statements derived in this section as follows:

$$
\begin{array}{ll}
 & Q \text{ is non-hierarchical} \\
\underset{\text{Proposition 14}}{\Leftrightarrow} & Q \text{ has a match with a pattern} \\
\underset{\text{Lemma 16}}{\Leftrightarrow} & Q \text{ has a lineage-preserving match with a pattern} \\
\underset{\text{Lemma 17}}{\Rightarrow} & Q \text{ is hard for \#P} \qquad \square
\end{array}
$$

3.4.1 Patterns and matches

Definition 9 (Query pattern)**.** *A* query pattern *(pattern, for short) over two attributes A, B and two relational operators Op_1, $Op_2 \in \{\bowtie, -\}$ is a binary tree with three leaves A, B, AB, root node Op_1, and inner node Op_2.*

Figure 3.11 shows query patterns. There are $2 \cdot 2 \cdot 2 \cdot 6 = 48$ different patterns: There are two distinct unlabeled binary trees with three leaves, the two operators can each be either $\bowtie$ or $-$, and there are 6 possible orders of the labels A, AB, and B. Figure 3.11 shows 24 of the 48 patterns and omits for each pattern the corresponding pattern obtained by swapping leaves A and B.

Definition 10 (Match of query and pattern)**.** *A $RA^{\not\cup}$ query Q* matches a pattern P over A and B *if Q contains three relations $R^{[A][\neg B]}$, $S^{[A][B]}$, $T^{[\neg A][B]}$ such that P can be embedded in Q in the intuitive way: The operators in Q given by the least common ancestors of R, S, T correspond to the operators in P, and the locations of R, S, T in Q with respect to these operators correspond to the locations of A, AB, B in P.*

Q matches a pattern *if it matches a pattern over some pair of its attributes.*

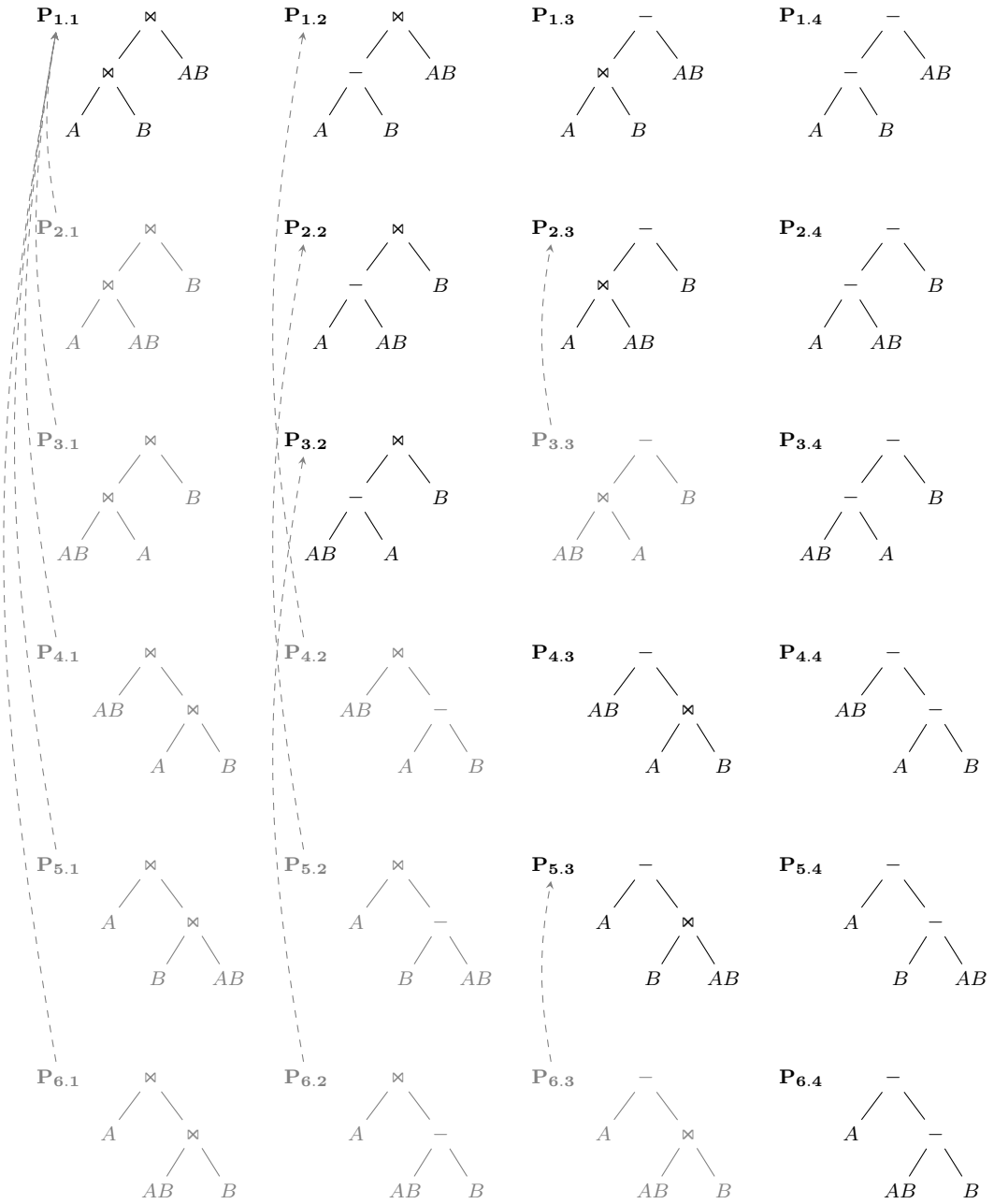


Figure 3.11: The 24 query patters $P_{1.1}$, ..., $P_{6.4}$. The 10 grey patterns can by reduced to other patterns as indicated by the arrows. For this reduction, note that firstly the labels A and B are symmetric with respect to the definition of of match between a pattern and a query and hence can be swapped, and secondly the $\bowtie$ operator is commutative and hence its left and right sub-query can be swapped without changing the query.

We also say that Q is an (R, S, T)-match for P to emphasise which three relations establish the match. Note that in this definition, attributes A and B are to be understood as being placeholders for any two distinct attributes. In the following, for a match between Q and P, we will denote the two operators of P by Op_1^P (root operator) and Op_2^P, and the corresponding operators in Q by Op_1 and Op_2.

Example 18. Figure 3.12 shows examples of queries and matches. □

Matching a pattern is intimately linked with the non-hierarchical property:

Proposition 14 (Non-hierarchical queries match patterns)**.** *A $RA^{\not\cup}$ query is non-hierarchical if and only if it matches a pattern.*

Proof. For the $\Rightarrow$ direction, let Q a non-hierarchical query. Then by Definition 6, Q has two attributes A and B such that it contains relations $R^{[A][\neg B]}$, $S^{[A][B]}$, $T^{[\neg A][B]}$. Clearly, Q matches exactly the pattern P whose two operators correspond to the operators Op_1 and Op_2 in Q. (Note that the patterns are exhaustive in the sense that there is exactly one pattern per possible combination of the operators $\ltimes, -$.)

The $\Leftarrow$ direction is immediate, since every query Q that matches a pattern contains three distinct relation symbols R, S, T that render Q non-hierarchical by Definition 6. □

Any query that matches a pattern satisfies a set of structural properties as detailed by the following lemma.

Lemma 15 (Properties of matching queries)**.** *Let P be a pattern and query Q an (R, S, T)-match for P. Then Q has the following properties:*

1. *Any sub-query of Q rooted in an operator on the path in Q between R and S exports $[A]$. Any sub-query of Q rooted in an operator on the path in Q between T and S in Q exports $[B]$.*

2. *Given two attributes A, B such that $[A] = [B]$, and such that the least common ancestor of the relations of A and B is a join $\ltimes_\rho$ and $A, B \in \mathcal{E}(Q)$, then Q is invariant under adding the join condition $(A = B)$ to ρ. In other words, (transitively inferred) join conditions can be pushed down to the least common ancestor of the joined attributes without altering the query.*

Proof. Property 1 holds since both R and S export $[A]$ and in order for the $[A]$-attributes of R and S to be in the transitive closure $[A]$, each operator on the path between R and S must export $[A]$. The symmetric argument holds for S and T and attributes $[B]$. Property 2 follows immediately from the fact that equality constraints are transitive, cf Remark 3. □

3.4.1.1 Lineage-preserving match

The notion of a *match* is further specialised to that of a *lineage-preserving match* in the two upcoming definitions. The name "lineage-preserving" hints at the following property of lineage-preserving matches: The relations of Q can be populated in such way that they leave the tuples and their annotations of R, S, and T unchanged. We will expand on this in Section 3.4.2.

Definition 11 (Left-deep operator)**.** *An operator Op is called* left-deep *in a $RA^{\not\downarrow}$ query Q if for every $-$ operator Op_- on the path between the root of Q and Op, Op is a* left *descendant of Op_-.*

Definition 12 (Lineage-preserving match)**.** *A $RA^{\not\downarrow}$ query Q is a* lineage-preserving match *for a pattern P over A and B if Q contains three relations $R^{[A][\neg B]}$, $S^{[A][B]}$, $T^{[\neg A][B]}$ such that P can be embedded in Q in the following way:*

1. *The locations of R, S, T in Q with respect to operators Op_1 and Op_2 correspond to the locations of A, AB, B in P.*
2. *The two distinct operators in P and Q correspond, i.e., Op_1 (Op_2, respectively) in Q is the same operator as the top-most (bottom-most, respectively) operator in P.*
3. *For every $-$ operator Op_- that is an ancestor of Op_1 in Q, it holds that if Op_1 is a right descendant of Op_-, then Op_- does not export $[A]$ or $[B]$.*
4. *If Op_2 is a left descendant of Op_1 in Q, then Op_2 is* left-deep *in the sub-query of Q rooted in the left child of Op_1.*

As in the case of Definition 10, we use the short notation *lineage-preserving* (R, S, T)*-match* when no confusion arises.

Example 19. Figure 3.12 shows examples of queries and lineage-preserving matches. □

Connection between matches and lineage-preserving matches. Lemma 16 establishes that any query that matches a pattern necessarily also has a lineage-preserving match with a (possibly different) pattern; furthermore, the relation symbols that establish the lineage-preserving match can be found by exploring the query tree in the following standard left-depth-first *in-order*.

Definition 13. *Let T be a binary tree, and $o_1, \ldots, o_n$ its nodes, and π a permutation of $\{1, \ldots, n\}$. The permutation is an* in-order *of the nodes if for any two nodes $o_{\pi(j)}, o_{\pi(i)}$ the following condition holds: $o_{\pi(i)}$ is a left descendant of $o_{\pi(j)}$ and $i < j$, or $o_{\pi(i)}$ a right descendant of $o_{\pi(j)}$ and $j < i$, or $o_{\pi(j)}$ occurs to left of $o_{\pi(i)}$ in T and $j < i$.*

Lemma 16 (Finding a lineage-preserving match)**.** *Let Q be a $RA^{\not\emptyset}$ query and $o_1, \ldots, o_n$ be the sequence of its operators in in-order, and $Q_1, \ldots, Q_n$ be the corresponding sequence of sub-queries of Q rooted in $o_1, \ldots, o_n$.*

If Q_i is the first sub-query in the above order that matches a pattern, then Q_i contains three relation symbols that establish a lineage-preserving match between Q and a pattern.

Proof. We show that Q satisfies properties 1–4 from Definition 12 for a pattern P'. Let Op_1 and Op_2 be the two operators in Q as established by a match with a pattern P. Note that $o_i = Op_1$ by construction.

Without loss of generality, assume that Q_i has an $(R^{[A][\neg B]}, S^{[A][B]}, T^{[\neg A][B]})$-match with P. While properties 1 and 2 of Definition 12 are already satisfied by picking the pattern P' suitable[8] for relations R, S, T, it remains to be shown that Q contains relations that satisfy properties 3 and 4.

Property 3. Proof by contradiction. Assume that there is an operator $o_j = -$ that is an ancestor of $o_i = Op_1$ such that Op_1 is a right descendent of o_j. Note that $j < i$ due to the in-order of the query operators. Assume that o_j exports $[A]$ or $[B]$. Then the left sub-query of o_j contains at least one relation that exports $[A]$ or $[B]$. This relation together with a subset of R, S, T would establish a match of Q_j with some pattern; this is a contradiction to the assumption that Q_i is the first sub-query (in the in-order sequence of sub-queries) to establish a match. Hence o_j cannot export $[A]$ or $[B]$.

Property 4. Assume that Op_2 is a left descendant of Op_1 that is not left-deep in the left sub-query Q_L of Op_1. Let Op_- be the top-most $-$ operator in Q_L such that Op_2 is its right descendant. There are the following cases:

- Case 1: R and T are descendants of Op_2. Then S is a right descendant of Op_1 and by Lemma 15 it follows that Op_- exports $[A]$ and $[B]$. Thus the left sub-query of Op_- contains relations $X^{[A][\neg B]}$ and $Z^{[\neg A][B]}$, or it contains a relation $Y^{[A][B]}$. In the former case, the three relations X, S, Z establish a lineage-preserving match, with $Op_2 = Op_-$ and Op_1 as before. The latter case is a contradiction to the assumption that Q_i is the first sub-query (in the in-order sequence of sub-queries) of Q that matches a pattern, because the sub-query rooted at Op_- precedes Q_i in Q's in-order and matches a pattern via R, Y, T.

- Case 2: R and S are descendants of Op_2. Then T is a right descendant of Op_1 and by Lemma 15 it follows that Op_- exports $[B]$. Thus the left sub-query of Op_- contains a relation $Y^{[A][B]}$, or it contains a relation $X^{[\neg A][B]}$. In the former case, the three relations R, Y, T establish a lineage-preserving match, with $Op_2 = Op_-$ and Op_1 as before. The latter case is a contradiction to the assumption that Q_i is the first sub-query (in the in-order sequence of sub-queries) of Q that matches a pattern, because

[8] Note that the patterns are exhaustive in the sense that any arrangement of three relations $(R^{[A][\neg B]}, S^{[A][B]}, T^{[\neg A][B]})$ and two operators corresponds to exactly one pattern.

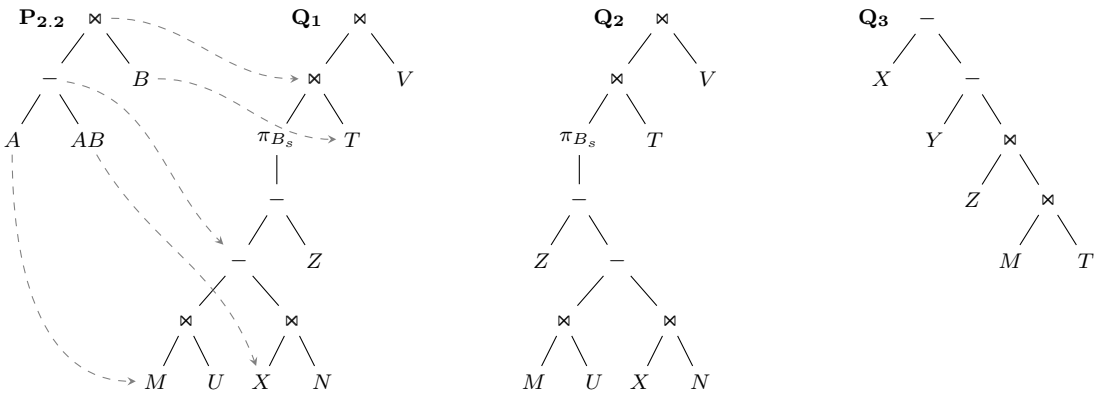


Figure 3.12: Illustrations of pattern $P_{2.2}$ and queries over the following schema: $M(A_m)$, $N(A_n)$, $T(B_t, C_t)$, $U(B_u)$, $V(B_v, C_v)$, $X(A_x, B_x)$, $Y(A_y, B_y)$, $Z(A_z, B_z)$. Pattern $P_{2.2}$ matches Q_1, for instance via $M(A_m)$, $X(A_x, B_x)$, $T(B_t, C_t)$ (indicated by arrows). Note that Q_1 also matches many other patterns (for instance via M, U, X, U, X, N, etc.). Q_1 is also a lineage-preserving match for $P_{2.2}$ via M, X, T, because Op_2 (the least common ancestor of M and X is a left descendant of the top-most $-$ operator. Although Q_2 matches $P_{2.2}$ (via the same relations as Q_1), the relations M, X, T do not establish a lineage-preserving match for $P_{2.2}$, because Op_2 is a right descendant of the top-most $-$ operator. However, relations M, Z, T establish a lineage-preserving match of Q_2 with pattern $P_{3.2}$, cf. Figure 3.11. Query Q_3 is a lineage-preserving (M, X, T)-match for pattern $P_{4.3}$.

the sub-query rooted at Op_- precedes Q_i in Q's in-order and matches a pattern via X, S, T.

- Case 3: T and S are descendants of Op_2. Symmetric to case 2. □

3.4.2 Hardness reductions for queries with a lineage-preserving match

We next show that any query that has a lineage-preserving match with one of the 24 patterns $P_{1.1}, \ldots, P_{6.4}$ (cf. Figure 3.11) is hard for #P. These patterns are the smallest hard patterns for $\text{RA}^{\not\vee}$.

Lemma 17. *Any $RA^{\not\vee}$ query that has a lineage-preserving match for one of the patterns $P_{1.1}, \ldots, P_{6.4}$ (cf. Figure 3.11) has #P-hard data complexity.*

The proof spans the remainder of Section 3.4.2. For each of the patterns we give a direct reduction from the #2DNF problem: Let Q be a query that is a lineage-preserving (R, S, T)-match for a pattern P, and let $\Psi = \bigvee_{(i,j)\in E} x_i y_j$ be a 2DNF formula with $|E|$ clauses over disjoint variable sets $\mathbf{X}$ and $\mathbf{Y}$. We show how to construct in polynomial time a tuple-independent database $\mathcal{D}$ such that the annotation of $Q(\mathcal{D})$ is exactly Ψ (or $\neg\Psi$) and hence $P_{Q(\mathcal{D})} = P_\Psi = \#\Psi \cdot 2^{-|\text{vars}(\Psi)|}$ (or $P_{Q(\mathcal{D})} = 1 - P_\Psi$).

The basic idea for constructing $\mathcal{D}$ is to populate the relations R, S, T with tuples and annotations such that the annotation of $Q(\mathcal{D})$ is exactly Φ *under the assumption that the*

remaining relations in Q do not alter the annotations as produced by R, S, T. How exactly the remaining relations need to be filled to satisfy this assumption depends on the particular structure of the matched pattern and will follow from Lemma 12.

By Definition 12, Q contains two distinct operators Op_1 and Op_2 and relations $R^{[A][\neg B]}$, $S^{[A][B]}$, $T^{[\neg A][B]}$. In the following, sub-queries Q_R, Q_S, Q_T of Q are defined to be the left or right sub-queries of Op_1 or Op_2 that contain exactly one of R, S, or T, respectively, cf. Figure 3.13. Additionally, if Op_2 has sub queries Q_R, Q_S, then Q_{RS} is the sub-query of Op_1 that contains Q_R and Q_S. Sub queries Q_{RT} and Q_{ST} are defined symmetrically for matches in which R and T or S and T are descendants of Op_2, respectively.

The remainder of the proof treats the case of each pattern separately and spreads over Sections 3.4.2.1 – 3.4.2.6.

Remark 6 (Symmetric matches). By exploiting symmetries in queries and patterns, it suffices to give an explicit reduction for the 14 patterns shown in dark colour in Figure 3.11. First, note that each of the 24 patterns not show in the figure can be obtained from one the shown patterns by swapping labels A and B; since all definitions are symmetric in A and B, it suffices to consider the 24 patterns in the figure.

The symmetry of the join operator allows us to directly obtain the following reductions (indicated with arrows in Figure 3.11): $P_{2.1} \to P_{1.1}$, $P_{3.1} \to P_{1.1}$, $P_{4.1} \to P_{1.1}$, $P_{5.1} \to P_{1.1}$, $P_{6.1} \to P_{1.1}$, $P_{4.4} \to P_{1.2}$, $P_{5.2} \to P_{2.2}$, $P_{6.2} \to P_{3.2}$, $P_{3.3} \to P_{3.2}$, $P_{6.3} \to P_{5.3}$. □

3.4.2.1 Reduction for patterns $P_{1.x}$

Pattern $P_{1.1}$. Let Q be a query that is a lineage-preserving match for $P_{1.1}$. Then by Definition 12, Q contains operators $Op_1 = \ltimes$ and $Op_2 = \ltimes$ and relations $R^{[A][\neg B]}, S^{[A][B]}, T^{[\neg A][B]}$ arranged as in Figure 3.13. By Lemma 15, any operator on the path R – Op_2 – Op_1 – S exports $[A]$, and every operator on the path T – Op_2 – Op_1 – S exports $[B]$. Operator Op_1 expresses a join on both $[A]$ and $[B]$ (cf. Remark 3).

Figure 3.13 depicts such a query and marks the sub queries Q_R, Q_S, Q_T, Q_{RT} as the four sub-queries of Op_1, Op_2. Since Q is a lineage-preserving match, it satisfies the following additional structural properties:

1. If Op_1 is a right descendant of a $-$ operator, then this operator does not export $[A]$ or $[B]$.

2. Operator Op_2 is left-deep in Q_{RT}, i.e., it is the left descendant of any $-$ operator on the path between Op_1 and Op_2.

We next show how to fill relations R, S, T as $\mathcal{Q}^A$, $\mathcal{Q}^B$, and $\mathcal{Q}^{AB}$-relations according to Equations 3.6 and 3.8 with values from domains $\mathbf{X}$ for $[A]$ and $\mathbf{Y}$ for $[B]$, respectively. The

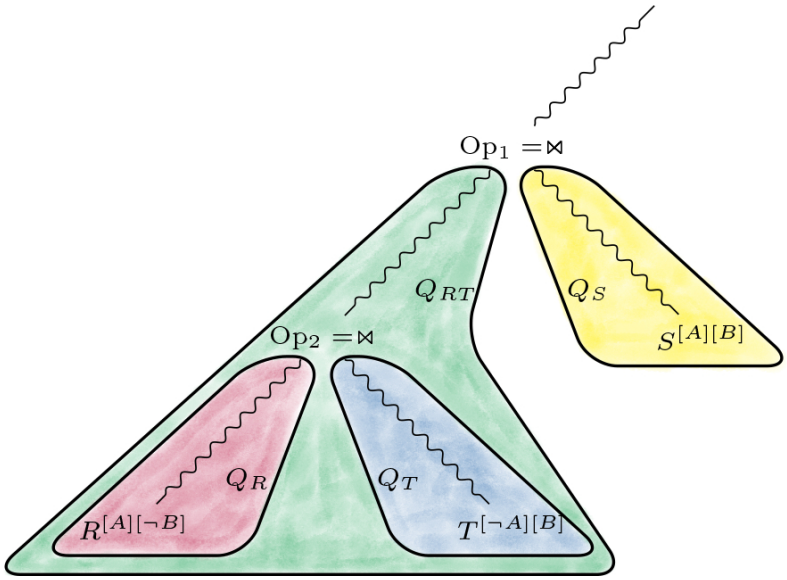


Figure 3.13: Schematic illustration of a query that is a lineage-preserving match for pattern $P_{1.1}$. A curly edge indicates that arbitrary other operators may occur on this path. By Definition 12, Op_2 is left-deep in the left sub-query Q_{RT} of Op_1, i.e., it is the left descendant of any $-$ operator on the path between Op_1 and Op_2.

reduction uses the following annotation functions:

$$\Phi_R : \mathbf{X} \cup \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_R(x_i) = \Phi_R(x_i, y_j) = x_i$$

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \top & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

$$\Phi_T : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_T(y_j) = \Phi_T(y_j, x_i) = y_j$$

$$\Phi_{RT} : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RT}(x_i, y_j) = x_i \wedge y_j$$

$$\Phi_{RST} : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RST}(x_i, y_j) = \begin{cases} x_i \wedge y_j & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

Relations R, S, T are filled by populating R's $[A]$-attributes with values $\mathbf{X}$ and annotations Φ_R, T's $[B]$-attributes with values $\mathbf{Y}$ and annotations Φ_T, and S with the cross product $\mathbf{X} \times \mathbf{Y}$ for attributes $[A]$ and $[B]$, and annotations Φ_S. All other attributes are set to ■. Using the notation introduced in Section 3.3, this can be written more concisely as

$$\begin{aligned} \text{red}_A(R) &= \mathbf{X}|^{\neg^{\text{pol}(Q_R,R)}}\Phi_R \\ \text{red}_{AB}(S) &= \mathbf{X} \times \mathbf{Y}|^{\neg^{\text{pol}(Q_S,S)}}\Phi_S \\ \text{red}_B(T) &= \mathbf{Y}|^{\neg^{\text{pol}(Q_T,T)}}\Phi_T \end{aligned}$$

Recall that $\mathrm{pol}(Q, R)$ identifies the even ($\mathrm{pol}(Q, R) = 0$) or odd ($\mathrm{pol}(Q, R) = 1$) polarity of relation symbol R in query Q (cf. Equation (2.8)); we use the convention $\neg^1 Q \equiv \neg Q$ and $\neg^0 Q \equiv Q$.

Figure 3.14 depicts the relations R, S, T. By applying the results of Section 3.3 and Lemma 12, the remaining relations in Q_R, Q_S, and Q_T can be filled such that $Q_R \in \mathcal{Q}^A$, $Q_T \in \mathcal{Q}^B$, and $Q_S \in \mathcal{Q}^{AB}$, i.e., the values and annotations of R, S, T are preserved in Q_R, Q_S, and Q_T.

Since the filling of R, S, T accounts for their polarity in their sub-queries Q_R, Q_S, Q_T, the latter relations take the following simple form[9]:

$$\mathrm{red}_A(Q_R) = \mathbf{X}|\Phi_R \qquad \mathrm{red}_{AB}(Q_S) = \mathbf{X} \times \mathbf{Y}|\Phi_S \qquad \mathrm{red}_B(Q_T) = \mathbf{Y}|\Phi_T$$

Moreover, the sub-query $Q_R \bowtie Q_T$ satisfies

$$\mathrm{red}_{AB}(Q_R \bowtie Q_T) = \mathbf{X} \times \mathbf{Y}|\Phi_{RT}.$$

By applying Lemma 12 on Q_{RT}, the relation represented by the sub-query $Q_R \bowtie Q_T$ can be preserved in Q_{RT}; note that since Op_2 is left-deep in Q_{RT}, Op_2 has even polarity in Q_{RT} and hence the annotations of tuples in Q_{RT} carry the same sign as in $Q_R \bowtie Q_T$. This yields

$$\mathrm{red}_{AB}(Q_{RT}) = \mathbf{X} \times \mathbf{Y}|\Phi_{RT} \qquad \mathrm{red}_{AB}(Q_S) = \mathbf{X} \times \mathbf{Y}|\Phi_S$$

and finally

$$\mathrm{red}_{AB}(Q_{RT} \bowtie QS) = \mathbf{X} \times \mathbf{Y}|\Phi_{RST}.$$

The sub-query rooted in Op_1 thus has exactly one tuple (x_i, y_j) annotated with $x_i \wedge y_j$ for each clause $x_i \wedge y_j$ of Ψ, and one tuple (x_i, y_j) annotated with $\bot$ for each combination of variables x_i, y_j that are not a clause in Ψ.

Since by Definition 12 there does not exist a $-$ operator above Op_1 that exports $[A]$ or $[B]$, we can use the techniques of Lemma 12 to fill the relations of Q that are not descendants of Op_1 such that these annotations are preserved by any operator above Op_1. Finally, since by Definition 10 Q does not export $[A]$ or $[B]$, those attributes are eventually projected out above Op_1, yielding as result a single tuple of ■ values annotated with the disjunction of the annotation of $Q_{RT} \bowtie Q_S$. If this projection is followed by $-$ operators (that *do not* export $[A]$ or $[B]$), then each of these $-$ operators will flip the sign of the annotation; to

[9]The match of Q with pattern $P_{1.1}$ does not prohibit Op_2 from expressing a join on $[B]$. In that case, the sub-queries of Op_2 are $Q_R^{[A][B]}$ and $Q_T^{[A][B]}$ and satisfy $\mathrm{red}_{AB}(Q_R) = \mathbf{X} \times \mathbf{Y}|\Phi_R$ and $\mathrm{red}_{AB}(Q_T) = \mathbf{Y} \times \mathbf{X}|\Phi_T$ in addition to the following reductions. Note that both the case of Op_2 carrying a join $[B]$ and the case of Op_2 not carrying this join are covered by the definition of $\mathcal{Q}^A$ in Equations (3.6) and (3.7).

$\mathcal{Q}^A$-relation R		
A_r	$[\neg A]$	$\mathbf{\Phi}$
x_1	■	$\neg^{\text{pol}}\mathbf{x_1}$
x_2	■	$\neg^{\text{pol}}\mathbf{x_2}$
$\cdots$		
$x_{\lvert X\rvert}$	■	$\neg^{\text{pol}}\mathbf{x}_{\lvert\mathbf{X}\rvert}$

$\mathcal{Q}^B$-relation T		
A_r	$[\neg A]$	$\mathbf{\Phi}$
y_1	■	$\neg^{\text{pol}}\mathbf{y_1}$
y_2	■	$\neg^{\text{pol}}\mathbf{y_2}$
$\cdots$		
$y_{\lvert Y\rvert}$	■	$\neg^{\text{pol}}\mathbf{y}_{\lvert\mathbf{Y}\rvert}$

$\mathcal{Q}^{AB}$-relation S			
A_s	B_s	$[\neg A], [\neg B]$	$\mathbf{\Phi}$
x_1	y_1	■	$\neg^{\text{pol}}\top$
x_1	y_2	■	$\neg^{\text{pol}}\top$
x_1	y_3	■	$\neg^{\text{pol}}\bot$
$\cdots$			
$x_{\lvert X\rvert}$	$y_{\lvert Y\rvert}$	■	$\neg^{\text{pol}}\bot$

Figure 3.14: Relations R, S, T for the hardness reduction of a query with a lineage-preserving match for pattern $P_{1.1}$. The filling of S assumes that the formula Ψ is $x_1y_1 \vee x_1y_2$ and thus tuples (x_1, y_1) and (x_1, y_2) are the only S-tuples annotated with $\top$ (assuming even polarity of S in Q_S). To avoid clutter, the full polarity function is omitted from the annotation columns; for relation R, $\neg^{\text{pol}}$ stands for $\neg^{\text{pol}(Q_R,R)}$, for T the polarity is $\neg^{\text{pol}(Q_T,T)}$, and for S it is $\neg^{\text{pol}(Q_S,S)}$.

summarise, query Q satisfies

$$\text{red}_\emptyset(Q) = ■|\neg^{\text{pol}(Q,Op_1)}\Psi$$

and we can conclude that the probability $P_{Q(\mathcal{D})}$ of query Q on the database $\mathcal{D}$ constructed above is either exactly P_Ψ or $1 - P_\Psi$.

Pattern $P_{1.2}$. A query that is a lineage-preserving match for $P_{1.2}$ has the same form as depicted in Figure 3.13, except that $Op_2 = -$. The annotation functions Φ_R, Φ_S, Φ_{RT}, and Φ_{RST} are as in the case of pattern $P_{1.1}$, and the annotation function Φ_T carries an additional negation to account for the flipped polarity of T (when compared to pattern $P_{1.1}$) due to the $-$ operator Op_2:

$$\Phi_T : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_T(y_j) = \Phi_T(y_j, x_i) = \neg y_j$$

We fill relations R, S, T as

$$\begin{aligned}\text{red}_A(R) &= \mathbf{X}|\neg^{\text{pol}(Q_R,R)}\Phi_R\\ \text{red}_{AB}(S) &= \mathbf{X} \times \mathbf{Y}|\neg^{\text{pol}(Q_S,S)}\Phi_S\\ \text{red}_B(T) &= \mathbf{Y}|\neg^{\text{pol}(Q_T,T)}\Phi_T\end{aligned}$$

By Lemma 15, it follows that Op_2 exports $[A]$ and $[B]$ and hence Q_R and Q_T export both $[A]$ and $[B]$. The sub-queries are the following relations:

$$\text{red}_A(Q_R) = \mathbf{X} \times \mathbf{Y}|\Phi_R \qquad \text{red}_{AB}(Q_S) = \mathbf{X} \times \mathbf{Y}|\Phi_S \qquad \text{red}_B(Q_T) = \mathbf{Y} \times \mathbf{X}|\Phi_T$$

The query $Q_R - Q_T$ thus satisfies

$$\text{red}_{AB}(Q_R - Q_T) = \mathbf{X} \times \mathbf{Y}|\Phi_{RT}$$

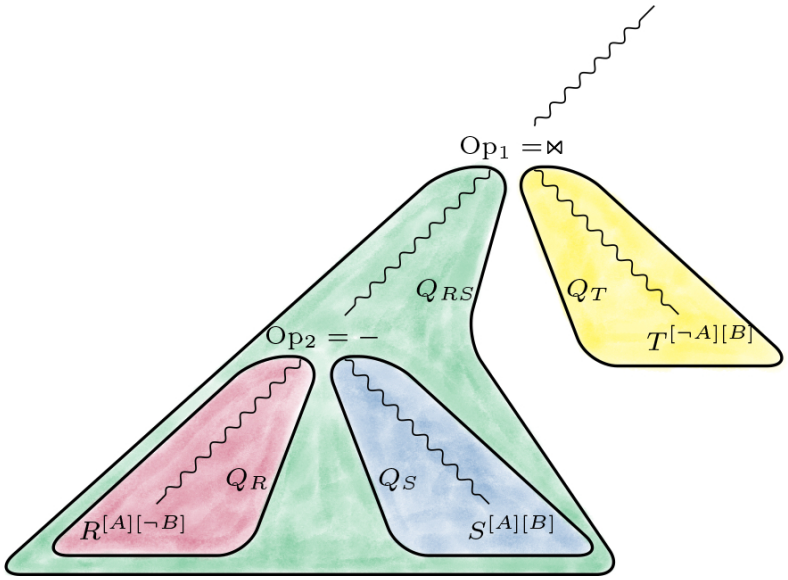


Figure 3.15: Schematic illustration of a query that is a lineage-preserving match for pattern $P_{2.2}$. A curly edge indicates that arbitrary other operators may occur on this path. By Definition 12, Op_2 is left-deep in the left sub-query Q_{RS} of Op_1, i.e., it is the left descendant of any $-$ operator on the path between Op_1 and Op_2.

and the remainder of the reduction is identical to the case of pattern $P_{1.1}$.

Pattern $P_{1.3}$. The reduction is identical to the case of $P_{1.1}$ except for the definition of Φ_S which carries an extra negation to account for the swapped polarity of S:

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \bot & \text{if } (i,j) \in E \\ \top & \text{if } (i,j) \notin E \end{cases}$$

Pattern $P_{1.4}$. The reduction is identical to the case of $P_{1.2}$ except for the definition of Φ_S which carries an extra negation to account for the swapped polarity of S:

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \bot & \text{if } (i,j) \in E \\ \top & \text{if } (i,j) \notin E \end{cases}$$

3.4.2.2 Reduction for patterns $P_{2.x}$

Let Q be a query that is lineage-preserving match for one of $P_{2.1}$ – $P_{2.4}$. Such a query is depicted in Figure 3.15 (for the case of $P_{2.2}$) and Q satisfies these structural constraints:

- By Lemma 15, any operator on the path R – Op_2 – S exports $[A]$, and every operator on the path S – Op_2 – Op_1 – T exports $[B]$. Operator Op_2 expresses an equality constraint on $[A]$, and operator Op_1 expresses an equality constraint (for patterns $P_{2.1}$ and $P_{2.2}$) or a difference constraint $B \leftrightarrow B'$ (for patterns $P_{2.3}$ and $P_{2.4}$) on $[B]$.

- If Op_1 is a right descendant of a $-$ operator, then this operator does not export $[A]$ or $[B]$.
- Operator Op_2 is left-deep in Q_{RS}, i.e., it is the left descendant of any $-$ operator on the path between Op_1 and Op_2.

Pattern $P_{2.2}$. Relations R, S, T are filled using Φ_R, Φ_S, and Φ_T exactly as in the case of pattern $P_{1.3}$. Additionally, define the following annotation functions:

$$\Phi_{RS} : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RS}(y_j, x_i) = \begin{cases} x_i & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

$$\Phi_{RS}(y_j) = \bigvee_{(i,j)\in E} x_i$$

$$\Phi_{RST} : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RST}(y_j, x_i) = \begin{cases} x_i \wedge y_j & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

$$\Phi_{RST}(y_j) = y_j \wedge \bigvee_{(i,j)\in E} x_i$$

Then $Q_R - Q_S$ and Q_{RS} are $\mathcal{Q}^B$-queries that satisfy

$$\mathrm{red}_B(Q_R - Q_S) = \mathbf{Y}|\Phi_{RS} \qquad \mathrm{red}_B(Q_{RS}) = \mathbf{Y}|\Phi_{RS}$$

and Q_T is a Q^B-query with

$$\mathrm{red}_B(Q_T) = \mathbf{Y}|\Phi_T$$

Finally, the join $Q_{RS} \bowtie Q_T$ satisfies

$$\mathrm{red}_B(Q_{RS} \bowtie Q_T) = \mathbf{Y}|\Phi_{RST}$$

and the reasoning about operators above Op_1 is as in the case of pattern $P_{1.1}$; the eventual projection $\pi_{-[B]}$ yields

$$\mathrm{red}_\emptyset(Q) = \blacksquare|\neg^{\mathrm{pol}(Q,Op_1)}\Psi$$

for the annotation of query Q.

Pattern $P_{2.3}$. Fill the relations R, S, T as in the case of pattern $P_{1.2}$; the remaining analysis is then identical to $P_{2.2}$.

Pattern $P_{2.4}$. This case is identical to $P_{2.2}$ if relation T is filled according to annotation function

$$\Phi_T : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_T(y_j) = \Phi_T(y_j, x_i) = \neg y_j$$

3.4.2.3 Reduction for patterns $P_{3.x}$

The structure of queries matching any of the pattern $P_{3.x}$ is equivalent to those matching a pattern $P_{2.x}$ with relations R and S swapped. Also the structural constraint are identical and hence the reductions are very similar.

Pattern $P_{3.2}$. This case is similar to that of $P_{2.2}$ when the signs of Φ_R and Φ_S are chosen as follows:

$$\Phi_R : \mathbf{X} \cup \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_R(x_i) = \Phi_R(x_i, y_j) = \neg x_i$$

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \top & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

Then with Φ_{RS} as in the case of $P_{2.2}$, sub-queries $Q_S - Q_R$ and Q_{RS} are $\mathcal{Q}^B$-queries and satisfy

$$\mathrm{red}_B(Q_S - Q_R) = \mathbf{Y}|\Phi_{RS} \qquad \mathrm{red}_B(Q_{RS}) = \mathbf{Y}|\Phi_{RS}$$

The remainder of this reduction is identical to the case of $P_{2.2}$.

Pattern $P_{3.4}$. This case is identical to the case of $P_{2.4}$ when the annotation functions Φ_R and Φ_S from $P_{3.2}$ are used.

3.4.2.4 Reduction for patterns $P_{4.x}$

By virtue of Definition 12 (cf. also Figure 3.16), a query Q that has a lineage-preserving match with one of $P_{4.3}$, $P_{4.4}$, $P_{5.3}$, $P_{5.4}$, or $P_{6.4}$ satisfies the following structural constraint: If Op_1 is a right descendant of a $-$ operator, then this operator does not export $[A]$ or $[B]$. Furthermore, for patterns $P_{4.x}$, attributes $[A]$ and $[B]$ are exported by every operator on the paths from S to R and S to T, respectively.

For queries matching a pattern $P_{4.x}$, it is only possible to directly encode the 2DNF formula Ψ as a database $\mathcal{D}$ such that the annotation of $Q(\mathcal{D})$ is exactly Ψ, if the polarity of Op_2 is odd in Q_{RT}. In the case of even polarity, we show that we can derive a database $\mathcal{D}$

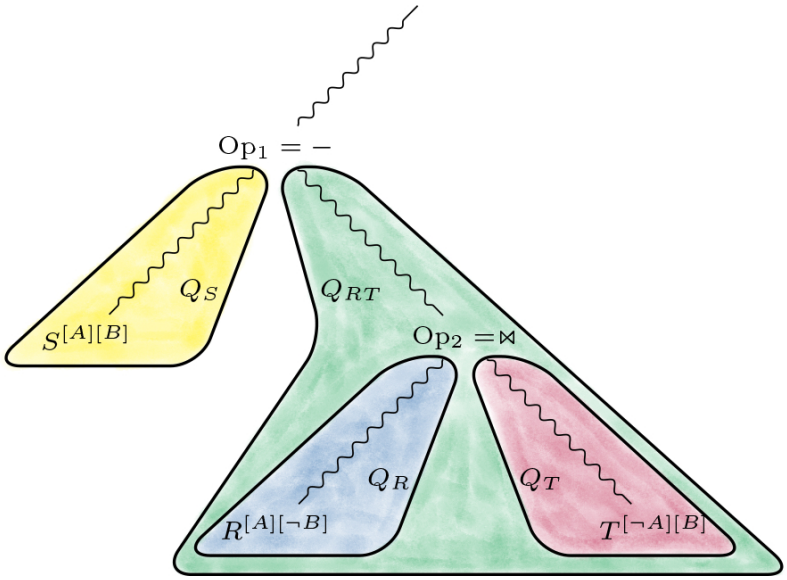


Figure 3.16: Schematic illustration of a query that is a lineage-preserving match for pattern $P_{4.3}$. A curly edge indicates that arbitrary other operators may occur on this path.

and another formula Υ from Ψ such that $P_{Q(\mathcal{D})} = P_\Upsilon$ and linearly many calls to an oracle for P_Υ suffice to determine $\#\Psi$.

As before, let $\Psi = \psi_1 \vee \dots \psi_m$ be a 2DNF formula over variables $\mathbf{X}$ and $\mathbf{Y}$ with m clauses. Let Θ be the set of assignments of variables $\mathbf{X} \cup \mathbf{Y}$. Then the number of models of Ψ is defined by $\#\Psi = \sum_{\theta \in \Theta : \theta \models \Psi} 1$. If we partition Θ into disjoint sets $\Theta_0 \cup \dots \cup \Theta_m$, such that $\theta \in \Theta_i$ if and only if θ satisfies exactly i clauses of Ψ, then this sum can equivalently by written as

$$\#\Psi = \sum_{\theta \in \Theta_1 : \theta \models \Psi} 1 + \dots + \sum_{\theta \in \Theta_m : \theta \models \Psi} 1 = |\Theta_1| + \dots + |\Theta_m|.$$

We will next show how to compute the numbers $|\Theta_i|$ (and hence $\#\Psi$) using an oracle for P_Υ.

Let $\mathbf{Z} = \{z_1, \dots, z_m\}$ be a set of variables disjoint from $\mathbf{X}$ and $\mathbf{Y}$ and define formula Υ as

$$\Upsilon = \bigvee_{i=1}^{m} \neg z_i \wedge \neg \psi_i \qquad \text{or, equivalently} \qquad \neg\Upsilon = \bigwedge_{i=1}^{m} (z_i \vee \psi_i) \tag{3.13}$$

We fix the probabilities of variables $\mathbf{X}$ and $\mathbf{Y}$ to 0.5 and of variables $\mathbf{Z}$ to $p_z \in [0, 1]$ for being true. The probability $P_{\neg\Upsilon}$ can be expanded by conditioning on the number of satisfied

clauses of Ψ:

$$1 - P_\Upsilon = P_{\neg\Upsilon} = \sum_{k=0}^{m} \underbrace{P\left(\neg\Upsilon \,\middle|\, \begin{matrix}\text{exactly } k \text{ clauses}\\ \text{of } \Psi \text{ are satisfied}\end{matrix}\right)}_{p_z^{m-k}} \cdot \underbrace{P\left(\begin{matrix}\text{exactly } k \text{ clauses}\\ \text{of } \Psi \text{ are satisfied}\end{matrix}\right)}_{\frac{1}{2}^{|\mathbf{X}|+|\mathbf{Y}|} \cdot |\Theta_k|}$$

$$= \frac{1}{2}^{|\mathbf{X}|+|\mathbf{Y}|} \sum_{k=0}^{m} p_z^{m-k} |\Theta_k|$$

Intuitively, the first term simplifies to p_z^{m-k}, because if exactly k clauses ψ_i are satisfied in $\neg\Upsilon$, then in order to satisfy the remaining $m-k$ clauses $(z_i \vee \psi_i)$ at least $m-k$ of the z_i must be satisfied, and this occurs with probability p_z^{m-k}.

This is a polynomial in p_z of degree m, with coefficients $|\Theta_0|, \ldots, |\Theta_m|$. The $m+1$ coefficients can be derived from $m+1$ pairs (p_z, P_Υ) using Lagrange's polynomial interpolation formula. We can conclude that $m+1$ oracle calls to P_Υ suffice to determine $\#\Psi = \sum_{i=0}^{m} |\Theta_i|$.

It remains to be shown how Υ can be encoded as the annotation of a query that has a lineage-preserving match to one of the patterns $P_{4.x}$; given this encoding, any algorithm that evaluates $P_{Q(\mathcal{D})}$ constitutes the above oracle. We give encodings for the two patterns $P_{4.3}$ and $P_{4.4}$ separately.

Pattern $P_{4.3}$. As indicated above, we need to distinguish two cases by the polarity of Op_2 in the sub-query Q_{RT}: If the polarity is odd, we can use a filling similar to that of pattern $P_{1.1}$; if it is even we fill to obtain formula Υ as outlined above.

Case 1: Odd polarity ($\mathrm{pol}(Q_{RT}, Op_2) = 1$). We fill R, S, T like in the case of pattern $P_{1.1}$ according to annotation functions

$$\Phi_R : \mathbf{X} \cup \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_R(x_i) = \Phi_R(x_i, y_j) = x_i$$

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \top & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

$$\Phi_T : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_T(y_j) = \Phi_T(y_j, x_i) = y_j$$

$$\Phi_{RT} : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RT}(x_i, y_j) = x_i \wedge y_j$$

$$\Phi_{RST} : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RST}(x_i, y_j) = \begin{cases} x_i \wedge y_j & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

Note that since Op_2 has odd polarity in Q_{RT}, and since both $[A]$ and $[B]$ are exported by every operator on the path between Op_1 and Op_2, Q_{RT} is a $\mathcal{Q}^{AB}$-query with annotations

$$\mathrm{red}_{AB}(Q_{RT}) = \mathbf{X} \times \mathbf{Y} | \neg \Phi_{RT}.$$

and hence $Q_S - Q_{RT}$ is a $\mathcal{Q}^{AB}$ query with

$$\mathrm{red}_{AB}(Q_S - Q_{RT}) = \mathbf{X} \times \mathbf{Y} | \Phi_{RST}.$$

The remainder of this reduction is identical to the case of pattern $P_{1.1}$.

Case 2: Even polarity ($\mathrm{pol}(Q_{RT}, Op_2) = 0$). In this case, the goal is to encode the formula Υ as introduced in Equation (3.13). Using the annotation functions from above modified to

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \neg z_k & \text{if } x_i \wedge y_j \text{ is clause } \psi_k \text{ in } \Psi \\ \bot & \text{else} \end{cases}$$

$$\Phi_{RST} : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \quad \Phi_{RST}(x_i, y_j) = \begin{cases} \neg z_k \wedge \neg \psi_k & \text{if } x_i \wedge y_j \text{ is clause } \psi_k \text{ in } \Psi \\ \bot & \text{else} \end{cases}$$

we obtain

$$\mathrm{red}_{AB}(Q_{RT}) = \mathbf{X} \times \mathbf{Y} | \Phi_{RT}$$
$$\mathrm{red}_{AB}(Q_S - Q_{RT}) = \mathbf{X} \times \mathbf{Y} | \Phi_{RST}$$

since Op_1 has even polarity in Q_{RT}. As before, the eventual projection $\pi_{-[A]-[B]}$ yields one answer tuple whose annotation is the disjunction of the annotation of $Q_S - Q_{RT}$ which is exactly Υ, cf. Equation (3.13).

Pattern $P_{4.4}$. The reduction is equivalent to the case of pattern $P_{4.3}$ when the sign of the annotation functions Φ_T is flipped (in both the case $\mathrm{pol}(Q_{RT}, Op_2) = 1$ and the case $\mathrm{pol}(Q_{RT}, Op_2) = 0$).

3.4.2.5 Reduction for patterns $P_{5.x}$

Pattern $P_{5.3}$. We first distinguish two cases: $[B]$ is exported by Op_1, and $[B]$ is not exported by Op_1:

If $[B]$ is exported by Op_1. Without loss of generality, assume that Op_1 is the *first* operator that allows for a match by virtue of Lemma 16. Then Q_R contains a relation X

that exports $[B]$ and is joined with R in Q_R. If this relation is $X^{[A][B]}$, then Q is a lineage-preserving (R, X, T)-match for one of the patterns $P_{2.x}$ or $P_{3.x}$; if this relation is $X^{[\neg A][B]}$, then Q is a lineage-preserving (R, S, X)-match for one of the patterns $P_{1.x}$. In both cases, we proceed according to the above cases for the matched pattern.

If $[B]$ is not exported by Op_1. The sub-query Q_{ST} contains a projection operator $Op_\pi = \pi_{-[B]}$ such that every operator between Op_π and Op_1 exports $[A]$ but not $[B]$, and every operator between Op_π and Op_2 exports $[A]$ and $[B]$. Let Q_π be the sub-query rooted in Op_π.

The first step is to show that one may without loss of generality assume that Op_2 is left-deep in Q_π. Assume to the contrary that there is a $-$ operator Op_- between Op_π and Op_2 that has Op_2 as a right descendant; clearly, Op_- exports $[A]$ and $[B]$ and hence its left sub-query contains relations $X^{[A][\neg B]}$ and $Y^{[\neg A][B]}$ or it contains a relation $X^{[A][B]}$. In the former case, Q is a lineage-preserving (R, S, Y)-match for pattern $P_{5.4}$ in which Op_2 is left-deep in Q_π, and the latter case, Q is a lineage-preserving (R, X, T)-match for pattern $P_{6.4}$ in which Op_2 is left-deep in Q_π.

Next, two cases need to be analysed separately depending on the polarity of Op_π in Q_{ST}:

Case 1: Even polarity ($\mathrm{pol}(Q_{ST}, Op_\pi) = 0$). Let $\mathbf{V} = \mathbf{X} \cup \mathbf{Y}$ and $\mathbf{N} = \{1, \ldots, |E|\}$ be the set of integers that numbers consecutively $\Psi's$ clauses: $\Psi = \psi_1 \vee \cdots \vee \psi_{|E|}$. We use the following annotation functions:

$$\Phi_R : \mathbf{N} \to \mathcal{B}(\mathbf{V}) \qquad \Phi_R(i) = \top$$

$$\Phi_S : \mathbf{N} \times \mathbf{V} \to \mathcal{B}(\mathbf{V}) \qquad \Phi_S(i, v) = \begin{cases} \top & \text{if clause } \psi_i \text{ contains variable } v \\ \bot & \text{else} \end{cases}$$

$$\Phi_T : \mathbf{V} \cup \mathbf{V} \times \mathbf{N} \to \mathcal{B}(\mathbf{V}) \qquad \Phi_T(v) = \Phi_T(v, i) = \neg v$$

$$\Phi_{ST} : \mathbf{N} \times \mathbf{V} \to \mathcal{B}(\mathbf{V}) \qquad \Phi_{ST}(i, v) = \begin{cases} \neg v & \text{if clause } \psi_i \text{ contains variable } v \\ \bot & \text{else} \end{cases}$$

$$\Phi_{\pi ST} : \mathbf{N} \to \mathcal{B}(\mathbf{V}) \qquad \Phi_{\pi ST}(i) = \neg\psi_i$$

$$\Phi_{RST} : \mathbf{N} \to \mathcal{B}(\mathbf{V}) \qquad \Phi_{RST}(i) = \psi_i$$

Relations R, S, T are filled as before, accounting for their respective polarity in Q_R, Q_S, Q_T. Then $Q_T \bowtie Q_S$ is a $\mathcal{Q}^{AB}$-relation that satisfies

$$\mathrm{red}_{AB}(Q_T \bowtie Q_S) = \mathbf{N} \times \mathbf{V} | \Phi_{ST}$$

R	
A_r	$\mathbf{\Phi}$
1	$\top$
2	$\top$

T	
B_t	$\mathbf{\Phi}$
x_1	$\neg\mathbf{x_1}$
y_1	$\neg\mathbf{y_1}$
y_2	$\neg\mathbf{y_2}$

S		
A_s	B_s	$\mathbf{\Phi}$
1	x_1	$\top$
1	y_1	$\top$
1	y_2	$\bot$
2	x_1	$\top$
2	y_1	$\bot$
2	y_2	$\top$

$Q_T \ltimes Q_S$		
A_s	B_s	$\mathbf{\Phi}$
1	x_1	$\neg\mathbf{x_1}$
1	y_1	$\neg\mathbf{y_1}$
1	y_2	$\bot$
2	x_1	$\neg\mathbf{x_1}$
2	y_1	$\bot$
2	y_2	$\neg\mathbf{y_2}$

$Q_\pi = Q_{ST}$	
A_s	$\mathbf{\Phi}$
1	$\neg\mathbf{x_1} \vee \neg\mathbf{y_1}$
2	$\neg\mathbf{x_1} \vee \neg\mathbf{y_2}$

Q_{RST}	
A_r	$\mathbf{\Phi}$
1	$\mathbf{x_1 y_1}$
2	$\mathbf{x_1 y_2}$

Figure 3.17: Relations R, S, T for the hardness reduction of a query with a lineage-preserving match for pattern $P_{5.3}$ where (1) Op_1 does not export $[B]$ and (2) the projection operator $\pi_{-[B]}$ on the path between Op_1 and Op_2 has even polarity in Q_{ST}. Only attributes $[A]$ and $[B]$ are depicted, and to avoid clutter it is assumed that R, S, T have even polarity in their respective sub-queries Q_R, Q_S, and Q_T. The filling is for the formula $\Psi = \psi_1 \vee \psi_2 = x_1y_1 \vee x_1y_2$.

Operator Op_π turns the $\mathcal{Q}_{AB}$-relation $Q_T \ltimes Q_S$ into a $\mathcal{Q}^A$-relation Q_{ST} with

$$\mathrm{red}_A(Q_\pi) = \mathbf{N}|\Phi_{\pi ST}$$

and since Op_π has even polarity in Q_{ST}, this annotation can be preserved for Q_{ST}:

$$\mathrm{red}_A(Q_{ST}) = \mathbf{N}|\Phi_{\pi ST}$$

Finally, the annotations of R can be preserved in Q_R and the sub-query $Q_R - Q_{ST}$ flips the sign of the annotations of Q_{ST}; this yields

$$\mathrm{red}_A(Q_R) = \mathbf{N}|\Phi_R$$
$$\mathrm{red}_A(Q_{RST}) = \mathbf{N}|\Phi_{RST}$$

As in the previous cases, the eventual projection $\pi_{-[A]}$ reconstructs Ψ:

$$\mathrm{red}_\emptyset(Q) = \blacksquare|\neg^{\mathrm{pol}(Q,Op_1)}\Psi$$

Example 20. Figure 3.17 indicates how R, S, T are filled for a query matching pattern $P_{5.3}$ and formula $\Psi = x_1y_1 \vee x_1y_2$ and how annotations are propagated through the query. □

Case 2: Odd polarity ($\mathrm{pol}(Q_{ST}, Op_\pi) = 1$)**.** Using the annotation functions

$$\Phi_R : \mathbf{X} \cup \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_R(x_i) = \Phi_R(x_i, y_j) = x_i$$

$$\Phi_S : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_S(x_i, y_j) = \begin{cases} \top & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

$$\Phi_T : \mathbf{Y} \cup \mathbf{Y} \times \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_T(y_i) = \Phi_T(y_j, x_i) = y_i$$

$$\Phi_{ST} : \mathbf{X} \times \mathbf{Y} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{ST}(x_i, y_j) = \begin{cases} y_j & \text{if } (i,j) \in E \\ \bot & \text{if } (i,j) \notin E \end{cases}$$

$$\Phi_{\pi ST} : \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{\pi ST}(x_i) = \bigvee_{(i,j) \in E} y_j$$

$$\Phi_{RST} : \mathbf{X} \to \mathcal{B}(\mathbf{X} \cup \mathbf{Y}) \qquad \Phi_{RST}(x_i) = x_i \wedge \bigvee_{(i,j) \in E} y_j$$

one obtains the following reductions

$$\mathrm{red}_A(Q_R) = \mathbf{X}|\Phi_R \qquad \mathrm{red}_{AB}(Q_T \bowtie Q_S) = \mathbf{X} \times \mathbf{Y}|\Phi_{ST}$$

$$\mathrm{red}_{AB}(Q_S) = \mathbf{X} \times \mathbf{Y}|\Phi_S \qquad \mathrm{red}_A(Q_\pi) = \mathbf{X}|\Phi_{\pi ST}$$

$$\mathrm{red}_B(Q_T) = \mathbf{Y}|\Phi_T \qquad \mathrm{red}_A(Q_{ST}) = \mathbf{X}|\neg\Phi_{\pi ST}$$

where the sign of the annotations of $\mathrm{red}_A(Q_\pi)$ and $\mathrm{red}_A(Q_{ST})$ is flipped because Op_π has odd polarity in Q_{ST}. This yields

$$\mathrm{red}_A(Q_{RST}) = \mathbf{X}|\Phi_{RST}$$

for the sub-query rooted in $Op_1 = -$ and

$$\mathrm{red}_\emptyset(Q) = \blacksquare|\neg^{\mathrm{pol}(Q, Op_1)}\Psi$$

for the annotation of query Q.

Pattern $P_{5.4}$. The analysis of pattern $P_{5.4}$ is analogous to the case of pattern $P_{5.3}$ where the sign of the annotation function Φ_S is flipped.

3.4.2.6 Reduction for patterns $P_{6.x}$

Pattern $P_{6.4}$. The analysis of pattern $P_{6.4}$ is analogous to the case of pattern $P_{5.3}$ where the sign of the annotation function Φ_T is flipped.

3.5 Beyond non-repeating queries: The tractability frontier for quantified queries

This section investigates the data complexity of the probabilistic query evaluation problem for quantified queries that express binary relationships amongst sets of entities. These queries can be expressed in relational algebra using nested negation and repeated relation symbols. For example, a set-inclusion query could find non-critical overseas suppliers, i.e., overseas suppliers for parts that also have domestic suppliers. We define such queries in relational algebra and analyse their data complexity with respect to tuple-independent databases: For tractable queries, we give an explicit $\mathcal{O}(|\mathcal{D}|)$-algorithm for computing the probability of the query annotation based on Shannon expansion and the inclusion-exclusion principle; for intractable queries, we detail a hardness reduction from #2DNF.

Tractable quantified queries

S

sid	item	$\mathbf{\Phi}$
1	a	$\mathbf{x_1}$
1	b	$\mathbf{x_2}$
1	c	$\mathbf{x_3}$
2	a	$\mathbf{y_1}$
2	b	$\mathbf{y_2}$
3	a	$\mathbf{z_1}$
3	c	$\mathbf{z_2}$

$S_{\subseteq}$

s_1	s_2	$\mathbf{\Phi}$
1	1	$(\mathbf{x_1} \vee \mathbf{x_2} \vee \mathbf{x_3})$
1	2	$(\mathbf{x_1} \vee \mathbf{x_2} \vee \mathbf{x_3})(\mathbf{y_1} \vee \mathbf{y_2})\neg(\mathbf{x_1}\neg\mathbf{y_1} \vee \mathbf{x_2}\neg\mathbf{y_2} \vee \mathbf{x_3})$
1	3	$(\mathbf{x_1} \vee \mathbf{x_2} \vee \mathbf{x_3})(\mathbf{z_1} \vee \mathbf{z_2})\neg(\mathbf{x_1}\neg\mathbf{z_1} \vee \mathbf{x_2} \vee \mathbf{x_3}\neg\mathbf{z_2})$
2	1	$(\mathbf{y_1} \vee \mathbf{y_2})(\mathbf{x_1} \vee \mathbf{x_2} \vee \mathbf{x_3})\neg(\mathbf{y_1}\neg\mathbf{x_1} \vee \mathbf{y_2}\neg\mathbf{x_2})$
2	2	$(\mathbf{y_1} \vee \mathbf{y_2})$
2	3	$(\mathbf{y_1} \vee \mathbf{y_2})(\mathbf{z_1} \vee \mathbf{z_2})\neg(\mathbf{y_1}\neg\mathbf{z_1} \vee \mathbf{y_2})$
3	1	$(\mathbf{z_1} \vee \mathbf{z_2})(\mathbf{x_1} \vee \mathbf{x_2} \vee \mathbf{x_3})\neg(\mathbf{z_1}\neg\mathbf{x_1} \vee \mathbf{z_2}\neg\mathbf{x_3})$
3	2	$(\mathbf{z_1} \vee \mathbf{z_2})(\mathbf{y_1} \vee \mathbf{y_2})\neg(\mathbf{z_1}\neg\mathbf{y_1} \vee \mathbf{z_2})$
3	3	$(\mathbf{z_1} \vee \mathbf{z_2})$

$S_{=}$

s_1	s_2	Φ
1	1	$\mathbf{\Phi(s_1 \subseteq s_1)}$
1	2	$\mathbf{\Phi(s_1 \subseteq s_2)} \wedge \mathbf{\Phi(s_2 \subseteq s_1)}$
1	3	$\mathbf{\Phi(s_1 \subseteq s_3)} \wedge \mathbf{\Phi(s_3 \subseteq s_1)}$
2	2	$\mathbf{\Phi(s_2 \subseteq s_2)}$
2	3	$\mathbf{\Phi(s_2 \subseteq s_3)} \wedge \mathbf{\Phi(s_3 \subseteq s_2)}$
3	3	$\mathbf{\Phi(s_3 \subseteq s_3)}$

Symmetric cases omitted

$$
\begin{aligned}
S_{\subseteq} &= \pi_{s_1}\delta_{\text{sid}\to s_1}(S) \bowtie \pi_{s_2}\delta_{\text{sid}\to s_2}(S) - \pi_{s_1,s_2}\big(\delta_{\text{sid}\to s_1}(S) \bowtie \pi_{s_2}\delta_{\text{sid}\to s_2}(S) - \delta_{\text{sid}\to s_2}(S) \bowtie \pi_{s_1}\delta_{\text{sid}\to s_1}(S)\big)\\
S_{=} &= \pi_{s_1,s_2}[S_{\subseteq} \bowtie_{s_1=s_4\wedge s_2=s_3} \delta_{s_1\to s_3,s_2\to s_4}(S_{\subseteq})]\\
S_{\neg} &= [\pi_{s_1}(\delta_{\text{sid}\to s_1}(S)) \bowtie \pi_{s_2}(\delta_{\text{sid}\to s_2}(S))] - S_{\subseteq}\\
S_{<>} &= \pi_{s_1,s_2}[S_{\neg} \bowtie_{s_1=s_4\wedge s_2=s_3} \delta_{s_1\to s_3,s_2\to s_4}(S_{\neg})]\\
S_{d=} &= \sigma_{s_1\neq s_2}S_{=}\\
S_{d\subseteq} &= \sigma_{s_1\neq s_2}S_{\subseteq}\\
S_{\subset} &= S_{\subseteq} - S_{=}
\end{aligned}
$$

Figure 3.18: Definition of queries for computing set inclusion, equality, and incomparability. The tables show an example database (S) and the result of applying the set inclusion ($S_{\subseteq}$) and the set equality ($S_{=}$) query. In the table for query $S_{=}$, $\Phi(s_i \subseteq s_j)$ is the annotation of the tuple (i, j) in relation $S_{\subseteq}$.

Assume we are given a tuple-independent relation S with schema $S(\text{sid}, \text{item})$ that specifies pairs of set identifiers and items in these sets. We would like to compute the pairs of set identifiers (s_1, s_2) and their probabilities, such that s_1 is included in/strictly included in/equal to/incomparable with s_2. The corresponding queries are denoted by $S_{\subseteq}$, $S_{\subset}$, $S_{=}$, and $S_{<>}$ respectively, and defined in Figure 3.18. In addition, $S_{d=}$ and $S_{d\subseteq}$ are the restrictions of $S_{=}$ and $S_{\subseteq}$ to pairs of different set identifiers.

Example 21. Relation S from Figure 3.18 defines three uncertain subsets of $\{a, b, c\}$: $s_1 = \{a, b, c\}$, $s_2 = \{a, b\}$, and $s_3 = \{a, c\}$. In the absence of uncertainty, we have that $s_2 \subset s_1$ and $s_3 \subset s_1$, and s_2 and s_3 are incomparable. Under the possible worlds semantics, however, further relationships may hold between the sets. For instance, the set s_1 is included in s_2 for those valuations of the random variables that satisfy the annotation associated with $(1, 2)$ in $S_{\subseteq}$. This annotation reads as follows. If a or b are in s_1, then they must also be in s_2; this is expressed by the term $\neg(x_1 \neg y_1 \vee x_2 \neg y_2)$. If c is in s_1, then $(1, 2)$ may not be in $S_{\subseteq}$; this is expressed by $\neg x_3$. The disjunctions $x_1 \vee x_2 \vee x_3$ and $y_1 \vee y_2$ ensure that the two sets have at least one item and are thus recorded in S. The disjunction $y_1 \vee y_2$ is redundant but shown for a better understanding. The direct application of the translation $[\![S_{\subseteq}]\!]$ according to Algorithm 1 yields an equivalent yet syntactically slightly different annotation than that depicted in Figure 3.18. □

Remark 7. Since a set is always equal to itself, one would expect that (i, i) occurs in $S_{=}$ with probability 1 for all sets i from S. However, Figure 3.18 states that the annotation of $(1, 1) \in S_{=}$ is $x_1 \vee x_2 \vee x_3$, whose probability is not always 1. This is correct due to the closed world assumption in relational databases: In the worlds in which the annotation is false, there is no set with id 1 and hence the set-inclusion query cannot produce pairs involving this set. □

The probability that a pair of set identifiers is in the answer to a quantified query from Figure 3.18 can be computed efficiently.

Theorem 18. *Let S be a tuple-independent relation over schema $S(sid, item)$ that defines sets and their items, and $S_{\subseteq}$, $S_{d\subseteq}$, $S_{\subset}$, $S_{=}$, $S_{d=}$, and $S_{<>}$ be the queries defined in Figure 3.18. Any tuple in the answer to these queries has an annotation of size $\mathcal{O}(|S|)$ and its probability can be computed in time $\mathcal{O}(|S|)$.*

Proof. The probabilities of the annotations for such queries can be computed efficiently using the inclusion-exclusion principle. We show an alternative approach for $S_{\subseteq}$ using recurrences and Shannon expansion. The annotations associated with tuples in $S_{\subseteq}$ (cf. Example 21) have the general form

$$(x_1 \vee \ldots \vee x_m)(y_1 \vee \ldots \vee y_n)\neg(x_1 \neg y_1 \vee \ldots \vee x_k \neg y_k \vee x_{k+1} \vee \ldots \vee x_m),$$

where k represents the number of items the two sets have in common, and $m - k$ is the number of items in the first set and not in the second. It is easy to see that this is equivalent to $(x_1 \vee \ldots \vee x_k)(y_1 \vee \ldots \vee y_n)\neg(x_1 \neg y_1 \vee \ldots \vee x_k \neg y_k)\neg(x_{k+1} \vee \ldots \vee x_m)$, and since the variables in the last negated disjunction occur only once, we can compute the probability of this disjunction efficiently and separately from the rest. We are thus left with

$(x_1 \vee \ldots \vee x_k)(y_1 \vee \ldots \vee y_n)\neg(x_1\neg y_1 \vee \ldots \vee x_k\neg y_k)$. Let

$$\begin{aligned}\Sigma_i^{xy} &= \neg(x_i\neg y_i \vee \ldots \vee x_k\neg y_k),\\ \Sigma_i^{x,xy} &= (x_i \vee \ldots \vee x_k)\Sigma_i^{xy},\\ \Sigma_i^{x,y,xy} &= (y_i \vee \ldots \vee y_n)\Sigma_i^{x,xy}.\end{aligned}$$

We then have the following for any i with $1 \leq i < k$:

$$\begin{aligned}\Sigma_i^{xy} &= (x_iy_i \vee \neg x_i)\Sigma_{i+1}^{xy}\\ \Sigma_i^{x,xy} &= x_iy_i\Sigma_{i+1}^{xy} \vee \neg x_i\Sigma_{i+1}^{x,xy}\\ \Sigma_i^{x,y,xy} &= x_iy_i\Sigma_{i+1}^{xy} \vee \neg x_i[y_i\Sigma_{i+1}^{x,xy} \vee \neg y_i\Sigma_{i+1}^{x,y,xy}]\end{aligned}$$

Each formula has a constant number of variables and refers to at most three formulas where one pair of variables (x_i, y_i) is removed. The recursion depth is thus bounded in the number of variables in S. Given the probabilities for the referred formulas, the probability of each referring formula can be computed efficiently, since all terms in the sums are pairwise mutually exclusive. We thus have $\mathcal{O}(|S|)$ time complexity for probability computation of $\Sigma_1^{x,y,xy}$ and of annotations in $S_{\subseteq}$.

Similar recurrences can be obtained for $S_{=}$ and $S_{<>}$ under the same variable order elimination. □

We next discuss the case of relational division. In the TPC-H scenario, a useful query with division would find the most likely suppliers for *all* parts of a given brand, see Figure 3.5. Similar to set-relation queries, we can use recurrence formulas to obtain a linear-time algorithm for computing the probabilities of tuples in the answer to a division query.

Theorem 19. *Let $T = R \div S$, where R and S are any tuple-independent relations. Then, any tuple in T has an annotation of size $\mathcal{O}(|R| + |S|)$ and its probability can be computed in time $\mathcal{O}(|R| + |S|)$.*

Proof. Let $R(\bar{A}, \bar{B})$ and $S(\bar{B})$ be the schemas of R and S, respectively. The schema of T is thus $T(\bar{A})$. Let $\{y_1, \ldots, y_n\}$ and $\{x_1, \ldots, x_m\}$ be the variables associated with tuples in S and with tuples $(\bar{a}, \bar{b})$ in R for a value $\bar{a}$, respectively; the following analysis applies separately to each value $\bar{a}$ in R.

The annotation of $\bar{a}$ in T has the form $(x_1\vee\ldots\vee x_m)\neg(y_1\neg x_1\vee\ldots\vee y_k\neg x_k\vee y_{k+1}\vee\ldots\vee y_n)$, where $k \leq m$ is such that $x_1, \ldots, x_k$ are those variables associated with tuples $(\bar{a}, \bar{b})$ in R where $\bar{b}$ is in S. The term $\neg(y_{k+1} \vee \ldots \vee y_n)$ can be factored out and its probability computed efficiently since it is a sum of independent variables that do not occur elsewhere

in the annotation. We are left with $(x_1 \vee \ldots \vee x_m)\neg(y_1\neg x_1 \vee \ldots \vee y_k\neg x_k)$. Let

$$\begin{aligned}\Sigma_i^x &= (x_i \vee \ldots \vee x_m)\\ \Sigma_i^{xy} &= \neg(y_i\neg x_i \vee \ldots \vee y_k\neg x_k)\\ \Sigma_i^{x,xy} &= \Sigma_i^x\Sigma_i^{xy}.\end{aligned}$$

Using Shannon expansion (cf. page 6), we can decompose them as follows:

$$\begin{aligned}\Sigma_i^x &= x_i \vee \neg x_i\Sigma_{i+1}^x\\ \Sigma_i^{xy} &= [x_i \vee \neg x_i\neg y_i]\Sigma_{i+1}^{xy}\\ \Sigma_i^{x,xy} &= x_i\Sigma_{i+1}^{xy} \vee \neg x_i\neg y_i\Sigma_{i+1}^{x,xy}.\end{aligned}$$

These recurrence formulas share the properties of those for set-relation queries: They can be computed in constant time given the values of the referred formulas, and the referred formulas have at least one variable less than the referring one. □

Remark 8. All recurrence formulas for our quantified queries use the same variable order for Shannon expansion: $x_1, y_1, \ldots, x_k, y_k$.

Intractable quantified queries

Some of the queries discussed in Section 3.5 become $\#P$-hard when one or more of their attributes are projected out:

Theorem 20. *For any $x \subset \{s_1, s_2\}$, the data complexity of the queries $\pi_\emptyset(R \div S)$, $\pi_x(S_{d=})$, $\pi_x(S_{d\subseteq})$, $\pi_x(S_\subset)$, and $\pi_x(S_{<>})$ is $\#P$-hard.*

Proof. As before, the proof is by direct reduction from the model counting problem for 2DNF formulas. We detail the reduction for the case of $\pi_\emptyset(S \div P)$, cf. Figure 3.5; the reductions for the remaining queries are analogous. Let S be deterministic, that is, all tuples in S are annotated with $\top$. We construct P such that the negation of each variable in the given 2DNF formula Φ annotates one distinct tuple in P. For each clause $c_{ij} = y_iy_j$ in Φ, we construct tuples in S with the same sid and with the pid values of those variables in P that are not in c_{ij}. By this construction, the query annotation becomes Φ. □

Although Boolean relational division $\pi_\emptyset(S \div P)$ is hard in general, its tractability depends on the input probability distribution in case each tuple in P is paired in S with each distinct value in $\pi_{sid}(S)$. Assume without loss of generality that the tuples in S are annotated with variables $y_1, \ldots, y_n$ and those of P are annotated with variables $\neg x_1^1$ through $\neg x_{n+k_m}^m$. Following the annotation pattern construction given in Figure 3.5, the query annotation

becomes:

$$\Phi = \bigvee_{i=1}^{m} (\neg x_1^i \vee \ldots \vee \neg x_{n+k_i}^i) \neg (y_1 x_1^i \vee \ldots \vee y_n x_n^i)$$

By negating Φ and removing redundant terms, we obtain:

$$\neg\Phi = \left[\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n+k_i} x_j^i\right] \vee \left[\bigwedge_{i=1}^{m} y_1 x_1^i \vee \ldots \vee y_n x_n^i\right]$$

By applying the inclusion-exclusion principle, the probability of $\neg\Phi$ is then:

$$\neg\Phi = P\left[\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n+k_i} x_j^i\right] + P\left[\bigwedge_{i=1}^{m} y_1 x_1^i \vee \ldots \vee y_n x_n^i\right] - P\left[\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n+k_i} x_j^i\right] \cdot P\left[y_1 \vee \cdots \vee y_n\right]$$

The first and the third term can be computed trivially regardless of the probability distribution. The second term can be computed efficiently in case of uniform distribution:

Proposition 21. *The number of models of the propositional formula*

$$\bigwedge_{i=1}^{m} y_1 x_1^i \vee \ldots \vee y_n x_n^i \qquad \textit{is} \qquad \sum_{j=1}^{n} \binom{n}{j} (2^n - 2^{n-j})^m.$$

The proof is immediate by analysing the combinatorial structure of the formula. The formula of Proposition 21 admits efficient model counting — and thus probability computation in the case of a uniform probability distribution for the variables — due to its symmetry: For any choice of k out of n variables y_j set to true, the number of satisfying assignments is the same and only depends on k, n, and m. In case of arbitrary input probability distributions, however, the formula is no longer symmetric and setting different k variables y_j to true can lead to different probabilities. In fact, arbitrary positive bipartite 2CNF formulas can be obtained by appropriately setting variables x_j^i to true or false.

3.6 Discussion

We close this chapter with an overview of related work and a discussion of the extendability of the dichotomy result to non-repeating queries with union.

3.6.1 Queries with union

It is tempting to extend the dichotomy of Theorem 5 to queries with union in the following straightforward way [34]: Let us say that a query Q (with union) is non-hierarchical if it has a union-free sub-query — obtained by non-deterministically choosing the left or right

IsHierarchical$^{\cup}$(Query Q)
  foreach *pair of attributes A,B occurring in Q* **do**
    if $\{A, AB, B\} \in \textsc{Crit}_{A,B}(Q)$ **then**
      return "No"
  return "Yes"

$$\textsc{Crit}_{A,B}(R) = \begin{cases} \emptyset & \text{if } [A] \notin \mathcal{E}(R) \text{ and } [B] \notin \mathcal{E}(R) \\ \{A\} & \text{if } [A] \in \mathcal{E}(R) \text{ and } [B] \notin \mathcal{E}(R) \\ \{B\} & \text{if } [B] \in \mathcal{E}(R) \text{ and } [A] \notin \mathcal{E}(R) \\ \{AB\} & \text{if } [A] \in \mathcal{E}(R) \text{ and } [B] \in \mathcal{E}(R) \end{cases}$$

$$\textsc{Crit}_{A,B}(\pi(Q)) = \textsc{Crit}_{A,B}(Q)$$

$$\textsc{Crit}_{A,B}(Q_1 \ltimes Q_2) = \{c_1 \cup c_2 \mid c_1 \in \textsc{Crit}_{A,B}(Q_1) \text{ and } c_2 \in \textsc{Crit}_{A,B}(Q_2)\}$$

$$\textsc{Crit}_{A,B}(Q_1 \cup Q_2) = \textsc{Crit}_{A,B}(Q_1) \cup \textsc{Crit}_{A,B}(Q_2)$$

Algorithm 4: The algorithm IsHierarchical$^{\cup}$ decides whether all union-free sub-queries of Q are hierarchical.

sub-query of each union operator in Q — and this sub-query is non-hierarchical in the sense of Definition 6, and conjecture that the dichotomy of Theorem 5 holds for the thus defined hierarchical and non-hierarchical classes of queries. This section demonstrates that this conjecture is only correct for one direction of the dichotomy: All non-hierarchical queries with union are #P-hard; conversely, there exists queries that are #P-hard despite all of their union-free sub-queries being hierarchical.

3.6.1.1 The hierarchical property for queries with union

We commence by defining the syntax and semantics of the union operator in relational algebra queries. Syntactically, a query may incorporate a union operator, viz:

- Union: $Q_1 \cup_\rho Q_2$, where $\mathcal{E}(Q_1) = \{A_1, \ldots, A_n\}$, $\mathcal{E}(Q_2) = \{B_1, \ldots, B_n\}$, and ρ is the following conjunction of correspondence conditions: $\rho = (A_1 \leftrightarrow B_1) \wedge \cdots \wedge (A_n \leftrightarrow B_n)$; note that each attribute of Q_1 and Q_2 participates in exactly one $\leftrightarrow$ condition.

The union operator is subject to the same well-formedness constraints as the difference operator, i.e., in $Q_1 \cup_\rho Q_2$ the schemata of Q_1 and Q_2 are compatible with the correspondence conditions in ρ. In the following, we denote the query language comprising the union operator by RA$^{\cup}$.

Next, we define formally the set $\mathcal{Q}_{\not\cup}(Q)$ of *union-free sub-queries of* Q. $\mathcal{Q}_{\not\cup}$ consists of the union-free queries that can be obtained from Q by choosing non-deterministically the left or the right sub-query of every union operator in Q:

Definition 14 (Union-free sub-queries, $\mathcal{Q}_{\not\cup}$)**.** *Let Q be a RA$^{\cup}$ query. The set $\mathcal{Q}_{\not\cup}(Q)$ of* union-free sub-queries *of Q is recursively defined by:*

$$\mathcal{Q}_{\not\cup}(Q) = \begin{cases} \{Q\} & \text{if } Q \text{ is union-free} \\ \mathcal{Q}_{\not\cup}(Q_1) \cup \mathcal{Q}_{\not\cup}(\delta_\rho(Q_2)) & \text{if } Q = Q_1 \cup_\rho Q_2 \\ \{q_1 \ltimes_\rho q_2 \mid q_1 \in \mathcal{Q}_{\not\cup}(Q_1) \text{ and } q_2 \in \mathcal{Q}_{\not\cup}(Q_2)\} & \text{if } Q = Q_1 \ltimes_\rho Q_2 \\ \{q_1 -_\rho q_2 \mid q_1 \in \mathcal{Q}_{\not\cup}(Q_1) \text{ and } q_2 \in \mathcal{Q}_{\not\cup}(Q_2)\} & \text{if } Q = Q_1 -_\rho Q_2 \\ \{\pi(q) \mid q \in \mathcal{Q}_{\not\cup}(Q_1)\} & \text{if } Q = \pi(Q_1) \end{cases}$$

The renaming operator δ_ρ in the rule for $Q = Q_1 \cup_\rho Q_2$ renames the attributes exported by Q_2 to the names of the corresponding attributes in Q_1 in order to yield a valid query. For example a query $R(A,B) \cup_{(A \leftrightarrow X) \wedge (B \leftrightarrow Y)} S(X,Y)$ yields $\delta_\rho S(X,Y) = \delta_{X \to A, Y \to B} S(X,Y)$.

Example 22. Let $Q = ((R \cup S) \bowtie U) \cup (V \bowtie W)$. Then $\mathcal{Q}_\emptyset(Q)$ contains the three queries $R \bowtie U$, $S \bowtie U$, and $V \bowtie W$. □

The decision problem

Problem IsHierarchical$^\cup$: Input: RA$^\cup$ query Q
Output: "Yes" if all union-free sub-queries of Q are hierarchical, "No" else

is decided by the algorithm IsHierarchical$^\cup$, cf. Algorithm 4. The algorithm iterates over all pairs A, B of attributes and stops when it finds attributes for which Q has a union-free sub-query that contains three relation symbols $R^{[A][\neg B]}$, $S^{[A][B]}$, $T^{[\neg A][B]}$. The set constructed by $\text{Crit}_{A,B}$ is a subset of $2^{\{A,B,AB\}}$; for each sub-query Q' of Q it contains the set $c \subseteq \{A, B, AB\}$ if and only if Q' has a union-free sub-query that contains the relations R, S, T corresponding to the values in c. For example, $\{A, AB\} \in \text{Crit}_{A,B}(Q)$ if and only Q has a union-free sub-query that contains a relation $R^{[A][\neg B]}$ and a relation $S^{[A][B]}$.

Proposition 22. *The decision problem IsHierarchical$^\cup$ is in PTIME.*

Proof. The for-loop in IsHierarchical$^\cup$ requires $\mathcal{O}(|Q|^2)$ invocations of Crit. The latter can be implemented as a DFS-traversal of the tree representation of Q; at each leaf node (i.e., relation symbol) of Q, the evaluation of the transitive closure $[A] \in \mathcal{E}(R)$ and $[B] \in \mathcal{E}(R)$ can be implemented in PTIME (for instance via graph connectivity); at each inner node, the set union is clearly in PTIME since the size of the set $\text{Crit}_{A,B}$ for each sub-query of Q is at most 8. □

3.6.1.2 Queries with non-hierarchical union-free sub-queries

We show that a RA$^\cup$ query that has a non-hierarchical union-free sub-query is #P-hard; let Q be such a query, and Q' a non-hierarchical union-free sub-query of Q. Let $\mathcal{D}$ be a database in which every relation that occurs in Q but not in Q' is empty; then clearly $Q(\mathcal{D}) = Q'(\mathcal{D})$. Hence in order to prove $\#P$-hardness of Q, it suffices to apply the hardness reduction from Section 3.4.2 to Q' and populate the remaining relations to be empty. This construction establishes:

Corollary 23 (Lemma 17)**.** *Let Q be a RA$^\cup$ query that has a non-hierarchical union-free sub-query. Then the data complexity of the probabilistic query evaluation problem on tuple-independent databases has #P-hard data complexity.*

3.6.1.3 Queries with hierarchical union-free sub-queries

Let us consider the query

$$Q_{RA} = \pi_{\emptyset}\Big[S(A_s, B_s) - \pi_{A_m,B_m}\Big(M(A_m, B_m) \bowtie_{A_m=A_n} N(A_n) \cup U(A_u, B_u) \bowtie_{B_u=B_v} V(B_v)\Big)\Big]$$

and its $\mathrm{RC}^{\exists,\neg}$-translation

$$\begin{aligned} Q_{RC} = {} & \exists_A \exists_B \big(S(A,B) \wedge \neg M(A,B) \wedge \neg U(A,B)\big) \quad \vee \\ & \exists_A \exists_B \big(S(A,B) \wedge \neg M(A,B) \wedge \neg V(B)\big) \quad \vee \\ & \exists_A \exists_B \big(S(A,B) \wedge \neg N(A) \wedge \neg U(A,B)\big) \quad \vee \\ & \exists_A \exists_B \big(S(A,B) \wedge \neg N(A) \wedge \neg V(B)\big) \end{aligned}$$

We will show that Q_{RA} is $\#P$-hard although all of its union-free sub-queries are hierarchical, viz:

Proposition 24. *There exists a #P-hard query whose union-free sub-queries are hierarchical.*

Proof. The proof is by reduction from model-counting problem #2DNF. Let $\Psi = \bigvee_{(i,j)\in E} x_i y_j$ be a 2DNF formula with $|E|$ clauses over disjoint variable sets $\mathbf{X}$ and $\mathbf{Y}$.

First, define the query Q_{hard} to be the fourth conjunct of query Q_{RC}:

$$Q_{\text{hard}} = \exists_A \exists_B \big(S(A,B) \wedge \neg N(A) \wedge \neg V(B)\big).$$

Let $\mathcal{D}$ be a database in which M and U are the cross-product of the active domains of variables A and B, and each tuple is annotated with $\top$. The first three disjunction-free sub-queries of Q_{RC} cannot be satisfied by $\mathcal{D}$ and hence $Q_{RC}(\mathcal{D}) = Q_{\text{hard}}(\mathcal{D})$. Also note that Q_{hard} is equivalent to the prototypical non-hierarchical, #P-hard conjunctive query $\exists_A \exists_B N(A) \wedge S(A,B) \wedge V(B)$ if the annotations of tuples in N and V are negated.

It remains to be shown how to populate in the above database $\mathcal{D}$ the relations N, S, V such that the annotation of $Q_{\text{hard}}(\mathcal{D})$ and hence of Q_{RC} is exactly Ψ. For every variable $x_i \in \mathbf{X}$, let N in $\mathcal{D}$ contain a tuple x_i annotated with x_i; symmetrically, for every variable $y_j \in \mathbf{Y}$, let V in $\mathcal{D}$ contain a tuple y_j annotated with y_j. For every clause $x_i y_j$ in Ψ, let S in $\mathcal{D}$ contain the tuple (x_i, y_j), annotated with $\top$. Then the annotation of the Boolean query result of Q_{hard} and of Q_{RC} on $\mathcal{D}$ is exactly Ψ and hence $P_{Q_{RC}(\mathcal{D})} = P_\Psi = \#\Psi \cdot 2^{-|\text{vars}(\Psi)|}$. $\square$

Intuitively, even if all union-free sub-queries are hierarchical, a union-operator with odd polarity may yield non-hierarchical disjunction-free sub-queries in the $\mathrm{RC}^{\exists,\neg}$-translation that have to be ruled out by the correct categorisation of tractable non-repeating relational

algebra queries. In this light, a dichotomy for non-repeating relational algebra queries with union seems feasible, but is left open for future work.

3.6.2 Related work

It is known that negation is a substantial source of complexity already for uncertain databases without probabilities [3]. In the context of probabilistic databases, the MystiQ system supports a limited class of queries with nested NOT EXISTS operators [89]. A framework for the evaluation of full relational algebra queries including negation is presented in Chapter 4. The results in this chapter are to the best of our knowledge the first complexity results for queries with negation.

Our dichotomy is in line with and contributes to a succession of complexity results for queries on probabilistic databases: Starting from a first example of a #P-hard query [40], PTIME/#P-hard tractability dichotomies have been established for non-repeating conjunctive queries [21] and unions of conjunctive queries [25]; a complexity trichotomy has been proven for positive queries with HAVING aggregates [72]. Complexity results for queries with inequality constraints have been investigated in [60, 61]. The majority of the hardness reductions in the above works are from the #P-hard model-counting problem for positive 2DNF formulas [84] or bipartite positive 2DNF formulas [67]. The complexity class #P was first defined by in [84].

OBDDs have originally been proposed in [12, 13], and a recent overview is given in [90]. The class of *inversion-free* unions of conjunctive queries (UCQ) is equivalent to the class of UCQ queries that admit polynomial-size OBDDs on any tuple-independent database [48]; the notion of inversion-freeness is similar to $\exists$-consistency and allows for Boolean combinations of OBDDs of sub-queries with only polynomial overhead. OBDD constructions have been leveraged to show tractability of a large class of queries with inequalities on tuple-independent databases, and of non-hierarchical conjunctive queries on restricted database instances [60]. UCQs with inequalities have been characterised in terms of their corresponding OBDDS [47].

A recent overview of various topics in probabilistic databases has been compiled in [79].

The closest in spirit to the results and proof techniques of this chapter is the PTIME/#P dichotomy theorem for unions of conjunctive queries (UCQ) [25]. The algorithm for tractable UCQ queries is an alternation of two steps: First, a given UCQ query expression is translated into an equivalent conjunctive-normal-form expression, such that the probability of the latter can be expressed using the inclusion/exclusion principle in terms of the probabilities of disjunctive expressions. Each of the resulting expressions has a root variable and satisfies properties similar to what we call *canonicalised*, guaranteeing that multiple occurrences of the same relation are joined by the root variable via the same attribute. These properties are captured by the notion of a *separator* variable. Similar to the case of root variables in our algorithm, the existence of the separator variable provides

that the annotations of the query expression are independent for different valuations of the separator variable. The PTIME algorithms for UCQ and for $\mathrm{RA}^{\not\cup}$ queries are thus based on the same high-level idea: The input query is rewritten into a relational calculus query that satisfies certain structural properties which in turn allow to compute its probability in PTIME. The main challenge in the case of UCQ queries are repeating relation symbols, and the main challenge for $\mathrm{RA}^{\not\cup}$ queries are difference operators which effectively turn a non-repeating $\mathrm{RA}^{\not\cup}$ query into a $\mathrm{RC}^{\exists,\neg}$ query *with* repeating relation symbols; in this light, despite $\mathrm{RA}^{\not\cup}$ queries being non-repeating, our evaluation technique solves obstacles similar to those encountered in the case of repeating UCQ.

The hardness proof of the UCQ dichotomy is a reduction from the 2DNF model counting problem, but bears no further resemblance to our result with respect to its proof technique. Our proof in terms of separate reductions for each of the 24 patterns can be thought of as a non-trivial extension of the hardness proof for non-repeating conjunctive queries [21], which latter proof constitutes cases $P_{x.1}$ in our proof. While in a purely conjunctive query the projection operators can always be applied last and all join operations commute, the main new challenge in the presence of difference operators is that the latter severely restrict equivalence-preserving transformations of the query. Consequently, a specific hardness reduction is required for each structurally different class of queries.

3.6.3 Conclusion and extensions

This chapter discusses a fundamental aspect of query processing on probabilistic data, viz. the classification of queries into those which are tractable with respect to data complexity, and those which are considered intractable because they are provably #P-hard. The following Chapter 4 presents experimental evidence that such a classification is indeed vital, since the performance of exact query evaluation degenerates exponentially with the size of the database and is infeasible for any but the smallest instances. The existence of the dichotomy in terms of the efficiently decidable *hierarchical* property allows a database management system to switch between exact query evaluation (for tractable queries) and approximate query evaluation (for intractable queries).

Possible further research directions related to the results of this chapter comprise the extension of the dichotomy to more expressive queries. We gave evidence that a naïve classification of queries with union in terms of *non-hierarchical union-free sub-queries* captures some, but not all non-repeating #P-hard queries; a finer analysis would thus be required to established the exact dichotomy. A dichotomy for full relational algebra, i.e., in particular for queries with repeating relation symbols, seems unattainable since basic reasoning tasks on such queries — such as equivalence, emptiness, or subsumption — are undecidable.

Both the result of this chapter and of related works analysing the complexity of queries on probabilistic data inherently focus on data complexity and do not touch on query or combined complexity [72, 25]. As argued in Section 2.4, the combined complexity of $\mathrm{RA}^{\not\cup}$ is

in PSPACE; on the other hand, Theorem 5 shows that hierarchical $\mathrm{RA}^{\not\vee}$ queries have PTIME data complexity, but our proposed algorithm has non-elementary complexity with respect to the query size (cf. Section 3.2.3). The trade-off between data and query complexity is left open by the results of this thesis: Can hierarchical $\mathrm{RA}^{\not\vee}$ queries be evaluated in polynomial-time data complexity and (at the same time) elementary query complexity?

Chapter 4

Exact and approximate evaluation of relational algebra queries

Approximate query evaluation is preferred over exact query evaluation in cases where fast approximate answers are favoured over delayed exact answers; in probabilistic databases, this is usually the case for intractable queries for which exact evaluation is infeasible for any but the smallest database instances. This chapter describes the approximate probability computation algorithm used by the SPROUT query engine [31, 44, 63, 32]. The algorithm works for *arbitrary propositional formulas* and thus for any relational algebra query and pc-table, it is *anytime*, and *admits absolute and relative error guarantees*. Given a formula Φ whose exact probability is p, it computes a probability $\hat{p}$ such that

(i) $p - \epsilon \leq \hat{p} \leq p + \epsilon$ for a given absolute error ϵ, or

(ii) $p(1 - \epsilon) \leq \hat{p} \leq p(1 + \epsilon)$ for a given relative error ϵ.

More precisely, the algorithm can compute a contiguous interval of approximations $\hat{p}$ within the given error bounds. It starts with trivial lower and upper bounds $[0, 1]$ and incrementally tightens the interval until the approximation is reached. It is thus an anytime algorithm, since it returns an approximation even if it is stopped before it ends and finds increasingly better bounds the longer it runs.

The algorithm is an interplay of two techniques: (1) efficient computation of lower and upper bounds on the probability of a formula, and (2) decomposition of a formula into an equivalent disjunction or conjunction of independent or mutually exclusive simpler formulas. Given bounds on the probability of the simpler formulas, we can then efficiently compute bounds on the probability of the original formula.

The algorithm works as follows. Given a formula, we first compute probability bounds via the first technique. If the bounds are tight enough to meet the error guarantee, then we stop; otherwise, we decompose the formula into simpler formulas using the second technique, compute probability bounds for the simpler formulas using the first technique and use them

to derive bounds for the original formula. This decomposition step is repeated until the error guarantee is met. By exhaustively decomposing the formula down to literals as approached by knowledge compilation techniques [26], we can compute its exact probability.

The first technique can compute probability bounds without error guarantees in polynomial time. The second technique can compute bounds with error guarantees, and although each decomposition step takes polynomial time, it may require exponentially many steps to reach the desired approximation. The quality of bounds computed by the first technique is crucial since good bounds can drastically decrease the number of decomposition steps performed by the second technique. Our approach is to derive such bounds via model-based approximation as explained next.

Recall that given a formula Φ, $\mathcal{M}(\Phi)$ denotes the set of its models (or satisfying assignments). A pair of formulas Φ_L and Φ_U from a language $\mathcal{L}'$ of propositional formulas represents a model-based approximation of Φ if $\mathcal{M}(\Phi_L) \subseteq \mathcal{M}(\Phi) \subseteq \mathcal{M}(\Phi_U)$, that is, the set of models of Φ includes the set of models of Φ_L and is included in the set of models of Φ_U. We call Φ_L and Φ_U lower and upper bounds for Φ, respectively. The lower bound Φ_L is optimal if there is no other formula Φ'_L in $\mathcal{L}'$ such that $\mathcal{M}(\Phi_L) \subset \mathcal{M}(\Phi'_L) \subseteq \mathcal{M}(\Phi)$; the case of optimal upper bounds is symmetric. Our interest in model bounds stems from the observation that model bounds imply probability bounds: $P(\Phi_L) \leq P(\Phi) \leq P(\Phi_U)$. The bounds Φ_L and Φ_U are chosen such that they are optimal in a more restrictive language $\mathcal{L}'$ of formulas for which probability computation can be done in polynomial time. We consider the language of so-called read-once formulas, called 1OF for short, and its subset of positive formulas in disjunctive normal form, called iDNF for short. These languages are defined in Chapter 2.

There are two main observations behind the design of our algorithm. Firstly, sufficient approximations can often be obtained within a few decomposition steps and there is thus no need to exhaustively decompose the input formula down to literals. This motivates the incremental nature of the algorithm as well as the use of effective termination checks. According to our experiments, a relatively small number of well-chosen decomposition steps computable within a few seconds usually suffice to guarantee good precision. The formulas obtained after these decomposition steps may still be large, yet they account for a very small fraction of the overall probability mass. The second observation concerns tractable query evaluation in probabilistic databases. We can effectively derive orders of decomposition steps under which any approximation and in particular the exact probability can be obtained in polynomially many steps when decomposing formulas annotating results of tractable conjunctive query without self-joins. Most notably, this is achieved without a-priori knowledge of the query or the input database.

In addition to the anytime algorithm, we prove in this chapter a syntactic characterisation of optimal iDNF lower and upper bounds for positive formulas in disjunctive normal form. We give algorithms that enumerate such bounds with polynomial delay and thus

allow to inspect several lower and upper bounds and choose among them a pair whose probabilities are closest to the exact probability of the input formula. For formulas with clauses of arity at most k, such as those annotating results of positive relational queries on tuple-independent databases [79], we give a polynomial-time algorithm that can find an optimal iDNF lower bound whose probability is within a factor k from the largest probability of optimal iDNF lower bounds; finding the latter is NP-hard. For nested positive formulas, we show how to efficiently compute 1OF lower and upper bounds.

Organisation of this chapter

This chapter is organised as follows. Sections 4.1 and 4.2 discuss model-based approximations and incremental decomposition of formulas. We report on experiments in Section 4.3. Related work and possible further research directions are discussed in Section 4.4. Proofs of formal statements and details on the experiments can be found in Section 4.5 and Appendices A and B, respectively.

4.1 Model-based approximations

The problem investigated in this section is: Given a formula Φ in a propositional language $\mathcal{L}$, find formulas in a (more restrictive) language $\mathcal{L}'$ that approximate Φ. We restrict our investigation to positive (or without loss of generality, unate) and irreducible formulas; the reason for this restriction is efficiency, since for arbitrary formulas it is NP-hard to decide whether true ($\top$) or false ($\bot$) are bounds. This defeats our goal of efficient model-based approximation.

Before looking into approximating positive DNF formulas within the language of iDNF (Sections 4.1.1 and 4.1.2) and of positive nested formulas within the language of 1OF (Section 4.1.3), we define the notion of optimal bounds.

Definition 15 (Optimal lower and upper bounds)**.** *Let $\mathcal{L}'$ and $\mathcal{L}$ be languages of propositional formulas such that $\mathcal{L}' \subseteq \mathcal{L}$. Given a formula $\Phi \in \mathcal{L}$, the formulas $\Phi_L, \Phi_U \in \mathcal{L}'$ are lower and upper bounds for Φ with respect to $\mathcal{L}'$ if $\mathcal{M}(\Phi_L) \subseteq \mathcal{M}(\Phi)$ and $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Phi_U)$, respectively.*

If in addition there are no formulas $\Phi'_L, \Phi'_U \in \mathcal{L}'$ such that $\mathcal{M}(\Phi_L) \subset \mathcal{M}(\Phi'_L) \subseteq \mathcal{M}(\Phi)$ and $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Phi'_U) \subset \mathcal{M}(\Phi_U)$, then Φ_L is a greatest lower bound (GLB) and Φ_U is a least upper bound (LUB) for Φ with respect to $\mathcal{L}'$.

The notions GLB and LUB provide an intuitive semantic characterisation of *optimal* bounds. We consider languages $\mathcal{L}'$ such that (i) we can efficiently find optimal bounds in $\mathcal{L}'$, and (ii) $\mathcal{L}'$ allows for efficient probability computation. The 1OF and iDNF languages satisfy both desiderata.

Our interest in model-based approximation stems from the observation that, for formulas over independent random variables, model-based bounds imply probability bounds:

Proposition 25. *For any formulas Φ and Ψ with $\Phi \models \Psi$ and thus $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Psi)$ it holds that $P(\Phi) \leq P(\Psi)$.*

The statement follows immediately by applying the definition of the probability of a formula, cf. Equation (2.3). Before we discuss iDNF and 1OF approximations, we give a simple syntactic characterisation of (not necessarily optimal) bounds for positive DNF formulas:

Proposition 26. *Let Φ and Ψ be positive DNF formulas. The following statements are equivalent:*

1. *$\Phi \models \Psi$;*
2. *Ψ can be obtained from Φ by adding clauses and removing literals from Φ's clauses;*
3. *Φ can be obtained from Ψ by removing clauses and adding literals to Ψ's clauses.*

The proof is given in Section 4.5. According to Proposition 26, *all* bounds for a given positive DNF formula Φ can be defined by simple *syntactic* manipulations of Φ: Lower bounds by removing clauses or adding literals to existing clauses, and upper bounds by adding clauses or removing literals from existing clauses. Removing all clauses results in the lower bound $\bot$. Removing all literals from a clause results in the upper bound $\top$. Indeed, by removing clauses from a formula we reduce its set of models. By adding variables to clauses, we further constrain their satisfiability and hence reduce the set of models. In both cases, one obtains formulas that are lower bounds. The inverse manipulations lead to upper bounds. More importantly, Proposition 26 states that all bounds can be gained in this way. The following sections show that it is not necessary to add literals to clauses in order to find *optimal* iDNF lower bounds, while removing variables may be required in order to get to optimal iDNF upper bounds.

Example 23. Recall the formulas

$$\begin{aligned}\Phi_1 &= x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \vee x_3y_6z_4\\ \Phi_2 &= x_1y_1z_1 \vee x_1y_2z_2 \vee x_1y_3z_1 \vee x_1y_4z_2 \vee x_1y_5z_3 \vee x_1y_6z_4 \vee\\ &\quad x_2y_3z_1 \vee x_2y_4z_2 \vee x_2y_5z_3 \vee x_2y_6z_4 \vee x_3y_5z_3 \vee x_3y_6z_4\end{aligned}$$

from Figure 2.3; neither Φ_1 nor Φ_2 are in iDNF. The following iDNF formulas are lower and upper bounds for Φ_1 (several others are possible):

$$\Phi_L = x_1y_1z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \qquad \Phi_U = z_1 \vee x_1y_2 \vee x_2y_4z_2 \vee x_3.$$

Indeed, Φ_L is a lower bound since it is a subset of Φ. Each clause in Φ implies at least one clause of Φ_U, hence Φ_U is an upper bound. A closer inspection reveals that each clause in

Φ_U can be obtained by dropping literals from clauses in Φ. We will show in the remainder of this section that the bounds Φ_L and Φ_U are indeed *optimal* for iDNF. □

The following sections characterise syntactically the optimal iDNF bounds for positive formulas, and give algorithms to find such bounds.

4.1.1 iDNF greatest lower bounds

Our main tool used to find optimal iDNF lower bounds is the syntactic notion of maximal lower bounds.

Definition 16 (Maximal lower bound for iDNF)**.** *Let Φ be a positive DNF formula. An iDNF formula Φ_L is a* maximal lower bound (MLB) *for Φ if it satisfies the conditions:*

1. *(*Lower bound*) Φ_L is a subset of Φ: $\Phi_L \subseteq \Phi$;*
2. *(*Maximality*) Φ_L cannot be further extended: There is no clause $\varphi \in \Phi$ with $vars(\varphi) \cap vars(\Phi_L) = \emptyset$.*

The first condition ensures that Φ_L is indeed an iDNF lower bound. The second condition enforces that it is a maximal lower bound. An MLB for a formula Φ is thus a maximal set of pairwise independent clauses of Φ.

Example 24. The MLBs for formula Φ_1 from Figure 2.3 are:

$$1 : x_1y_1z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \qquad 2 : x_1y_1z_1 \vee x_2y_4z_2 \vee x_3y_6z_4$$
$$3 : x_1y_2z_2 \vee x_2y_3z_1 \vee x_3y_5z_3 \qquad 4 : x_1y_2z_2 \vee x_2y_3z_1 \vee x_3y_6z_4$$ □

With respect to the iDNF language, maximal lower bounds correspond precisely to greatest lower bounds. This is particularly important, since MLB is a syntactic notion defined in terms of sets of clauses, and GLB is a semantic notion defined in terms of sets of models.

Theorem 27. *Let Φ be a positive DNF formula. An iDNF formula Φ_L is a maximal lower bound for Φ if and only if Φ_L is a greatest lower bound for Φ.*

The formal proof is given in Section 4.5. Intuitively, the implication GLB $\Rightarrow$ MLB holds since if any of the two MLB conditions fail, then we cannot obtain a GLB. For the other direction, it can be shown that no strict upper bound Φ'_L for Φ_L is an iDNF lower bound for Φ. The latter case has two subcases: There is a clause $\varphi \in \Phi$ that is in Φ'_L and either is or is not in Φ_L. In the first subcase, φ necessarily shares variables with one clause in $\varphi_L \in \Phi_L$, and, according to Lemma 1, it must be contained in φ_L. But then, φ must be φ_L,

GreedyIDnfGlb(Positive DNF Φ with max-arity k)

```
▷ outputs An iDNF GLB of Φ
Sort clauses of Φ in descending order φ_1, ..., φ_n of their probabilities
P(φ_1) ≥ P(φ_2) ≥ ··· ≥ P(φ_n).
Φ_L ← ∅
for i = 1...n do
    if vars(φ_i) ∩ vars(Φ_L) = ∅ then
        Φ_L ← Φ_L ∪ {φ_i}
output Φ_L
```

Algorithm 5: Finding an iDNF greatest lower bound with probability at most factor k smaller than the largest probability of iDNF greatest lower bounds.

otherwise Φ_L can be further extended and hence is no MLB. The second subcase follows similarly.

The syntactic definition of MLBs suggests an algorithm for enumerating all GLBs by recursively constructing all maximal subsets of pairwise independent clauses of Φ. This algorithm needs time exponential in the number of clauses. The following proposition shows that this is necessarily so:

Proposition 28. *The positive DNF formula*

$$\Phi = (x_1y_1 \vee x_1y_2) \vee \cdots \vee (x_ny_{2n-1} \vee x_ny_{2n})$$

has $2n$ clauses and 2^n iDNF GLBs. Any disjunction of either x_iy_{2i-1} or x_iy_{2i} for all $1 \leq i \leq n$ is an iDNF GLB of Φ.

The proof is immediate from the statement. Since there can be exponentially many GLBs that are by definition incomparable with respect to their model sets, it may be desirable to find "the best" GLB for a formula according to different criteria. Possible rankings of GLBs are by the number of clauses or, useful in our context, by their probabilities, provided they are defined over random variables. Additionally, one may strive to enumerate all GLBs with polynomial delay[1], which means that the time before finding the first GLB, as well as the time between every two consecutive GLBs, is polynomial only in the input size, and not in the number of GLBs. In the context of GLBs, a polynomial delay enumeration allows to inspect a sequence of GLBs and pick the one with highest probability. We obtain results for ranking and enumerating GLBs by exploiting the following correspondence between GLBs and independent sets in the clause-dependency graphs of formulas.

Definition 17 (Clause-dependency graph)**.** *Let Φ be a DNF formula. The* clause-dependency graph *of Φ is a graph $G = (\Phi, E)$, where nodes are clauses of Φ, and two nodes φ and ψ are connected if $vars(\varphi) \cap vars(\psi) \neq \emptyset$.*

[1] An overview of different complexity measures — amongst them polynomial delay — for enumeration problems was given in [49].

Each node $\varphi \in \Phi$ in G has a weight defined by

$$w(\varphi) = -\ln\bigl(1 - P(\varphi)\bigr) = -\ln\bigl(1 - \prod_{x \in \varphi} P_x\bigr). \tag{4.1}$$

The weight $W(\Psi)$ of a node set Ψ is $W(\Psi) = \sum_{\psi \in \Psi} w(\psi)$.

Proposition 29. *Let Φ be a positive DNF formula. A formula $\Phi_L \subseteq \Phi$ is an iDNF greatest lower bound for Φ if and only if the clauses of Φ_L represent a maximal independent set in the clause-dependency graph of Φ.*

The proof is given in Section 4.5. As the following proposition shows, the correspondence of Proposition 29 can be extended to the case of greatest lower bounds with the largest probability. Such bounds correspond to maximal independent sets with the largest weight.

Proposition 30. *Let Φ be a positive DNF formula and $G = (\Phi, E)$ be its clause-dependency graph. A formula Φ_L is an iDNF greatest lower bound of Φ with the largest probability among all iDNF greatest lower bounds of Φ if and only if Φ_L is a maximum-weight independent set in G.*

The proof is given in Section 4.5. By the correspondence between iDNF greatest lower bounds and maximal independent sets, known results for the latter, such as enumeration of maximal independent sets with polynomial delay, immediately carry over to the former.

Corollary 31 (Proposition 29, [82, 49])**.** *The set of all iDNF greatest lower bounds for a positive DNF formula with n clauses can be enumerated with $O(n^3)$ delay.*

Since finding a maximum independent set is NP-hard, we cannot efficiently enumerate the maximal independent sets, or equivalently the iDNF GLBs, in decreasing order of their sizes, and, more importantly here, of their probabilities.

Corollary 32 (Proposition 29, [35])**.** *Enumerating iDNF greatest lower bounds for a given positive DNF formula Φ in decreasing order of their probabilities is NP-hard.*

Although we cannot efficiently obtain an iDNF GLB with largest probability, a constant-factor approximation can be obtained for positive DNF formulas with max-arity k. For such a formula Φ, Algorithm 5 constructs an iDNF greatest lower bound Φ_L by iterating over Φ's clauses in decreasing order of their probabilities and greedily selecting a maximal set of independent clauses: Φ_L thus contains the first clause φ_1 in the order, then the next clause independent of φ_1, and so on. The probability of this lower bound is at most a factor k smaller than the largest among the probabilities of all iDNF greatest lower bounds for Φ:

Theorem 33. *Given a positive DNF formula Φ with max-arity k and n clauses, Algorithm 5 constructs an iDNF greatest lower bound Φ_L for Φ in time $O(n^2)$ such that $P(\Phi'_L) \leq k \cdot P(\Phi_L)$ for every iDNF greatest lower bound Φ'_L for Φ.*

The proof is given in Section 4.5.

4.1.2 iDNF least upper bounds

As for iDNF greatest lower bounds, we next give a syntactic characterisation of iDNF least upper bounds for positive DNF formulas. We first introduce a few necessary notions.

Definition 18 (Witness and critical witness)**.** *Let Φ and Ψ be formulas. A clause $\varphi \in \Phi$ is a* witness *for a clause $\psi \in \Psi$ if $\varphi \models \psi$. We also say that ψ* has the witness *φ. If in addition there is no clause $\psi' \in \Psi$ with $\psi \neq \psi'$ and $\varphi \models \psi'$, then φ* is a critical witness *for ψ, and ψ* has the critical witness *φ.*

Example 25. Consider the formulas

$$\Phi_1 = x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \vee x_3y_6z_4 \qquad \text{(cf. Figure 2.3)}$$
$$\Psi = x_1y_1 \vee x_2 \vee z_2 \vee x_1y_6$$

Clause $x_1y_1 \in \Psi$ has the critical witness $x_1y_1z_1 \in \Phi_1$, clause $x_2y_4z_2$ is a non-critical witness for x_2 and $z_2 \in \Psi$, and clause $x_1y_6 \in \Psi$ has no witness in Φ_1. □

We are now ready to present our main syntactic tool for finding iDNF optimal upper bounds.

Definition 19 (Minimal upper bound for iDNF)**.** *Let Φ be a positive DNF formula. An iDNF formula Φ_U is a* minimal upper bound (MUB) *for Φ if it satisfies the conditions:*

1. *(*Upper bound*) Every $\varphi \in \Phi$ is a witness for at least one clause in Φ_U;*
2. *(*Maximality*) There is no clause $\varphi_u \in \Phi_U$ that can be extended by a variable from $vars(\Phi)$ while preserving condition (1) and keeping Φ_U in iDNF;*
3. *(*Criticality*) Every clause $\varphi_u \in \Phi_U$ has at least one critical witness in Φ.*

Following Lemma 1, the first condition ensures that $\Phi \models \Phi_U$. The second condition ensures that we cannot obtain an iDNF refinement of Φ_U which is still an upper bound for Φ by making clauses in Φ_U more specific (thus narrowing the set of models of Φ_U). The third condition states that all clauses in Φ_U are necessary, and dropping any of them would lead to clauses in Φ violating Lemma 1.

This syntactic characterisation of upper bounds precisely matches the semantic notion of least upper bounds:

Theorem 34. *Let Φ be a positive DNF formula. An iDNF formula Φ_U is a minimal upper bound for Φ if and only if Φ_U is a least upper bound for Φ.*

The proof is given in Section 4.5. Intuitively, the implication LUB ⇒ MUB holds since, if any of the three MUB conditions fail, we cannot obtain an LUB. For the other direction, any difference between an MUB Φ_U for Φ and a potentially better upper bound Φ'_U requires

```
PolyMub(Positive DNF Φ, iDNF Φ_U)
    ▷ outputs iDNF MUBs of Φ with polynomial delay
    if Φ = {∅} then
        ▷ Φ is true and has the trivial MUB {∅}
        output Φ
    else if Φ = ∅ then
        output Φ_U
    else
        ▷ Remove redundant clauses from Φ
        ▷ Duplicates are automatically discarded since Φ_R is a set
        Φ_R ← {φ | φ ∈ Φ ∧ ∄ψ ∈ Φ : ψ ⊂ φ}
        x ← Pick variable from Φ_R
        foreach ψ ∈ Φ_R with x ∈ ψ do
            ▷ Add clause ψ to Φ_U and recursively find MUBs of the formula obtained from Φ_R by
              removing ψ and variables in vars(ψ) from all other clauses
            PolyMub({φ \ vars(ψ) | φ ∈ Φ_R ∧ φ ≠ ψ}, Φ_U ∪ {ψ})
```

Algorithm 6: Enumerating iDNF least upper bounds with polynomial delay. Initial call: $\textsc{PolyMub}(\Phi, \emptyset)$.

a clause $\varphi'_u \in \Phi'_U$ strictly containing (syntactically) a clause $\varphi \in \Phi_U$. A case analysis on a variable $x \in \varphi'_u$ and $x \notin \varphi_u$ shows that Φ_U does not satisfy the syntactic characterisation of an MUB: Either the criticality condition fails, or a clause in Φ_U may be expanded by x to obtain a smaller bound.

Example 26. The iDNF formula $\Phi^1_{1,u} = x_1y_1 \vee y_2 \vee x_2y_3z_1 \vee y_4z_2 \vee x_3y_5z_3 \vee y_6z_4$ is an MUB for Φ_1 from Figure 2.3. Every clause in $\Phi^1_{1,u}$ has exactly one witness in Φ_1 which is also critical. No clause can be extended further, since $\mathrm{vars}(\Phi^1_{1,u}) = \mathrm{vars}(\Phi_1)$. The iDNF formula $\Phi^2_{1,u} = x_1 \vee x_2 \vee x_3y_5z_3 \vee y_6z_4$ is also an MUB; each of the two clauses x_1 and x_2 has two critical witnesses. It can be checked that $\Phi^2_{1,u}$ cannot be extended by any variable from $\mathrm{vars}(\Phi_1)$. □

As in the case of lower bounds, the number of least upper bounds for a given formula can be exponential in its size:

Proposition 35. *The positive DNF formula*

$$\Phi = (x_1y_1 \vee x_1y_2) \vee \cdots \vee (x_ny_{2n-1} \vee x_ny_{2n})$$

has $2n$ *clauses and* 3^n *iDNF LUBs. Any disjunction of either* x_i*, or* $x_iy_{2i-1} \vee y_{2i}$*, or* $y_{2i-1} \vee x_iy_{2i}$ *for all* $1 \leq i \leq n$ *is an iDNF LUB of* Φ*.*

The proof is immediate from the statement. Algorithm 6 enumerates iDNF least upper bounds with polynomial delay using a simple strategy that relies on the compositional nature of minimal upper bounds: An MUB of an irreducible positive DNF formula $\Phi = \psi \vee \Phi'$,

where ψ is a clause and Φ' is a formula, is the disjunction of the MUB of ψ, which is ψ itself, and of the MUB of Φ', where we remove from Φ' the variables in ψ and subsumed clauses. Multiple MUBs can then be obtained by picking different clauses ψ that share a variable. This condition is necessary to guarantee distinct MUBs; indeed, these clauses are incomparable and each of them will be in an MUB. Note how empty clauses are handled: If the irreducible input formula is $\{\emptyset\}$, i.e., true, then PolyMub returns its only MUB $\{\emptyset\}$. In each recursive call, Φ cannot contain the empty clause, because removing from the irreducible formula Φ_R the variables from one of its clauses ψ cannot yield an empty clause.

Theorem 36. *Algorithm 6 enumerates iDNF least upper bounds of a positive DNF formula with n clauses with $O(n^3)$ delay.*

The formal proof is given in Section 4.5. Intuitively, Algorithm 6 is correct, i.e., every returned formula is an MUB and thus an LUB, since properties 1–3 in Definition 19 inductively carry over from the recursively generated MUB to the formula obtained by adding clause ψ. The returned MUBs are unique and thus incomparable, since they differ in the clause that contains variable x picked before the for-loop. The delay between consecutive MUBs is cubic in the number n of clauses in the input formula Φ, because the recursion depth of PolyMub is bounded by n, and each recursion step can be performed in time quadratic in n.

Algorithm 6 is not complete: For instance, it does not find MUBs $a \vee b$ and $x \vee y$ for the formula $ax \vee bx \vee ay \vee by$, because it cannot factorise it into $(a \vee b) \wedge (x \vee y)$. A complete enumeration algorithm for MUBs is discussed in the following sub-section.

4.1.2.1 Exhaustive enumeration of minimal upper bounds

While Algorithm 6 enumerates MUBs with polynomial delay, it is not complete for the set of all MUBs. As for greatest lower bounds, we can enumerate *all* iDNF least upper bounds of a given positive formula Φ: Algorithm 7 constructs *all* MUBs for a formula Φ by starting with a pessimistic upper bound $\Phi_U = \bigcup_{v \in \mathrm{vars}(\Phi)} \{\{v\}\}$ and refining it recursively: In every recursion step, the witness relationships between clauses in Φ and Φ_U are computed and tighter bounds are obtained by merging or removing clauses in Φ_U if allowed by the witness constraints. The set *mergeCandidates* contains pairs of clauses of Φ_U together with their common witness that may be merged without violating the *upper bound* condition. The algorithm is correct and complete:

Proposition 37. *Given a positive DNF formula, Algorithm 7 enumerates all of its iDNF least upper bounds.*

Proof. Every bound produced by the algorithm is indeed an MUB: The *upper bound* criterion is trivially satisfied by the initial pessimistic bound and remains satisfied, because only non-critical clauses are merged or removed. The base case of the recursion is reached

```
MUB(Positive formula Φ, iDNF upper bound Φ_U)
    ▷ outputs all iDNF MUBs of Φ
    ENUMERATE(Φ, ⋃_{v∈VARS(Φ)} {{v}}))

ENUMERATE(Positive formula Φ, iDNF upper bound Φ_U)
    Φ_0 ← {φ_u ∈ Φ_U : φ_u has no critical witness in Φ}
    mergeCandidates ← {(φ_u, w, φ'_u) | w ∈ Φ; φ_u, φ'_u ∈ Φ_0; w is witness of φ_u and φ'_u}
    if Φ_0 ≠ ∅ then
        foreach φ_u ∈ Φ_0 do                          ▷ Remove clauses without crit. witness
            ENUMERATE(Φ, Φ_U \ {φ_u})
        foreach (φ_u, w, φ'_u) ∈ mergeCandidates do    ▷ Merge clauses without critical witness
            ENUMERATE(Φ, (Φ_U \ {φ_u, φ'_u}) ∪ {{φ_u ∪ φ'_u}})
    else
        V ← vars(Φ) \ vars(Φ_U)
        foreach possible way to extend Φ_U by variables in V without altering the witness
        relations do                                   ▷ Condition 2 in Definition 19
            output Φ_U
```

Algorithm 7: Finding all MUBs of a positive formula Φ.

whenever all clauses in Φ_U have at least one critical witness (*criticality*), i.e., $\Phi_0 = \emptyset$ in the algorithm. Every bound is then expanded to satisfy *maximality*.

For completeness, let Φ_U be an arbitrary MUB and consider the non-deterministic version of Algorithm 7 in which the *foreach*-iterators are replaced with a non-deterministic choice. Then there exists the following sequence of non-deterministic choices in the algorithm that yields Φ_U: First, remove all variables that are not in Φ_U, then merge the clauses that give rise to Φ_U. □

The merging and variable removal steps have to occur intertwined in the algorithm in order to achieve completeness: Consider formula $ax \vee bx \vee ay \vee by$. The MUB $a \vee b$ cannot be obtained if clauses are merged first, and the MUB $ax \vee b \vee y$ cannot be obtained if clauses are removed first.

Although the delay between consecutive MUBs returned by Algorithm 7 is polynomial, the algorithm does not provide a polynomial delay enumeration procedure, since identical bounds can be returned several times as they may be reached via several computation branches.

Example 27. Consider the formula $ab \vee ax \vee xy$. Starting with the usual pessimistic bound $a \vee b \vee x \vee y$, the MUB $ab \vee x$ is found in a computation branch that first merges a and b and then removes y. A second branch might first remove a and y, and then expand clause b by a. A third branch will delete y and then merge a and b. All these branches lead to the same MUB $ab \vee x$. □

4.1.3 1OF lower and upper bounds

We now turn to approximation of nested formulas. Such formulas naturally annotate results of queries with negation in probabilistic databases, as per their construction using the translation in Algorithm 1. While their size is polynomial in the size of the input database, un-nesting them in DNF can lead to formulas of size exponential in the size of the input database. We therefore search for approximations that are nested as well and that can be computed without unfolding the input in DNF. We recall our restriction to positive (or unate) formulas for efficiency reasons. This restriction corresponds here to queries where all occurrences of each relation are either positive or negative.

The iDNF bounds are, by definition, in DNF and thus do not meet our requirements here. However, the 1OF language naturally allows for nested formulas. Consider for instance the unate formula $\Phi = (\bigvee_i x_i) \wedge \bigwedge_j (\neg y_j)$. Then any two clauses in the DNF of Φ would share variables, which means that we can only use one of these clauses as iDNF lower bound. The formula Φ is however in 1OF.

As in the iDNF case, Proposition 26 gives us a useful tool to find 1OF bounds. A practical caveat is that it talks about clauses in the DNF representation of a formula, whereas we deal here with nested formulas. We therefore consider an efficient heuristic for computing (a) lower bounds by dropping clauses, in particular by setting one of their literals to false, and (b) upper bounds by dropping literals from clauses, in particular by setting these literals to true. The following proposition formalises this intuition.

Proposition 38. *Let Φ be a positive formula, x be a variable in Φ, and Φ_L and Φ_U be formulas obtained from Φ by replacing one occurrence of x by $\bot$ and respectively $\top$. Then Φ_L and Φ_U are lower and respectively upper bounds of Φ.*

The proof is given in Section 4.5. We can hence obtain lower and upper *1OF bounds* of a positive (or equivalently, unate) formula by setting all but one occurrence of each distinct variable to false and respectively to true. Depending on which occurrences are chosen, we may obtain different 1OF bounds.

Example 28. The formula $\Phi = (x_1y_1 \vee \neg x_2y_2)[x_1(y_1 \vee y_3) \vee x_3(y_4 \vee y_5)]$ is not in 1OF since x_1 and y_1 occur twice. Setting the second occurrences of both these variables to false results in the 1OF lower bound $(x_1y_1 \vee \neg x_2y_2)x_3(y_4 \vee y_5)$. For an 1OF upper bound, we set the same occurrences of the two variables to true and obtain $x_1y_1 \vee \neg x_2y_2$.

Different bounds are obtained by setting different occurrences of the two variables to false and true, respectively. For example, setting the first occurrences of x_1 and y_1 to false results in the lower bound $(\neg x_2y_2)[x_1(y_1 \vee y_3) \vee x_3(y_4 \vee y_5)]$ and the upper bound $x_1(y_1 \vee y_3) \vee x_3(y_4 \vee y_5)$. □

4.2 Probability computation via decomposition

This section introduces our probability computation algorithm for propositional formulas. We first give in Section 4.2.1 an exact version and then lift it to a memory-efficient anytime approximation algorithm in Section 4.2.2. Considerations for efficient implementation are given in Section 4.2.2.4.

4.2.1 Exact probability computation

The key idea behind our algorithm is that the probability of formulas $\Phi \wedge \Psi$ and $\Phi \vee \Psi$ can be efficiently computed from the probabilities $P(\Phi)$ and $P(\Psi)$ of the subformulas Φ and Ψ, if these subformulas are independent or inconsistent:

- if Φ and Ψ are independent, then

$$P(\Phi \wedge \Psi) = P(\Phi) \cdot P(\Psi)$$
$$P(\Phi \vee \Psi) = 1 - (1 - P(\Phi)) \cdot (1 - P(\Psi))$$

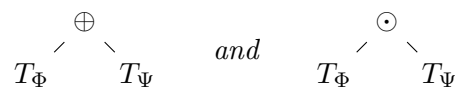
- if Φ and Ψ are inconsistent (i.e., there is no valuation of the random variables on which both are true: the disjunction is *exclusive*), then

$$P(\Phi \wedge \Psi) = 0$$
$$P(\Phi \vee \Psi) = P(\Phi) + P(\Psi).$$

The above equations can be derived from Equation (2.3). We use explicit notation to denote independence and mutual exclusiveness: $\oplus$ for independent-or, $\odot$ for independent-and and $\sqcup$ for exclusive-or. A *decomposition tree* is an alternative representation of formulas that makes independent and exclusive relationships between sub-formulas explicit.

Definition 20 (Decomposition Tree)**.** *A decomposition tree (d-tree) is recursively defined as follows:*

- *Every propositional formula Φ is a d-tree T with a single node representing Φ.*
- *If T_Φ and T_Ψ are d-trees representing independent formulas Φ and Ψ, then*

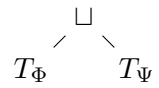


  are d-trees representing $\Phi \vee \Psi$ and $\Phi \wedge \Psi$, respectively.

- *If T_Φ and T_Ψ are d-trees representing mutually exclusive formulas Φ and Ψ, then*

COMPILE (Formula Φ)
  RemoveRedundantClauses (Φ)
  if Φ *is a literal or* $\top$ *or* $\bot$ **then**
    return Φ
  if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 \vee \Phi_2 = \Phi$ **then**
    return COMPILE(Φ_1) $\oplus$ COMPILE(Φ_2)
  if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 \wedge \Phi_2 = \Phi$ **then**
    return COMPILE(Φ_1) $\odot$ COMPILE(Φ_2)
  $\triangleright$ Apply Shannon expansion:
  Choose variable $x \in \mathbf{X}$ occurring in Φ according to elimination heuristics
  return $\big(x \odot \text{COMPILE}(\Phi|_{x \leftarrow \top})\big) \sqcup \big(\neg x \odot \text{COMPILE}(\Phi|_{x \leftarrow \bot})\big)$

Algorithm 8: Compilation of formulas into d-trees.



is a d-tree representing the formula $\Phi \vee \Psi$.

A d-tree in which all leaves are literals is complete.

A d-tree can also be seen as the parse tree of a formula composed of propositional formulas (with $\vee, \wedge, \neg$) and independent and mutual exclusive operators $\oplus, \odot, \sqcup$.

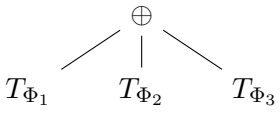
Example 29. Consider the formula $(x \vee y) \wedge ((z \wedge u) \vee (\neg z \wedge v))$. It is easy to verify that this formula satisfies the independence and mutual exclusiveness properties expressed by the equivalent partial d-tree $(x \oplus y) \odot (zu \sqcup \neg z v)$ and the complete d-tree $(x \oplus y) \odot \big((z \odot u) \sqcup (\neg z \odot v)\big)$. □

Definition 21 (Probability of D-tree)**.** *The probability of a d-tree* T *is recursively defined on its structure:*

- *If* T *is a leaf node for formula* Φ*, then* $P(T) = P_\Phi$
- *If* $T = T_\Phi \oplus T_\Psi$*, then* $P(T) = 1 - (1 - P(T_\Phi))(1 - P(T_\Psi))$
- *If* $T = T_\Phi \odot T_\Psi$*, then* $P(T) = P(T_\Phi) \cdot P(T_\Psi)$
- *If* $T = T_\Phi \sqcup T_\Psi$*, then* $P(T) = P(T_\Phi) + P(T_\Psi)$

Example 30. The probability of formula $(x \vee y) \wedge ((z \wedge u) \vee (\neg z \wedge v))$ and its d-tree $(x \oplus y) \odot (z \odot u \sqcup \neg z \odot v)$ is $\big(1 - (1 - P_x) \cdot (1 - P_y)\big) \cdot \big(P_z \cdot P_u + P_{\neg z} \cdot P_v\big)$. □

Since the operations $\oplus, \odot$ are associative, the independent decompositions readily extend to decompositions into more than two independent formulas; for example $\Phi_1 \vee \Phi_2 \vee \Phi_3$ (with Φ_1, Φ_2, Φ_3 independent) can be decomposed into



The probability of a d-tree can be computed efficiently, viz:

Proposition 39. *Given the probabilities of all the leaves of a d-tree, its probability can be computed in time linear in its size (assuming unit-cost arithmetic operations).*

Proof. It follows immediately from the definitions of $\sqcup$, $\oplus$, and $\odot$ that the operations required at each node can be computed in constant time. The probability of the d-tree can thus be computed in one bottom-up pass over the tree in time linear in the number of its nodes. □

Any formula can be compiled into a d-tree by finding independent subformulas and decomposing with respect to $\oplus$ and $\odot$, and otherwise by applying Shannon expansion to yield a mutually exclusive $\sqcup$-decomposition. Algorithm 8 achieves this generic compilation approach, where the compilation is exhaustive, i.e., the leaves of the d-tree represent literals. This is refined in Section 4.2.2, where we describe how a partial compilation of d-trees gives rise to approximate probability computation. The notations $\Phi|_{x\leftarrow\top}$ and $\Phi|_{x\leftarrow\bot}$ denote formulas obtained by replacing every occurrence of variable x in Φ by $\top$ and respectively $\bot$.

Example 31. Figure 4.1 shows a formula and its complete d-tree obtained by executing Algorithm 8 to completion. □

Algorithm 8 is correct, i.e.:

Proposition 40. *Let Φ be a formula and T be the d-tree returned by* Compile*(Φ). Then the formula represented by T is equivalent to Φ and $P_\Phi = P(T)$.*

Proof. The statement follows directly from the recursive definition of the represented formula and from the definition of the probability of a d-tree (Definitions 20 and 21). □

Any formula can be decomposed into a complete d-tree, since Shannon expansion (a.k.a. variable elimination in the Davis-Putnam algorithm for SAT solving [27]) can always be applied and leads to formulas with fewer variables. We can compute the probability of the input formula by inlining the equations from Definition 21 in Algorithm 8.

The order of the variable choices in Shannon expansion influences the size of the d-tree. In general, the compilation of a formula creates a d-tree of exponential size, and it is important to find compilation strategies that lead to d-trees of small sizes [26, 55]. For tractable queries, it is possible to efficiently identify optimal variable orders [62, 61, 63]. For non-tractable queries, we choose a variable that occurs most frequently in the formula. This choice is a heuristic that aims at creating independent sub-formulas by eliminating those variables early that most entangle different parts of the formula.

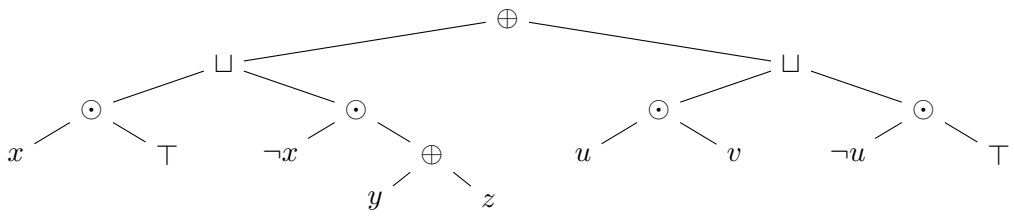


Figure 4.1: D-tree for $\Phi = x \vee \neg xy \vee \neg xz \vee uv \vee \neg u$.

We next analyse the complexity of Algorithm 8. All three decompositions steps can be done efficiently. Shannon expansion requires linear time for each subformula. The independent-or and independent-and partitioning finds connected components in the dependency graph of the input formula Φ. Given $\Phi = \bigwedge_i \Phi_i$ or $\Phi = \bigvee_i \Phi_i$, the dependency graph of Φ has one node for each subformula Φ_i and there is an edge between two nodes Φ_i and Φ_j if $\text{vars}(\Phi_i) \cap \text{vars}(\Phi_j) \neq \emptyset$. A more effective independent-and partitioning can be achieved if Φ is in DNF, since we can apply existing algebraic factorisation of DNF formulas [11]. For a relational encoding of DNF formulas of arity k and n clauses, as used in query evaluation on probabilistic databases [6], this factorisation is unique and requires time $O(k \cdot n \cdot \log n)$ [64].

We close this section with some observations regarding tractable classes of conjunctive queries without self-joins and their connection to linear-size d-trees [60, 62, 61, 63].

Firstly, the annotations of the results of any tractable conjunctive query without self-joins on a tuple-independent database are positive DNF formulas equivalent to formulas in 1OF and can be compiled in polynomial time into complete d-trees of size linear in the number of variables and whose inner nodes are $\oplus$ and $\odot$ only [60].

Secondly, the annotations of the results of any query in a previously-defined class of tractable conjunctive queries with inequalities on tuple-independent databases are positive DNF formulas that can be compiled in polynomial time into complete d-trees of size linear in the size of the formulas and whose inner nodes are $\sqcup$ only [61].

4.2.2 Approximate probability computation

Let us now turn to approximate probability computation. We start by discussing the correspondence between d-trees and probability bounds for represented formulas, then define our approximation algorithm (Section 4.2.2.2) and refine it to a memory-efficient implementation (Section 4.2.2.3).

4.2.2.1 Lower and upper probability bounds for d-trees

Algorithm 8 and Definition 21 give us a recipe for exact probability computation for any formula, provided we exhaustively decompose it into a d-tree where the leaves are literals with known exact probabilities. By using partial instead of complete decompositions, and

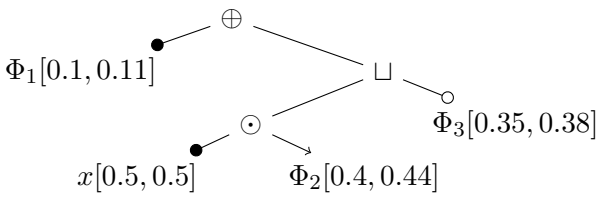


Figure 4.2: Partially decomposed d-tree. Leaves: Φ_1 and x are closed ($\bullet$), Φ_2 is current ($\rightarrow$), Φ_3 is open ($\circ$).

thus by allowing arbitrary formulas at leaves, the same approach yields lower and upper probability bounds for partial d-trees.

We next assume that for each leaf node Φ, we know probabilities P^L_Φ and P^U_Φ such that $P_\Phi \in [P^L_\Phi, P^U_\Phi]$.

Definition 22 (Probability Bounds of D-trees)**.** *The lower and upper probability bounds $P^L(T)$ and $P^U(T)$ of a d-tree T are recursively defined by its structure:*

- *If T is a leaf node for formula Φ, then*

$$P^L(T) = P^L_\Phi, \qquad P^U(T) = P^U_\Phi.$$

- *If $T = T_\Phi \oplus T_\Psi$, then*

$$P^L(T) = 1 - (1 - P^L(T_\Phi))(1 - P^L(T_\Psi))$$
$$P^U(T) = 1 - (1 - P^U(T_\Phi))(1 - P^U(T_\Psi))$$

- *If $T = T_\Phi \odot T_\Psi$, then*

$$P^L(T) = P^L(T_\Phi) \cdot P^L(T_\Psi)$$
$$P^U(T) = P^U(T_\Phi) \cdot P^U(T_\Psi)$$

- *If $T = T_\Phi \sqcup T_\Psi$, then*

$$P^L(T) = P^L(T_\Phi) + P^L(T_\Psi)$$
$$P^U(T) = P^U(T_\Phi) + P^U(T_\Psi)$$

Intuitively, the definition implies that the lower (upper) probability bound of a d-tree is obtained by propagating up the lower (upper) probability bounds of the leaf formulas according to Definition 21. This procedure yields correct bounds:

Proposition 41. *For any d-tree T, $P(T) \in [P^L(T),\ P^U(T)]$.*

APPROX(D-TREE T, ERROR GUARANTEE ϵ, ERROR TYPE type)

$L \leftarrow P^L(T)$
$U \leftarrow P^U(T)$
if TYPE $=$ *absolute* **and** $U - L \leq 2\epsilon$ **then**
  return $[U - \epsilon, L + \epsilon]$
if TYPE $=$ *relative* **and** $(1 - \epsilon) \cdot U - (1 + \epsilon) \cdot L \leq 0$ **then**
  return $[(1 - \epsilon) \cdot U, (1 + \epsilon) \cdot L]$

▷ Approximation not reached; choose and refine a leaf in T
Choose a leaf t of T representing a formula Φ
RemoveRedundantClauses (Φ)
if Φ *is a literal or* $\top$ *or* $\bot$ **then**
  ▷ Do nothing
else if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 \vee \Phi_2 = \Phi$ **then**
  Replace t in T by $\Phi_1 \oplus \Phi_2$
else if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 \wedge \Phi_2 = \Phi$ **then**
  Replace t in T by $\Phi_1 \odot \Phi_2$
else
  Choose variable $x \in \mathbf{X}$ occurring in Φ
  Replace t in T by $(x \odot \Phi|_{x \leftarrow \top}) \sqcup (\neg x \odot \Phi|_{x \leftarrow \bot})$
return APPROX(T, ϵ, TYPE)

Algorithm 9: Approximate probability computation with relative or absolute error guarantee for a partial d-tree T.

Proof. The statement follows immediately from the observation that the operators in Definition 21 are monotonically increasing as a function of the probability of their child nodes. □

Example 32. Consider the partial d-tree $T = \Phi_1 \oplus ((x \odot \Phi_2) \sqcup \Phi_3)$ of Figure 4.2, where the leaves are annotated with their lower and upper bounds. Then, the lower and upper bounds $[P^L(T), P^U(T)]$ of the d-tree can be computed as follows:

$$P^U(T) = 1 - (1 - 0.11) \cdot (1 - (0.5 \cdot 0.44 + 0.38)) = 0.644$$
$$P^L(T) = 1 - (1 - 0.1) \cdot (1 - (0.5 \cdot 0.4 + 0.35)) = 0.595$$ □

An immediate question is how good the bounds of a partial d-tree are and whether one can make guarantees about the approximation quality. We consider two types of such guarantees, given a fixed error threshold $\epsilon \in [0, 1]$:

Definition 23. *A value $\hat{p}$ is an* absolute *(or* additive*) ϵ-approximation of a probability p if $p - \epsilon \leq \hat{p} \leq p + \epsilon$. A value $\hat{p}$ is a* relative *(or* multiplicative*) ϵ-approximation of a probability p if $(1 - \epsilon) \cdot p \leq \hat{p} \leq (1 + \epsilon) \cdot p$.*

We will abbreviate $P^L(T)$ and $P^U(T)$ with L and U if the context is clear. Let us now investigate under which conditions the bounds $[L, U]$ of a given d-tree for a formula Φ are

sufficiently tight to specify an ϵ-approximation of P_Φ. The connection between bounds of d-trees and ϵ-approximations of formulas is given by the following proposition.

Proposition 42. *Let ϵ be a fixed error threshold, and Φ a formula with probability bounds $[L, U]$, i.e., $P_\Phi \in [L, U]$.*

- *If $U - \epsilon \leq L + \epsilon$, then any value in the interval $[U - \epsilon, L + \epsilon]$ is an absolute ϵ-approximation of P_Φ.*
- *If $(1 - \epsilon) \cdot U \leq (1 + \epsilon) \cdot L$, then any value in the interval $[(1 - \epsilon) \cdot U, (1 + \epsilon) \cdot L]$ is a relative ϵ-approximation of P_Φ.*

The proof is given in Section 4.5. A d-tree for a formula Φ is an ϵ-approximation of Φ if its bounds satisfy one of the above sufficient conditions. Both conditions can be checked in linear time in the size of the d-tree, because its lower and upper bounds can be computed in one bottom-up pass, cf. Proposition 39.

4.2.2.2 An anytime approximation algorithm

We next present two algorithms for computing ϵ-approximations of formula probabilities via d-trees. The algorithm detailed in this section resembles Best-first Search in that it incrementally builds a d-tree from a formula by decomposing one of its leaves until the desired approximation is reached. Since the d-tree can be exponential in the number of variables in worst case, the next section presents a refined version that mimics depth-first search and only keeps in memory one root-to-leaf path in the d-tree at any one time.

Algorithm 9 specifies our incremental refinement procedure for approximate probability computation. Starting with the d-tree $T = \Phi$ for a formula Φ, it checks whether T is already an ϵ-approximation for Φ. For this, it computes lower and upper bounds for Φ and its probability. If the desired approximation is reached, it returns the probability interval according to Proposition 42. Otherwise, it picks a leaf of the d-tree, e.g., a leaf with widest probability interval among all leaves, performs one decomposition step, and recurses.

The critical ingredient to this algorithm is the availability of good bounds for formulas at leaves. Given a positive formula Φ, we employ the techniques for model-based formula approximation (cf. Section 4.1) to approximate Φ and hence P_Φ by (i) optimal iDNF bounds if Φ is a DNF formula, and (ii) 1OF bounds if Φ is a nested formula.

In case a leaf represents a non-unate formula, we do not have access to efficiently computable model-based bounds, since deciding their existence is already a hard problem:

Proposition 43. *Let Φ be a formula. Deciding whether there exist non-trivial iDNF formulas $L \neq \bot$ and $U \neq \top$ such that $\mathcal{M}(L) \subseteq \mathcal{M}(\Phi) \subseteq \mathcal{M}(U)$ is NP-hard.*

This follows from hardness of satisfiability and tautology for propositional formulas. We thus cannot even decide efficiently whether the probability of a non-unate formula is 0 or

1. By default, we approach this obstacle by first repeatedly applying Shannon expansion until all leaves represent unate formulas. Once the leaves represent unate formulas, we can compute model-based bounds efficiently as described in Section 4.1.3.

In case of non-unate DNF formulas, such as those annotating results of positive queries on block-independent disjoint tables, we consider a different approach. We compute iDNF lower bounds using Algorithm 5. Such bounds are however not optimal in the iDNF language; for instance, the non-unate DNF formula $xy \vee x\neg y$ is equivalent to x, which represents the iDNF optimal lower and upper bounds, yet Algorithm 5 would output one of the clauses as iDNF lower bound. As upper bound, we compute a probability bound U instead of a model-based bound, as the sum of probabilities of the lower bound returned by Algorithm 5 and of all other clauses in the input DNF formula. If U is larger than 1, we set it to the trivial bound 1.

We close this section with two remarks. Firstly, in order to compute the new bounds $[P^L(T), P^U(T)]$ after a decomposition step, it is not necessary to recompute the probability bounds of the entire d-tree. If we memorise the local probability bounds at each node of T, then it suffices to propagate the new bounds of the decomposed leaf up the d-tree.

Secondly, it is not immediately clear that the overall probability bound of a d-tree improves between two subsequent refinement steps. Consider a leaf node with memorised probability bounds $[L, U]$. The decomposition of this leaf node may result in new probability bounds $[L', U']$ that are no improvement over $[L, U]$, i.e., $[L', U'] \not\subseteq [L, U]$. However, since both bounds are correct, we may memorise and propagate their intersection $[L, U] \cap [L', U'] \subseteq [L, U]$ and thereby obtain an *anytime* algorithm by guaranteeing that the bounds never get worse between subsequent refinement steps.

4.2.2.3 A memory-efficient variant of the decomposition algorithm

Algorithm 9 may construct an exponentially large d-tree before reaching the desired approximation. This memory usage is not feasible for large input. We next present a variant that only keeps in memory one root-to-leaf path of the d-tree at any one time. For exact computation, we accumulate the probability of the d-tree while exploring it depth-first. The explored branches can be safely discarded since their probabilities were already accumulated. The same idea can be used to devise an approximation approach as follows. We also explore the complete d-tree depth-first, but stop as soon as the desired approximation is reached, cf. Proposition 42. For this, we need to maintain a pair of lower and upper probability bounds for the not-yet-explored children of each node along the current root-to-leaf path. While this approximation approach performs better than the exact algorithm since it may avoid the complete exploration of the d-tree, it has one important practical shortcoming: It still needs to explore deep fragments of the d-tree, yet the deeper a branch goes, the more work it requires and the less probability mass it contributes to the probability of the d-tree.

Similarly to Algorithm 9, we would like to traverse a shallow upper part of the d-tree. In contrast to Algorithm 9, we would like traverse the tree depth-first and only keep one path in memory at any one time. This requires a criterion to stop the depth-first exploration when the probability mass represented by the sub-tree of a node is insignificant and can be discarded under the premise that subsequent exploration of the remaining tree can still yield the desired approximation. We next detail this approach; it represents the approximation algorithm implemented in the SPROUT engine.

In the sequel, we call leaves that may be further refined *open*, and leaves that are not further refined *closed*. We need to address the challenge of deriving an ϵ-approximation condition in the presence of closed leaves. Based on this, we can incrementally compile the input formula into a d-tree in depth-first traversal, and decide *locally* whether the current leaf under exploration can be closed or must be refined further. When a leaf gets closed, its bounds are accumulated and it can be removed from the d-tree.

For reasons explained later, we restrict our discussion to d-trees in which at most one child of each $\odot$ node can be closed without computing its exact probability. This constraint does not restrict our encoding of Shannon expansion using $\odot$ nodes, since one of their two children is always a literal for which the exact probability is known.

To understand the worst-case scenario when we want to close a leaf t in a d-tree T, we need to compute the largest bounds interval of T for any possible exact probability that each open leaf may take. If these worst-case bounds fail to satisfy the condition for an ϵ-approximation, then we might not reach the approximation by only refining the open leaves and must thus not close t. Let us formalise this intuition.

Definition 24. *The* bounds space *of a d-tree T is the set of possible bounds $[L, U]$ of T obtained by choosing for each open leaf any point interval between the bounds of that leaf and for each closed leaf Ψ its probability interval $[P_\Psi^L, P_\Psi^U]$.*

Let us denote by $L(T)$ the element of the bounds space obtained by choosing for each open leaf Φ the point interval $[P_\Phi^L, P_\Phi^L]$, where P_Φ^L is a lower bound for Φ.

Lemma 44. *For a d-tree T, $L(T)$ is the pair of bounds $[L, U]$ that maximises $U - L$ for absolute approximation and $(1-\epsilon)\cdot U - (1+\epsilon)\cdot L$ for relative approximation, over the entire bounds space of T.*

The proof is given in Section 4.5. Lemma 44 gives us a strategy to decide whether closing leaves in a d-tree still allows for an ϵ-approximation. This only holds for closing $\oplus$ and $\sqcup$ nodes; for $\odot$ nodes, $L(T)$ does not necessarily maximise the bounds in general. This explains our above restriction for $\odot$ nodes.

Since we can compute $L(T)$ in just one scan of T, finding the maximal values of $U - L$ and $(1 - \epsilon) \cdot U - (1 + \epsilon) \cdot L$ can be done very efficiently. Our main result concerning the closing of leaves is:

Theorem 45. *Let T be a d-tree for a formula Φ, and ϵ be a fixed error. If the bounds $L(T)$ satisfy the sufficient condition for an ϵ-approximation in Proposition 42, then there is a refinement of T that is an ϵ-approximation of P_Φ.*

Proof. The statement follows from Lemma 44 and the fact that refinement eventually leads to completion of T. □

Example 33. Consider the d-tree T of Figure 4.2 and an absolute error $\epsilon = 0.012$. We are at Φ_2 and would like to know (1) whether we can stop with an absolute ϵ-approximation, and in the negative case, (2) whether we can close Φ_2.

(1) We compute the lower and upper bounds of the d-tree as if all the leaves are closed. We plug in the lower bounds of the leaves and obtain $L = 0.1 \oplus ((0.5 \odot 0.4) \oplus 0.35) = 0.595$. Similarly for the upper bound: $U = 0.11 \oplus ((0.5 \odot 0.44) \oplus 0.38) = 0.644$. The condition $U - L = 0.049 \leq 2 \cdot 0.012 = 0.024$ is not satisfied. Hence, we cannot stop now.

(2) We compute $L(T) = [L, U']$, where L is as before and $U' = 0.11 \oplus ((0.5 \odot 0.44) \oplus 0.35) = 0.6173$. We then have that $U' - L = 0.0223 \leq 0.024$. This leaf may be closed. □

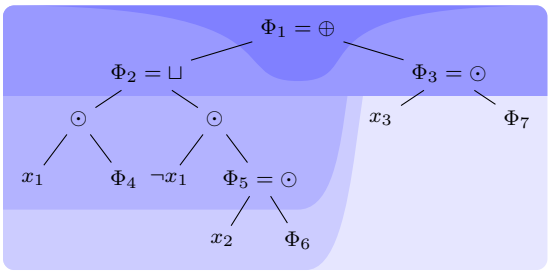
Our incremental algorithm follows the depth-first compilation scheme of Algorithm 8 where instead of constructing the d-tree, node probabilities are computed and propagated recursively. Before constructing a node, we perform two checks: (1) the sufficient condition of Proposition 42 that checks whether ϵ-approximation is already reached, and if this fails, then (2) the condition of Theorem 45 that decides whether the current node to be constructed can be safely closed. This algorithm closes d-tree branches eagerly in order to keep the d-tree height small, while still guaranteeing that the required ϵ-approximation can be reached.

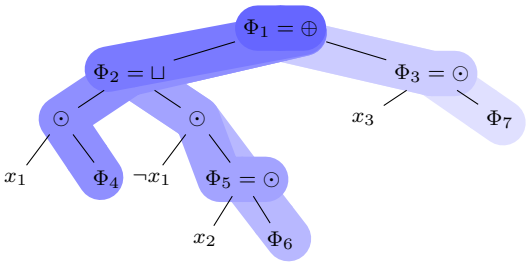
Example 34. We would like to find relative and absolute 0.1-approximations for Φ_1 from Figure 2.3 using our incremental decomposition and iDNF model-based bounds that are factor-k lower bounds from Theorem 33 and upper bounds according to Algorithm 6. For this, we incrementally construct the d-tree in Fig. 4.3 as discussed below.

For numerical calculations, we use the following probabilities: $P_{x_1} = 0.2$, $P_{x_2} = 0.4$, $P_{x_3} = 0.4$, $P_{y_1} = 0.4$, $P_{y_2} = 0.1$, $P_{y_3} = 0.0$, $P_{y_4} = 0.7$, $P_{y_5} = 0.5$, $P_{y_6} = 0.3$, $P_{z_1} = 0.4$, $P_{z_2} = 0.9$, $P_{z_3} = 0.5$, $P_{z_4} = 0.3$.

1st iteration. We start with the d-tree $T_1 = \Phi_1$ and compute model-based bounds L_1 and U_1 that yield probability bounds [0.348, 0.809]. This is not a 0.1-approximation, so we next decompose Φ_1 using independent-or: $\Phi_1 = \Phi_2 \oplus \Phi_3$.

2nd iteration. We obtain bounds [0.348, 0.809] for the probability of the d-tree $T_2 = \Phi_2 \oplus \Phi_3$, where we used the model-based bounds in Figure 4.3 for Φ_2 and Φ_3 with probability bounds [0.276, 0.736] and [0.1, 0.277]. T_2's bounds are as in the 1st Iteration, since independent-or partitioning does not alter iDNF bounds, and are thus not tight enough.





$\Phi_1 = x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2 \vee x_3y_5z_3 \vee x_3y_6z_4$
$= \Phi_2 \oplus \Phi_3$

$\Phi_2 = x_1y_1z_1 \vee x_1y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2$
$= (x_1 \odot \Phi_4) \sqcup (\neg x_1 \odot \Phi_5)$

$\Phi_3 = x_3y_5z_3 \vee x_3y_6z_4 = x_3 \odot \Phi_7$

$\Phi_4 = y_1z_1 \vee y_2z_2 \vee x_2y_3z_1 \vee x_2y_4z_2$

$\Phi_5 = x_2y_3z_1 \vee x_2y_4z_2 = x_2 \odot \Phi_6$

and

$L_1 = x_1y_1z_1 \vee x_2y_4z_2 \vee x_3y_5z_3$
$U_1 = x_1y_1z_1 \vee y_2z_2 \vee x_2y_3 \vee y_4 \vee y_5z_3 \vee x_3y_6z_4$

$L_2 = x_1y_1z_1 \vee x_2y_4z_2$
$U_2 = x_1y_1z_1 \vee y_2z_2 \vee x_2y_3 \vee y_4$

$\Phi_7 = y_5z_3 \vee y_6z_4$
$L_3 = x_3y_5z_3$
$U_3 = y_5z_3 \vee x_3y_6z_4$

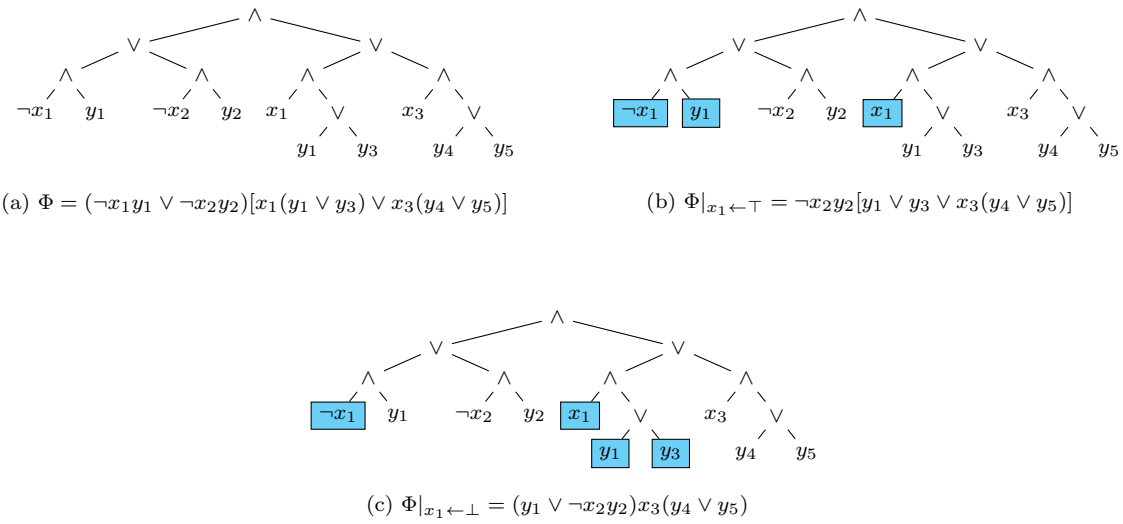
$L_4 = y_1z_1 \vee x_2y_4z_2$
$U_4 = y_1z_1 \vee y_2z_2 \vee x_2y_3 \vee y_4$

$\Phi_6 = y_3z_1 \vee y_4z_2$
$L_5 = x_2y_4z_2$
$U_5 = x_2y_3z_1 \vee y_4z_2$

Figure 4.3: D-tree for formula Φ_1 from Figure 2.3 used in Example 34. Some inner nodes carry an additional label (e.g., $\Phi_2 = \sqcup$) in order to better illustrate the decomposition steps taken. In the left d-tree, coloured areas represent d-trees T_1 (darkest) through T_5 (lightest) discussed in Example 34. In the right d-tree, coloured areas represent root-to-leaf paths as discovered by the algorithm's depth-first exploration: darker paths are discovered earlier, lighter paths are discovered later. At each point in time, the algorithm needs to memorise only one of these paths.

We next check whether we can close Φ_2 and compute the pessimistic bound $L(T_2)$ using the above bounds for Φ_2 and the point interval defined by a lower bound for the open node Φ_3. The pessimistic lower bound $L_2 \oplus L_3$ has probability 0.348 and the upper bound $U_2 \oplus L_3$ has probability 0.809. These bounds fail the stopping condition for both relative and absolute errors, hence we cannot close Φ_2. We further decompose Φ_2 using Shannon expansion by variable x_1.

3rd iteration. The bounds now become [0.348, 0.753] for d-tree $T_3 = ((x_1 \odot \Phi_4) \sqcup (\neg x_1 \odot \Phi_5)) \oplus \Phi_3$, yet no 0.1-approximation is reached. We continue with Φ_4 and first check whether we can close it. The pessimistic bounds are [0.348, 0.420] and suffice for both relative and absolute approximation; we thus close Φ_4. We continue depth-first with Φ_5; this node cannot be closed and we decompose it into $x_2 \odot \Phi_6$.



(a) $\Phi = (\neg x_1 y_1 \vee \neg x_2 y_2)[x_1(y_1 \vee y_3) \vee x_3(y_4 \vee y_5)]$

(b) $\Phi|_{x_1 \leftarrow \top} = \neg x_2 y_2[y_1 \vee y_3 \vee x_3(y_4 \vee y_5)]$

(c) $\Phi|_{x_1 \leftarrow \bot} = (y_1 \vee \neg x_2 y_2) x_3(y_4 \vee y_5)$

Figure 4.4: Masking for Shannon expansion: Parse tree for Φ (Figure (a)), and after masking all occurrences of x_1 to true (Figure (b)) and false (Figure (c)), respectively.

4th iteration. Since Φ_6 is in 1OF, we can directly compute its probability 0.63. The resulting d-tree T_4 has bounds $[0.348,\ 0.534]$ that constitute an absolute 0.1-approximation interval $[0.534 - 0.1,\ 0.348 + 0.1] = [0.434,\ 0.448]$.

Since we have not yet reached a relative approximation, we decompose the next open leaf in depth-first traversal without known exact probability, i.e., Φ_3, into $x_3 \odot \Phi_7$.

5th iteration. We detect that Φ_7 is in 1OF and compute its probability 0.318. The resulting d-tree T_5 has bounds $[0.368, 0.438]$ and is thus a relative 0.1-approximation interval $[(1 - 0.1) \cdot 0.438,\ (1 + 0.1) \cdot 0.368] = [0.3942,\ 0.4048]$. □

4.2.2.4 Considerations for an efficient implementation

We next discuss several aspects regarding the efficient implementation of our algorithm. The following desiderata have been considered in the design of the algorithm: (1) The input formula can be very large and copying parts of it after each decomposition step is not feasible. (2) For each decomposition step, we should only touch necessary nodes and as few times as possible. (3) We should use memory-conscious data structures to support our decomposition steps.

Formulas are decomposed into subformulas such that the latter represent syntactic restrictions of the former. This motivates the following compact representation of formulas in our implementation: Whenever a formula Ψ is constructed by decomposing a formula Φ, instead of explicitly storing Ψ, we represent it by a *mask* applied to Φ, where we specify which clauses and literals from Φ make up the new formula Ψ. A mask together with the original formula uniquely identify any subformula obtained by decomposition.

```
MASK(NODE n, BOOL b, ROOT NODE r)
    switch n do
        case leaf node
            n.SETMASKED()
            if n represents positive literal x then
                MASK(n.PARENT(), b, r)
            else
                ▷ n represents negative literal ¬x
                MASK(n.PARENT(), ¬b, r)
        case inner node
            if (b=true and n=∨) or (b=false and n=∧) then
                n.MASKALLLEAVES()
            if n.ALLLEAVESMASKED() and n ≠ r then
                MASK(n.PARENT(), b, r)
```

Algorithm 10: The masking procedure MASK sets a leaf node n to truth value b and then further simplifies the formula. The function n.SETMASKED() masks the node n, the function n.MASKALLLEAVES() masks all nodes that are descendants of n. These two functions are rendered in blue box in order to highlight the connection to the masked (blue) nodes in Figures 4.4 and 4.5.

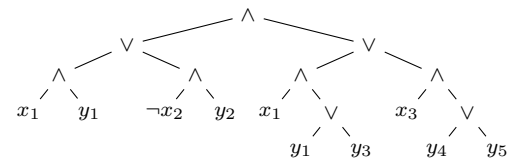
Algorithm 10 depicts the procedure MASK. When called with truth value b on a node n in the parse tree of a formula Φ in NNF, it computes the formula obtained from Φ by setting the variable x in node n to b and propagating $\top$ and $\bot$ in the parse tree of the formula by means of the equivalences $\bot \wedge \Psi \equiv \bot$,$\top \wedge \Psi \equiv \Psi$, $\bot \vee \Psi \equiv \Psi$, and $\top \vee \Psi \equiv \top$.

Example 35. Consider Shannon expansion for variable x_1 in formula Φ from Figure 4.4(a). Instead of constructing data structures for the formulas $\Phi|_{x_1 \leftarrow \top}$ and $\Phi|_{x_1 \leftarrow \bot}$, we can mask the parts of the formula affected by setting x_1 to $\top$ and $\bot$. The result of this masking operation is depicted in Figures 4.4(b) and 4.4(c); when the blue parts are ignored, those parse trees represent the formulas $\Phi_{x_1 \leftarrow \top}$ and $\Phi_{x_1 \leftarrow \bot}$.

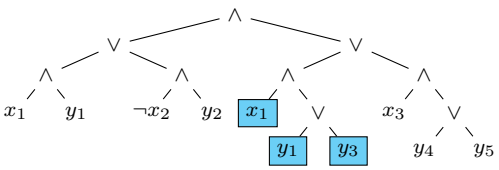
In order to compute $\Phi|_{x_1 \leftarrow \top}$, we need to mask every occurrence of x_1 in Φ with *true*, cf. Figure 4.4(a). We hence mask the leaf $\neg x_1$; since it is unsatisfiable under $x_1 \leftarrow \top$ and since its parent p is $\wedge$, we mask all other leaves under p in order to propagate the identity $\bot \wedge \Psi \equiv \bot$. We move up to the parent of p. This is a $\vee$-node with unmasked leaves and we stop. We then mask the second leaf of x_1. We stop since its parent is a $\wedge$-node that has un-masked leaves. The masked tree is shown in Figure 4.4(b) and represents $\Phi|_{x_1 \leftarrow \top}$.

To compute $\Phi_{x_1 \leftarrow \bot}$, we mask for all occurrences of x_1 to false: the leftmost leaf $\neg x_1$, and then the second leaf of x_1 together with all the other leaves of its parent. □

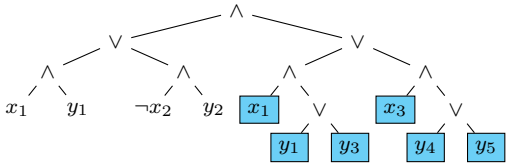
Besides its use to encode subformulas obtained by decomposition, this masking procedure can also be used to compute 1OF bounds for formulas as defined in Section 4.1.3.



(a) $\Phi = (x_1y_1 \vee \neg x_2y_2)[x_1(y_1 \vee y_3) \vee x_3(y_4 \vee y_5)]$



(b) Lower bound $(x_1y_1 \vee \neg x_2y_2)x_3(y_4 \vee y_5)$



(c) Upper bound $x_1y_1 \vee \neg x_2y_2$

Figure 4.5: Masking for bounds: Parse tree for Φ (Figure (a)), and for bounds of Φ obtained by masking the second x_1 and y_1 to false (Figure (b)) and true (Figure (c)), respectively.

Example 36. 1OF lower and upper bounds for unate formulas can be obtained by setting all but one occurrence of each variable to false and true, respectively. This can be implemented using masking: Both x_1 and y_1 occur twice in formula Φ of Figure 4.5(a); masking the second occurrences of x_1 and y_1 to false (true), yields a lower (upper) bound, cf. Figures 4.5(b) and 4.5(c), respectively. □

Compared to an alternative implementation that explicitly creates and passes copies of decomposed and simplified sub-trees to recursive calls, our approach of masking leaves has experimentally proven very effective in capturing the first two above-mentioned efficiency desiderata. To further speed up the masking procedere, and in response to the third desideratum, we record at each inner node the start and end index of the leaf nodes under that node in the list of all leaf nodes. The masking status of leaves is kept in a bitset, with one bit per leaf. Checking whether all leaves under a given node are masked then requires to check whether a range of bits in a bitset is set. Masking is compositional: subsequent masking can be added to an existing one by a bitwise-or operation.

Efficient independent decompositions are implemented using a linear algorithm for finding connected components in a graph. We keep a bitset to encode which variables occur under each child of an and/or-operation. Checking independence can then be formulated as bitset intersection. If two children are not independent, then the explanation is given by means of variables with bits set in this intersection. Also, 1OF formulas are not decomposed further; instead, we can directly evaluate their probability in one pass.

4.3 Experimental evaluation

In this section, we report on experiments with our approximate and exact techniques and existing techniques used by probabilistic database management systems. We found that our approximation technique is consistently more efficient than sampling-based approximation and offers a reliable, significant performance advantage over exact methods, even for small relative error thresholds. It can also keep pace with state-of-the-art techniques for tractable queries.

Algorithms. We compare the following algorithms:

MC (Monte-Carlo) computes an FPTRAS approximation of result probabilities. The MayBMS implementation that we compare against [44] uses the probabilistic adaptation of Vazirani's version of the Karp-Luby estimator [87] which computes fractional estimates that have smaller variance than the zero-one estimates of the Karp-Luby estimator. It combines this estimator with the Dagum-Karp-Luby-Ross optimal algorithm for Monte Carlo estimation [19], which determines the number of invocations of the Karp-Luby estimator needed to achieve the required bound by running the estimator a small number of times to estimate its mean and variance.

These techniques yield an efficiently computable unbiased estimator that in expectation returns the probability p of a DNF of n clauses such that computing the average of a polynomial number of such Monte Carlo steps (which are calls to the Karp-Luby unbiased estimator) is an (ϵ, δ)-approximation for the probability (i.e., a relative approximation): If the average $\hat{p}$ is taken over at least $\lceil 3 \cdot n \cdot \log(2/\delta)/\epsilon^2 \rceil$ Monte Carlo steps, then $P\big[|p - \hat{p}| \geq \epsilon \cdot p\big] \leq \delta$.

In our experiments, the probabilistic guarantee δ is fixed to 0.0001.

TRACT is the algorithm implemented in SPROUT for exact probability computation of tractable queries [62, 61]. It leverages the query structure to compute the equivalent read-once expression of the query annotation and evaluates the probability of the latter. Since it relies on the existence of read-once expressions for annotations, TRACT cannot evaluate intractable queries.

d-tree and d-tree$^\epsilon$ are the exact and approximate probability computation techniques as presented in this chapter.

Experimental setup. The experiments were performed on an AMD 64 x2 Dual Core Processor 4600+ (2.4GHz), 2GB running Linux 3.2.0-32, gcc 4.6.3, JDK1.6. Our algorithms are implemented in C (for positive queries) and Java/C (for queries with negation). They are integrated in the SPROUT query engine, which is an extension of PostgreSQL8.3.3.

Queries are executed in the psql shell and materialised to disk. Each point in the reported plots represents 25 runs. We used five different seeds to generate random input probabilities; for each seed, we run the query five times and report average wall-clock execution times

after dropping the largest and smallest times. For TPC-H queries, we report the average times and standard deviation over the five seeds. For the other queries, we only report the average wall-clock execution times.

Experiment design. We provide insight into the performance of our techniques across a variety of queries.

Queries. We consider four classes of queries: tractable TPC-H conjunctive queries, intractable (#P-hard) TPC-H conjunctive queries, intractable conjunctive queries that express patterns in graphs, and TPC-H queries with negation and self-joins. Queries and statistics for formulas annotating their results are given in Appendix A.

Probabilistic databases. We generated tuple-independent TPC-H databases up to scale factor 1 using TPC-H 2.7.0 and annotate tuples with distinct Boolean random variables whose probabilities are drawn uniformly at random. We also used block-independent-disjoint tables representing graphs as edge relations with two records for each edge encoding the probability for being in and missing from the graph. The queries under consideration also generate complex and highly correlated formulas that annotate the query results.

Easy-hard-easy pattern. An easy-hard-easy pattern similar to those in combinatorial algorithms for propositional satisfiability and constraint satisfaction has been observed in the context of query evaluation in probabilistic databases [55]: When the ratio of the number of variables to formula size is large, the formula readily decomposes into independent parts. Conversely, if there are only few variables in a large formula, then a few Shannon expansion steps completely decompose the formula. In both cases, satisfiability, model-counting, and probability are efficiently computable. However, in-between there is a critical region of variable-to-formula-size ratios where probability computation is hard. There is hence a pitfall in increasing the instance sizes in experiments: If we do not proportionally add interesting variability (and increase the probability space), then the instances get easier rather than harder. On the other hand, an easy-hard-easy pattern is also good news, because it shows that hard instances are only restricted to a narrow section of the space of possible input instances.

Absolute versus relative approximation. For probabilities close to 1, our algorithm behaves similarly for both absolute and relative approximation. To study relative approximation, we thus have to construct instances with small result probabilities. As pointed out in the previous paragraph, this is not entirely trivial. However, understanding the properties of relative approximation for our algorithm is important, since relative approximation is a staple of MC.

Experiments with tractable TPC-H conjunctive queries

We consider tractable conjunctive queries without self-joins (1, 15, 1B, 6B, 16B, 17B) and with inequality joins (IQ1B, IQ4B, IQ6) from the literature [62, 61]; queries marked with

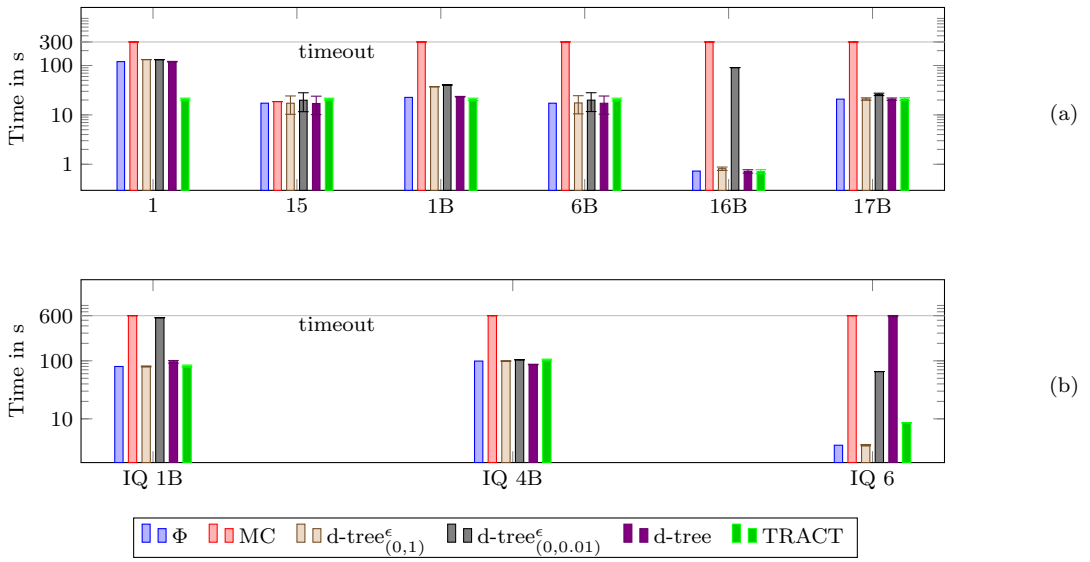


Figure 4.6: Experiments with tractable TPC-H queries on tuple-independent TPC-H data with scale factor 1. The graphs compare Monte-Carlo (MC) and d-tree-based (d-tree$^{\epsilon}$) relative ϵ-approximations with exact d-tree-based (d-tree) and TRACT performance. For the experiments d-tree$^{\epsilon}_{(0,0.01)}$ and d-tree$^{\epsilon}_{(0,1)}$, the probability of each tuple being present in the database is drawn from the intervals $(0, 0.01)$ and $(0, 1)$, respectively. The time taken to compute the result tuples and their annotations is depicted in column Φ. The x-axis specifies the queries used. In queries in (a), aggregations and inequality joins are dropped; queries in (b) use inequality joins. All approximations are relative with error $\epsilon = 0.01$.

"B" are Boolean, i.e., they return the probability that the query is true. These are modified versions of standard TPC-H queries without negation and aggregation but with special aggregates for probability computation. The queries 1, 1B, 6B are selections on the large lineitem table, all other queries are joins of two large tables (e.g., lineitem with supplier, orders, or part). Each query IQ1B, IQ4B, and IQ6 joins two relations using an inequality join and an equality join.

Figure 4.6 shows the running times of these queries on tuple-independent probabilistic TPC-H databases of scale factor 1. As expected, TRACT performs best in most cases, since it is specifically designed to tackle tractable queries. MC times out in all cases but query 15 (where the result annotation consists of three clauses), as it does not exploit the structure of the annotations for tractable queries for efficient evaluation; it needs a number of runs proportional to the number of clauses, even if these clauses are pairwise independent and allow for trivial probability computation.

The following aspects influence the performance of TRACT versus d-tree:

- Main-memory versus secondary-storage. TRACT is a secondary-storage algorithm and factorises the annotations of result tuples into their equivalent read-once form; this process can potentially require multiple passes over the data. The number of passes depends on the structure of the query only and increases with the number of many-to-many joins. In the d-tree algorithm, the annotations of result tuples are computed alongside standard, deterministic query evaluation, while the probability computation algorithm of d-tree operates in main memory only. Consequently, queries that induce only simple entanglement between annotations favour d-tree over TRACT, because the former algorithm can resolve the correlations quickly in main-memory while the latter algorithm may require multiple (secondary-storage) passes over the data to compute the factorisation. This behaviour is observed for queries 15, 1B, 6B, 16B, IQ 1B, and IQ 4B. Conversely, queries whose annotations can be factorised into read-once form with only a few scans favour TRACT, for example queries 1 and IQ 6.

- Sensitivity to input probabilities. When the input probabilities are randomly drawn from the much smaller interval (0,0.01), d-tree$^{\epsilon}$ needs to work harder and dig deeper in the decomposition tree to gain probability mass and it hence makes a small progress with each decomposition step, e.g., for queries 16B, IQ1B, and IQ6. In particular, for input probabilities drawn from (0,1), the probability of query 16B is very close to 1 and d-tree$^{\epsilon}$ converges quickly; for input probabilities drawn from (0,0.01), the query probability is approximately 0.47 and d-tree$^{\epsilon}$ takes longer to converge. On the other hand, the performance of TRACT is independent of the input probabilities.

- Early termination when an approximation is reached. Depending on the input probabilities and the required error threshold ϵ, d-tree$^{\epsilon}$ can be better than TRACT, if it

reaches an approximation in less than linearly many decomposition steps, as required by TRACT. This is the case for queries 15, 6B, and IQ6.

- Overhead due to bound computation. The approximation algorithm d-tree$^{\epsilon}$ computes bounds of d-tree leaves after every decomposition step in order to decide whether it can terminate. This explains why the exact d-tree algorithm can be faster than the approximate algorithm for queries that give rise to particularly simple annotations: While d-tree$^{\epsilon}$ "wastes time" computing bounds, d-tree decomposes the annotation completely and terminates.

- Static versus dynamic analysis. In case of query 1 (and its Boolean version, 1B), the annotation formula is a disjunction of independent literals; by static analysis of the query, TRACT expects this and can efficiently compute the overall probability, whereas d-tree first need to discover the independence of the literals.

In summary, the plots show that our d-tree-based exact and approximation algorithms are often very competitive as they perform about the same or even better than TRACT in some cases. The algorithms are able to detect the tractable structure in the input without knowledge of the query; this demonstrates that the variable elimination heuristics discussed in Section 4.2.1 are effective in decomposing the annotations of tractable queries. Nevertheless, TRACT can be considered the more stable algorithm, because it is insensitive to variations in the input data, and its performance can be predicted accurately by static analysis of the query. When the d-tree algorithms need visibly much more time than TRACT, it is due to the overhead of searches for independence partitioning of the formulas at the leaves and, in case of approximation, also of lower and upper bounds computation.

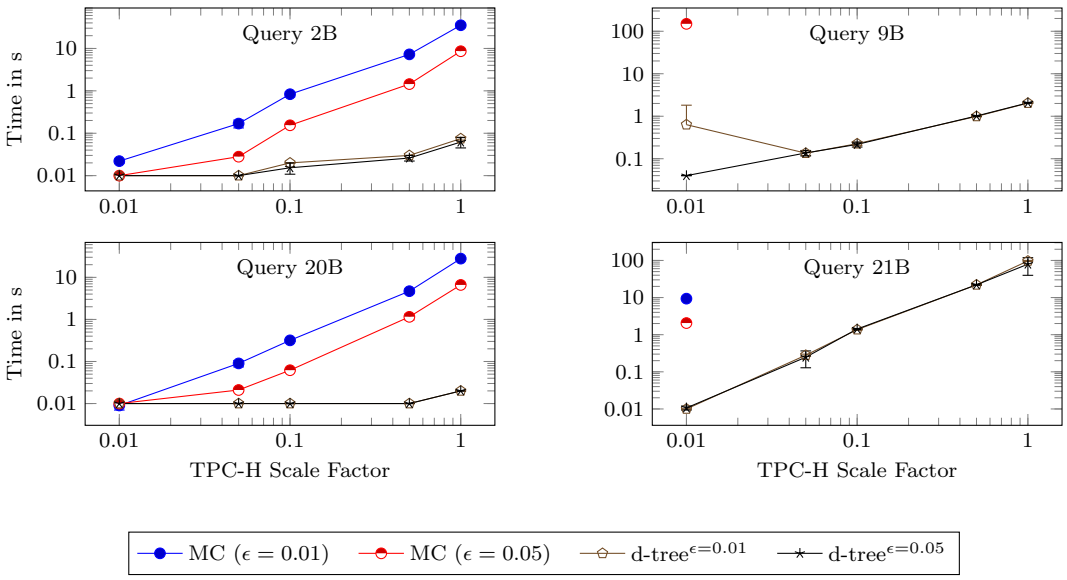
A further observation is that most of the execution time is dedicated to constructing the result tuples and their annotations (leftmost bar, Φ). TRACT even outperforms Φ in some cases since it can effectively use PostgreSQL pipelining mechanism to access annotations as they are produced and need not wait for the entire result to materialise.

The standard deviation is small for the exact and approximation algorithms when run over data with five different seeds for input probabilities. This suggests that similar input probability distributions lead to similar performance of our algorithms.

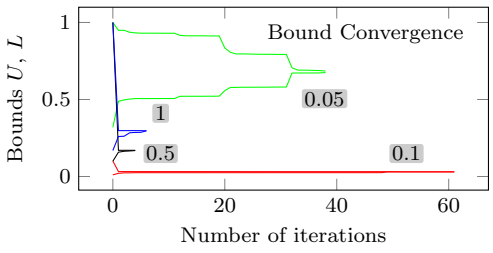
Experiments with intractable TPC-H conjunctive queries

We consider four Boolean intractable conjunctive queries: Query 20B is a join on supplier, nation, partsupplier, and part; query 21B is a join on supplier, lineitem, orders, and nation; query 2B is a join on part, supplier, partsupplier, nation, and region; and query 9B is a join on part, supplier, lineitem, partsupplier, orders, and nation.

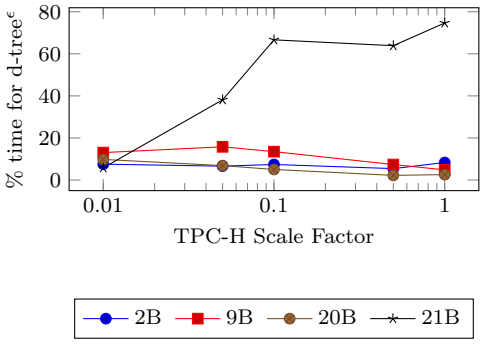
Figure 4.7a depicts the time needed by d-tree$^{\epsilon}$ and MC to compute the approximate probabilities of the four queries. Our algorithm d-tree$^{\epsilon}$ outperforms MC for such queries



(a) Comparison of Monte-Carlo-based (MC) and d-tree-based (d-tree$^{\epsilon}$) relative ϵ-approximation performance for TPC-H scale factors 0.01 through 1; the timeout is set to 300s.



(b) Convergence of lower and upper bounds (y-axis) after each decomposition step (x-axis) for d-tree$^{\epsilon=0.01}$, query 2B and scales 0.05, 0.1, 0.5, 1; scale 0.01 behaves similarly to 0.5.



(c) Percentage of the total query answering time taken by approximate probability computation with d-tree$^{\epsilon=0.01}$.

Figure 4.7: Experiments with intractable TPC-H queries 2B, 9B, 20B, 21B on tuple-independent tables.

on tuple-independent TPC-H data of varying scale factors. These queries have many joins, which leads to low marginal probabilities of clauses, while formulas annotating the results have up to 400 clauses and 1000 variables (query 20B), up to 75,000 clauses and 150,000 variables (query 21B), up to 640 clauses and 1,600 variables (query 2B), and up to 350,000 clauses and 729,000 variables (query 9B).

Statistics collected from runtime traces show that in general, as the size of formulas increase, so does the number of decomposition steps. However, two scenarios may change this trend. Firstly, for bound computation, with more input clauses, both the lower and upper bounds increase while maximal values of upper bounds are 1. If upper bounds reach 1 and lower bounds still increase, this can lead to quick convergence. This is for instance the case for query 9B for scale factor 1. Secondly, the formulas of some TPC-H queries (that have equality selections with constants) have the property that very few variables from one input table occur in most of the clauses. For instance, for queries 20B and 21B, there is only one variable coming from table nation. After eliminating this variable, the residual formula has many independent clauses and the approximation approach captures this and tightens the lower and upper bounds very quickly. Hence, the number of decomposition steps constructed remains small and is not affected by the formula size.

As shown in Figure 4.7c, most of the query execution time for d-tree$^{\epsilon}$ is spent in the first phase of constructing the result tuples and their annotations; except for query 21B, d-tree$^{\epsilon}$ needs less that 10% of the overall execution time.

Figure 4.7b shows that for query 2B, the bounds computed at each decomposition step quickly achieve a relative error of 0.01. The plateau witnessed for scale 0.1 actually represents very small improvements made to both lower and upper bounds over 60 steps. For scales 0.01, 0.5, and 1, it converges in maximum five steps. As mentioned before, the query evaluation problem in probabilistic databases can exhibit an easy-hard-pattern similar to that observed for SAT [55]. The existence of such patterns explains that the number of decomposition steps required does not necessarily increase monotonically with the size of the input database; indeed, Figure 4.7b shows that the query 2B requires fewer iterations for scale factors 0.5 and 1 than for scale factors 0.05 and 0.1.

Experiments with intractable positive graph queries

We next study queries on databases that encode social networks. We consider two classes of datasets modelled as block-independent disjoint tables. The first set consists of synthetic random graphs where all edges have the same probability P_e. An undirected random graph with n nodes is a probabilistic database in which the possible worlds are the subgraphs (obtained by removing zero or more edges) of the n-clique. In the case of $P_e = 1/2$, the probability distribution over this set of possible worlds is uniform and each world has probability $(1/2)^{n\cdot(n-1)}$. The second class of graph datasets are well-known social networks taken from the literature: One is Zachary's classic "Karate club" [93] with 34 nodes, and

the other represents friendship among a group of dolphins. The social networks generalise random graphs in that some edges are missing with certainty and the remaining edges have varying probability of being present in the graph. The idea here is that friendship between nodes is established by observation and there may be a varying degree of confidence in that a pair of nodes are friends (for dolphins), or varying degrees of friendship (for karatekas). The databases used in the experiments are given in Appendix B.

The four queries in our experiments detect the following patterns in graphs: triangles (G_Δ), paths of length 2 (G_{P2}), paths of length 3 (G_{P3}), and pairs of nodes that have at most two degrees of separation (G_S).

Our experimental results for queries on social networks and random graphs are reported in Figures 4.8 and 4.9. The experiments on the Dolphin and Karate social network (Figure 4.8) consider a fixed database and vary the approximation threshold ϵ. The results confirm that d-tree$^\epsilon$ consistently outperforms MC and that — unsurprisingly — the general trend is that tighter approximations require more time. Note however, that in same cases a better approximation can be reached in *less* time. This can be understood as follows: Consider two approximations thresholds ϵ and ϵ' with $\epsilon > \epsilon'$; let us assume that in the case of the (less tight) ϵ-approximation a particular leaf of the d-tree can be closed, while the same leaf needs to be decomposed in order to achieve the (tighter) ϵ'-approximation. If the further exploration of this leaf happens to be computationally easy — for example because the remaining formula decomposes readily into a read-once formula — and at the same time provides a substantial amount of probability mass, then the ϵ'-approximation may in fact be reached earlier than the ϵ-approximation.

In the case of random graphs (cf. Figure 4.9), for large edge probabilities (above 0.5), d-tree$^\epsilon$ converges quickly, since each clause has a non-negligible marginal probability. When we consider smaller edge probabilities (below 0.1), d-tree needs more time to converge, especially for queries involving more joins (such as the path queries). We witness an easy-hard-easy pattern for edge probabilities of 0.3 in case of G_Δ and G_{P2}. As expected, the time required by the MC algorithm scales polynomially with problem size and does not show an easy-hard-easy pattern. Consequently, there exist areas in the problem space that favour MC — typically for small problem sizes, for example for the path query G_{P_2} and graphs of size 12, and for the triangle query G_Δ, $P_e = 0.3$ and graphs between 10 and 25 nodes — and areas that favour d-tree$^\epsilon$ — for example for the path query G_{P_2} and graphs of at least 17 nodes, and for the triangle query G_Δ and graphs with more than 27 nodes.

It is worth pointing out that while the random graphs and social networks used here (on the order of 50 nodes) may not seem very large, they are actually substantial; a 40-nodes random graph has up to 780 edges. The triangle query uses a three-way self-join and generates a formula of 780 variables and 9880 clauses; the path query G_{P2} uses a three-way self-join and generates 60000 clauses; the path query G_{P3} uses a six-way self-join.

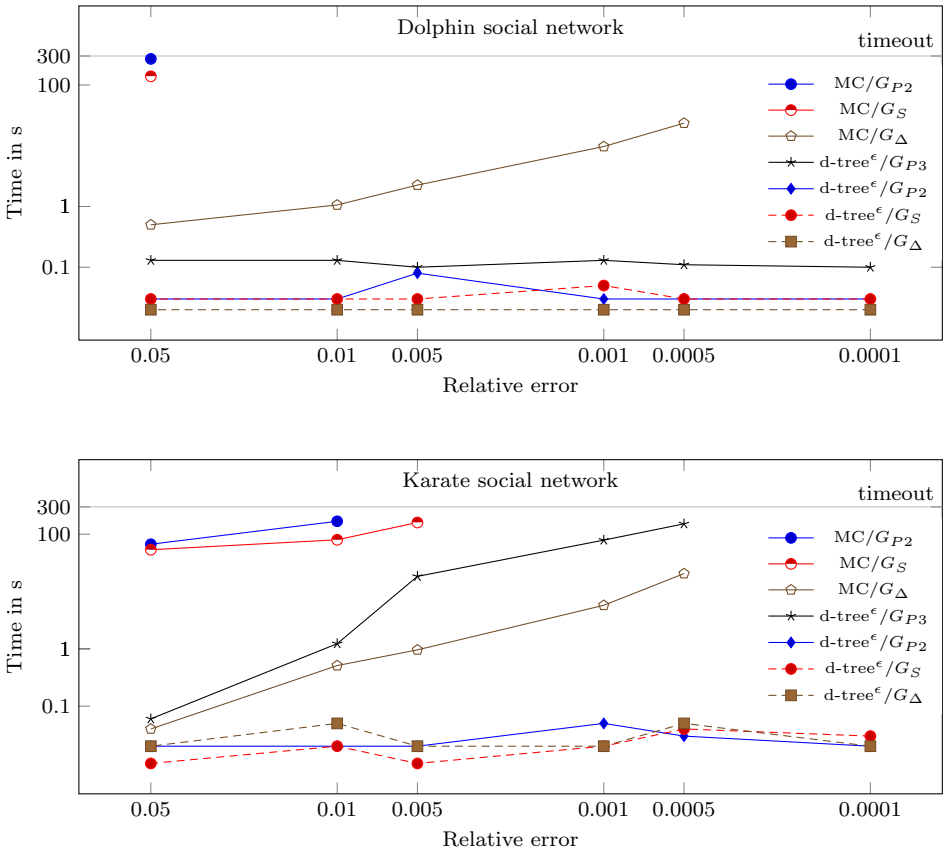


Figure 4.8: Experiments with graph queries on social networks.

Experiments with TPC-H queries with negation

Let us now turn to queries with negation. We investigate four such queries on tuple-independent TPC-H databases: N1 lists all parts that were only ever sold at quantities of at least five per order; N2 lists all parts that were only ever sold at quantities of at least five per order and whose availability is smaller than 8500 at all suppliers; N3 lists all suppliers that offer all parts with certain brand and size; and N4 lists all redundant suppliers, i.e., suppliers for parts that are also available from another supplier.

The number of tuples in the query answers is at least one order of magnitude larger than in the deterministic case. This is because tuples that do not qualify in the answer to a difference operation in the deterministic case, may qualify with a non-zero probability in the probabilistic case. We note that this is specific to queries with difference, and does not happen for positive queries.

Another aspect concerns the size of annotations in query results. For all queries but N3, the average size stabilises when the scale factor increases. This is inherent to TPC-H data: The number of input tuples that participate in the generation of one answer tuple remains roughly constant in case of our queries, regardless of the database size. However, the number of result tuples increases. The size of N3's answer is expected to increase with

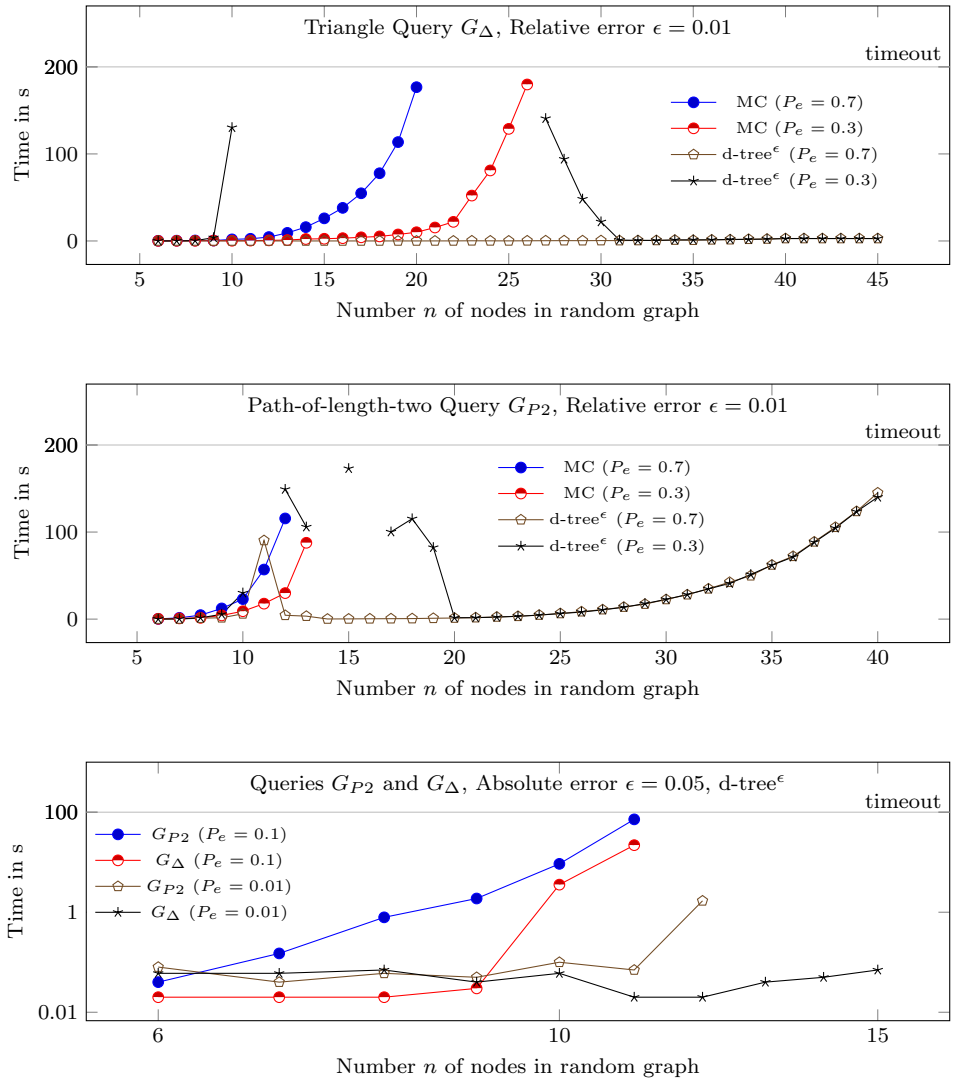


Figure 4.9: Experiments with graph queries on random graphs. P_e is the probability for each edge to be in the graph.

the scale factor as it depends on the number of suppliers.

Regarding running time, the computation of the answer tuples and their annotations is rather close to the deterministic case for our queries, since in both cases the queries have to "touch" all the relevant input tuples and most of the intermediate tuples. The overhead due to the materialisation of more tuples in the answer and to the computation and materialisation of the annotations is below 100% for N1, N2, and N4, but becomes one order of magnitude for N3.

Figure 4.10 compares the d-tree and d-tree$^\epsilon$ algorithms for queries N1, N2, N3, N4 on the TPC-H datasets. Queries N1 and N2 are tractable [34], and both the exact and approximate d-tree algorithms perform within 10 seconds for scale factor 1. Although the answer sizes increase linearly with the scale factor, the average annotation size stays nearly

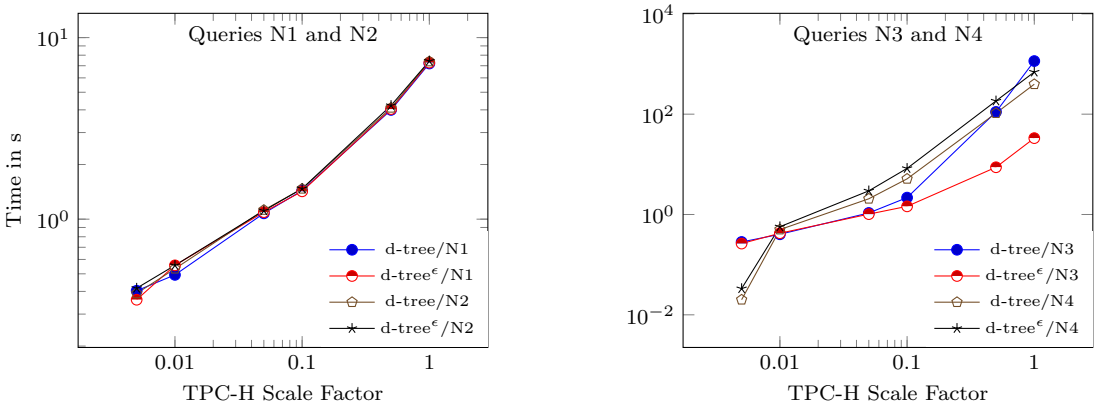


Figure 4.10: Performance of probability computation in the TPC-H scenario for queries with negation. All experiments use a relative error $\epsilon = 10^{-4}$.

constant (between 30 and 40 bytes), thus the performance of probability computation is of the same order of magnitude for all scales. Query N3 exhibits a considerable performance gap between exact and approximation computation: d-tree$^\epsilon$ performs better than d-tree, since it can find good 1OF bounds and stops.

Averaging over runs with different randomly assigned probabilities to input variables yields runtime standard deviation of less than 10% in case of query N3. However, we found this variation to be of the same order of magnitude as the runtime fluctuation over different runs with the same probability assignments to the input variables.

We verified the effectiveness of our heuristic for computing 1OF lower and upper bounds. In all scenarios with hard queries (N3, N4), all computed bounds were non-trivial, except for non-unate formulas where we first had to apply Shannon expansion on the variables occurring with both polarities. Non-trivial means here that the lower bound is not 0, unless the upper bound is also zero, and similarly, the upper bound is not 1, unless the lower bound is also 1.

The lower and upper bounds are also tight. We recorded all bounds over all the experiments, and computed the mean 10^{-4} and standard deviation 10^{-2} of their gaps.

4.4 Discussion

4.4.1 Related work

Model-based Approximations. The closest in spirit to this work on model-based bounds is by Selman on approximating CNF theories by model-based lower and upper bound conjunctions of Horn clauses [75]. The languages iDNF and 1OF used in this chapter are incomparable to conjunctions of Horn clauses and are natural in the context of probability computation due to their efficiency and connection to tractability of query evaluation in probabilistic databases.

The 1OF language has a long history and many names, such as read-once functions [38], fanout-free functions, or non-repeating trees [66], and also many applications including logic synthesis and circuit design [66] and probabilistic databases [60]. Problems such as satisfiability, model counting, and probability computation are hard for general propositional formulas but tractable for 1OF formulas. The equivalent 1OF of any positive DNF formula, if it exists, can be found in polynomial time [38] and is unique up to commutativity of the binary connectives [66]. The 1OF language is particularly relevant in the context of probabilistic databases, since the formulas annotating the results of tractable conjunctive queries without repeating relation symbols on tuple-independent probabilistic databases admit 1OF equivalents [60, 76] and their probability can be computed using relational query plans [21, 62].

Beyond 1OF, tractability of probability computation is lost quickly. It is NP-hard to decide if a positive formula admits an read-twice equivalent formula, i.e., formula where every variable occurs at most twice [29]; this is still open for positive DNF formulas. For read-4 formulas, model counting, and hence probability computation, is #P-hard [83].

A different direction is to consider complete languages to express the bounds, i.e., languages that can represent any propositional formula and that still allow for efficient probability computation. A prime example of such languages is the family of Binary Decision Diagrams, including OBDDs, FBDDs, and d-DNNFs [26]. Such languages allow for several equivalent representations of a formula but with an exponential gap between their sizes. Finding a minimal representation in any of these languages is NP-hard [58]. D-trees generalise ws-trees [55] with independent-and decompositions, which are crucial for the treatment of tractable queries because they capture 1OF formulas [60]. We have also generalised the formalism to partial decompositions, which are the foundation of the approximation techniques in Section 4.2.2. The and/xor trees of [57] are modeled on the ws-trees but are a weaker representation system in that they have tuples, rather than clauses, at their leaves.

An alternative approach to computing upper probability bounds for positive DNF formulas has been proposed in the context of propagation scores of conjunctive queries [36]. These bounds are not model-based, and in particular not optimal. Although the formulas annotating query results can be used to compute upper bounds on their probabilities, their interpretation is non-standard: Each occurrence of a variable v is considered a fresh variable with the same probability as v. Arbitrary positive DNF formulas are thus seen as iDNF formulas. One subtlety regarding applicability of this method to formulas annotating results of queries with self-joins concerns clauses with multiple occurrences of the same variable. Consider the formula $\Phi = (x \vee y) \wedge x$, where x occurs twice. This approach would thus consider instead the formula $\Phi' = (x_1 \vee y) \wedge x_2$, where x_1 and x_2 are distinct but have the probability of x. Since $\Phi = x$, it follows that Φ' is *not* an upper bound of Φ, but in fact a lower bound. Whereas in a DNF representation we could trivially remove such redundancies, this is not always possible for nested formulas without an exponential

blowup in the formula size. A possible approach easy to integrate in our algorithm is as follows: We detect clauses with multiple occurrences of the same variable x by checking whether any two leaves of x have an and-operation as a common ancestor. If this is the case, we replace one of the occurrences by *true* and mask.

Ré and Suciu discuss model-based lower bounds and Taylor/Fourier approximation of positive DNF formulas [71]. Their focus is on compressing a DNF formula by eliminating subformulas that only account for up to a given threshold of its probability mass.

Application to provenance databases. Besides probabilistic databases, our model-based approximations can also benefit provenance management for annotated databases. Similarly to the probabilistic case, formulas annotating query results encode symbolically all possible explanations for the existence of a tuple in the query answer in terms of combinations of the input tuples and can be interpreted as the provenance of query results. Provenance bounds can then represent coarser, more compact, and efficiently computable explanations of the query result. Lower bounds are *correct but possibly incomplete* explanations, while upper bounds are *complete but potentially incorrect* explanations in the following sense: Every explanation for a lower bound is necessarily an explanation for the result; however, there may exist explanations for the result that are not explanations for the lower bound. The situation for upper bounds is symmetric.

Application to relational databases. Existing work on approximate query answering in relational databases considers the use of synopses (histograms, join synopses, and wavelets) to speed-up the evaluation of aggregate queries with the goal of quickly reporting the leading digits of the answers. A survey of these synopsis-based techniques has been recently compiled [17]. Their focus is not on deriving optimal bounds within a given language.

Given a query Q in a query language $\mathcal{L}$, our approximation problem in the relational setting is to compute two queries Q_L and Q_U in a query language $\mathcal{L}'$ such that $Q_L(D) \subseteq Q(D) \subseteq Q_U(D)$ for any relational database D and the computational complexity for $\mathcal{L}'$ is lower than for $\mathcal{L}$. Within this framework, recent work characterises lower bounds for conjunctive queries where the bounds themselves are expressed as conjunctive queries [33, 8].

#SAT. There is a wealth of literature on approximate and exact techniques for model counting used in #SAT solvers [39]. Some of these techniques consider formulas with various restrictions (such as bounded treewidth), or focus on lower-bounding in extremely large combinatorial problems, with bounds off the true count by many orders of magnitude, e.g., [91]. Extensions of the Davis-Putnam procedure (which is based on Shannon expansion) have been used for counting the solutions to formulas [10]. The decomposable Negation Normal Form [26] (and variations thereof) is a propositional theory with efficient model counting, which uses Shannon expansion and independence partitioning. Similar ideas have

been used to design an exact probability computation algorithm for probabilistic databases, although without polynomial-time guarantees for tractable queries [55]. These approaches do not consider model-based approximation of propositional formulas. Our decomposition approach shares ideas with these techniques, yet two of its main aspects remain novel: (1) the combination of incremental compilation and of model-based bounds for fast approximate computation with error guarantees, and (2) polynomial-time evaluation for classes of tractable queries on probabilistic databases.

A different line of research is on randomised approximation algorithms. It was first shown by Karp, Luby, and Madras [51] that there is a fully polynomial-time randomised approximation scheme (FPTRAS) for DNF counting based on Monte Carlo simulation. This algorithm can be modified to compute the probability of a DNF over independent discrete random variables [40, 22, 70, 53]. MystiQ [70] uses an MC variant based on the Karp-Luby unbiased estimator for DNF counting [51] and MayBMS [44] uses the probabilistic adaptation of a version of the Karp-Luby estimator described in [87].

In contrast to our algorithm, Monte Carlo algorithms run in polynomial time yet they have the following limitations: (1) it can only guarantee probabilistically the computed approximation; (2) running one more Monte Carlo step does not necessarily lead to a refinement of probability bounds, and hence the approximation is not truly incremental [79]; (3) it only works for DNF formulas; (4) it sees the input as a black box and does not exploit their structure for faster evaluation [55]. Limitations (1) and (2) have been shown to be a fundamental constraint for its applicability to ranking in probabilistic databases [65]. In order to decide the relative rank of two answer tuples, it suffices to approximate their probability intervals until they are disjoint; this requires the approximation to be both deterministic and incremental. Limitation (3) prohibits the use of the Monte Carlo algorithm for queries with negation. Limitation (4) means that it does not understand tractable queries and cannot distinguish between structurally simple (such as, when the ratio of clauses to variables is very small or very large) and complex formulas.

The work by Karp, Luby, and Madras has started a line of research to de-randomise these approximation techniques, eventually leading to a polynomial time deterministic $(\epsilon, 0)$-approximation algorithm [80] (for k-DNF, i.e., the size of clauses is bounded, which is not an unrealistic assumption for probabilistic databases, where k is bounded by the number of joins for DNFs constructed by positive relational algebra). However, the constant in this algorithm is astronomical: this constant is above 2^{50} for 3-DNF.

Probabilistic databases. Query evaluation is a core task in probabilistic databases and restricted instances of this problem have been much investigated in recent years [79]. Common restrictions consider conjunctive queries without self-joins and tuple-independent databases, e.g., [22, 62]. Beyond these restrictions, there is work on tractability of positive queries [20], on queries with aggregates [72, 30] and with negation [89, 34]. The approximate

probability computation algorithm described in this chapter is used by the SPROUT query engine [31] to complement its optimised exact algorithm for tractable conjunctive queries without self-joins [62]. It considers relational algebra queries and pc-tables that go beyond these restrictions. Decomposition trees have previously been used in the seminal paper on deterministic approximations for positive queries [63]. The approximation algorithms used by the probabilistic database management systems MystiQ [70], MayBMS [44], and MCDB [46] are variations of the Monte Carlo algorithm presented above. Though used for probability computation and not for modelling purposes, the decomposition trees resemble the probabilistic XML model with independence and mutex inner nodes and data values at leaves [4].

More recent work extends the probability computation framework presented in this chapter. ProApproX evaluates queries on probabilistic XML [78], where the formula annotating the query answer is compiled into a restricted d-tree without using Shannon expansion. Using cost-based heuristics, different approximation algorithms, including the Monte Carlo algorithm discussed above, are run at the leaves; the bounds at the leaves are then used to derive overall probability bounds as discussed in Section 4.2. Top-k queries can be evaluated by incrementally approximating the probabilities of the results using our anytime algorithm until the lower bounds of k tuples are greater than the upper bounds of the remaining tuples [65]. Our model-based bounds were also used for first-order formulas, as recently proposed for top-k query evaluation in the presence of non-materialised views [28]. The d-tree formalism can support sensitivity analysis and explanation for queries [50] by efficiently computing the result probability in the face of changes to the input probabilities.

4.4.2 Conclusion and extensions

This chapter presents an approximation algorithm for computing probabilities of propositional formulas over discrete random variables. This is used by the SPROUT query engine to compute confidences in results to relational queries on probabilistic databases. Approximation is key to scalable query evaluation in probabilistic databases since already simple queries have #P-hard data complexity.

The proposed approximation algorithm has two key ingredients: an incremental decomposition of formulas into independent or mutually exclusive subformulas, and an approach to compute model-based bounds on these subformulas and use them to derive lower and upper bounds on the probability of the input formula. Together, the two components give rise to an anytime, memory-efficient approximation algorithm with relative or absolute error guarantees.

Extensions of the decomposition tree formalism and the model-based bounds put forward in this chapter have been used in the context of probabilistic databases for the exact evaluation of top-k queries [65, 28] and of queries with aggregates [30], for approximate query evaluation over probabilistic XML [78], for explaining query tractability via linear-

size d-trees [60, 61, 48], for conditioning probabilistic databases [55] and for explanation and sensitivity analysis [50]. A promising extension of this work is to consider model-bounds with respect to more expressive languages such as read-once formulas or ordered or free binary decision diagrams. The following Chapter 5 generalises the proposed decomposition methods to support queries with aggregates.

4.5 Proofs

This section contains proofs of formal statements in this chapter.

4.5.1 Proof of Proposition 26

Proposition 26. *Let Φ and Ψ be positive DNF formulas. The following statements are equivalent:*

1. $\Phi \models \Psi$*;*
2. Ψ *can be obtained from* Φ *by adding clauses and removing literals from* Φ*'s clauses;*
3. Φ *can be obtained from* Ψ *by removing clauses and adding literals to* Ψ*'s clauses.*

Proof. We first prove (1) $\Rightarrow$ (2); assume $\Phi \models \Psi$. Then by Lemma 1 for each clause $\varphi \in \Phi$ there is a clause $\psi \in \Psi$ such that $\varphi \supseteq \psi$. The following algorithm generates the formula Ψ from Φ by removing variables and adding clauses.

1. For every clause $\varphi \in \Phi$, find a clause $\psi \in \Psi$ such that $\varphi \supseteq \psi$; remove the variables $\text{vars}(\varphi) \setminus \text{vars}(\psi)$ from φ in Φ.
2. Add all clauses from Ψ that are not in Φ (as obtained from step 1.)

Note that this algorithm may produce redundant clauses; for example $ab \vee ac \models a$, and the algorithm would generate the formula $a \vee a$.

The reverse direction — (2) $\Rightarrow$ (1) — follows directly from Lemma 1, since Equation (2.2) is invariant under removing variables and adding clauses.

Finally, (2) and (3) are symmetric — Ψ can be obtained from Φ by adding clauses if and only if Φ can be obtained from Ψ by removing clauses, and similarly for variables — and hence equivalent. □

4.5.2 Proof of Theorem 27

Theorem 27. *Let Φ be a positive DNF formula. An iDNF formula Φ_L is a maximal lower bound for Φ if and only if Φ_L is a greatest lower bound for Φ.*

Proof. Let us first show the direction MLB ⇒ GLB. Let Φ be an irreducible positive DNF formula and Φ_L an MLB for Φ. Assume Φ_L is no GLB for Φ. Then there exists an iDNF formula Φ'_L such that $\mathcal{M}(\Phi_L) \subset \mathcal{M}(\Phi'_L) \subseteq \mathcal{M}(\Phi)$ and the following properties hold:

(i) $\Phi_L \subseteq \Phi$ (every clause in Φ_L is also a clause in Φ)

(ii) Every clause φ in Φ and not in Φ_L contains a conflicting variable, i.e., a variable $x \in \varphi$ such that there exists a clause $\bar{\varphi}_L \in \Phi_L$ with $x \in \varphi_L$ (Φ_L is MLB for Φ)

(iii) $\forall \varphi'_L \in \Phi'_L : \exists \varphi \in \Phi : \varphi'_L \supseteq \varphi$ ($\Phi'_L \models \Phi$ and Lemma 1)

(iv) $\forall \varphi_L \in \Phi_L : \exists \varphi'_L \in \Phi'_L : \varphi_L \supseteq \varphi'_L$ ($\Phi_L \models \Phi'_L$ and Lemma 1)

(v) No two clauses $\varphi, \bar{\varphi} \in \Phi$ satisfy $\varphi \models \bar{\varphi}$ (Φ is irreducible)

Properties (i) and (ii) follow from Φ_L being an MLB for Φ, (iii) – (iv) from $\mathcal{M}(\Phi_L) \subset \mathcal{M}(\Phi'_L) \subseteq \mathcal{M}(\Phi)$. We prove that Φ_L is equivalent to Φ'_L by case differentiation:

Case 1: $\top \in \Phi$. Then since $\top$ has no conflicting variables with any other clause it follows that $\top \in \Phi_L$ and thus we have $\mathcal{M}(\Phi_L) = \mathcal{M}(\Phi'_L) = \mathcal{M}(\Phi)$ which is a contradiction to the assumption that Φ_L is no GLB for Φ.

Case 2: $\top \notin \Phi$. Let $\varphi'_L \in \Phi'_L$ be any clause in Φ'_L and $\varphi \in \Phi$ a clause such that $\varphi'_L \supseteq \varphi$ according to (iii). With respect to property (i), we have the following two cases:

Case 2(a): $\varphi \in \Phi_L$. Let $\bar{\varphi}'_L \in \Phi'_L$ be a clause with $\varphi \supseteq \bar{\varphi}'_L$ according to (iv). Together with $\varphi'_L \subseteq \varphi$ from above we have $\bar{\varphi}'_L \supseteq \varphi \supseteq \varphi'_L$, and since none of $\varphi, \varphi'_L, \bar{\varphi}'_L$ can be the empty clause (see case 1), φ'_L and $\bar{\varphi}'_L$ must share at least one variable. Since $\Phi'_L \in$ iDNF, it follows that φ'_L and $\bar{\varphi}'_L$ are the same clause. Thus $\varphi \supseteq \bar{\varphi}'_L = \varphi'_L \supseteq \varphi$, i.e., $\varphi = \varphi'_L$. It follows that $\Phi_L = \Phi'_L$ which is a contradiction to the assumption $\mathcal{M}(\Phi_L) \subset \mathcal{M}(\Phi'_L)$.

Case 2(b): $\varphi \notin \Phi_L$. According to (ii), there is a variable $x \in \varphi$ such that there exists a clause $\varphi_L \in \Phi_L$ with $x \in \varphi_L$; furthermore, $\varphi_L \in \Phi$ due to (i). From $\varphi'_L \supseteq \varphi$ it follows $x \in \varphi'_L$. From (iv) it follows that there exist a clause $\bar{\varphi}'_L \in \Phi'_L$ such that $\varphi_L \supseteq \bar{\varphi}'_L$. We distinguish two cases:

Case 2(b).i: $x \in \bar{\varphi}'_L$. It follows $\bar{\varphi}'_L = \varphi'_L$, because $\Phi'_L \in$ iDNF. From $\varphi_L \supseteq \bar{\varphi}'_L = \varphi'_L \supseteq \varphi$ it follows that $\varphi_L \supseteq \varphi$. Since $\varphi_L, \varphi \in \Phi$, this is a contradiction to the assumption that Φ is irreducible, property (v).

Case 2(b).ii: $x \notin \bar{\varphi}'_L$. Then according to (iii) there is a $\bar{\varphi} \in \Phi$ with $\bar{\varphi}'_L \supseteq \bar{\varphi}$ and thus $x \notin \Phi$ which in turn implies $\bar{\varphi} \neq \varphi$ because $x \in \varphi$. Transitivity of $\supseteq$ implies $\varphi_L \supseteq \bar{\varphi}$ which is a contradiction to the assumption that Φ is irreducible, property (v).

Secondly, we prove the direction GLB ⇒ MLB. Assume that a GLB Φ_L for Φ is no MLB for Φ. Then at least one of the two MLB properties in Definition 16 is unsatisfied. We show that in either case Φ_L is no GLB for Φ.

Case 1: Assume $\Phi_L \not\subseteq \Phi$. Then Φ_L contains a clause φ_L that is not in Φ.

Case 1(a): If there is a clause $\varphi \in \Phi$ such that $\varphi_L \supseteq \varphi$, then $\varphi_L \supset \varphi$. Let x be a variable that occurs in φ_L but not in φ. Then the iDNF formula obtained from Φ_L by removing the variable x from φ has strictly more models than Φ_L and thus Φ_L is no GLB for Φ.

Case 1(b): If there is no such clause, then $\Phi_L \not\models \Phi$ following Lemma 1 and thus Φ_L is no lower bound and in particular no greatest lower bound for Φ.

Case 2: If there is a clause $\varphi \in \Phi$ such that $\Phi_L \cup \{\varphi\} \in$ iDNF, then $\mathcal{M}(\Phi_L) \subset \mathcal{M}(\Phi_L \cup \{\varphi\})$ because none of the variables in φ occur in Φ_L and thus Φ_L is no GLB for Φ. □

4.5.3 Proof of Proposition 29

Proposition 29. *Let Φ be a positive DNF formula. A formula $\Phi_L \subseteq \Phi$ is an iDNF greatest lower bound for Φ if and only if the clauses of Φ_L represent a maximal independent set in the clause-dependency graph of Φ.*

Proof. Let Φ be a positive DNF formula and $G = (\Phi, E)$ its clause-dependency graph. Then we have the following equivalences:

$\Phi_L \subseteq \Phi$ is an iDNF GLB for Φ

iff All clauses in Φ_L are pairwise independent
and there is no clause in Φ that is independent from Φ_L (By Definition 16)

iff No two clauses in Φ_L share a variable
and all clauses in $\Phi - \Phi_L$ share a variable with some clause in Φ_L

iff No nodes from Φ_L are connected in G
and every node in $\Phi - \Phi_L$ is connected to some node in Φ_L (By Definition 17)

iff Φ_L is a maximal independent set in G (By definition of maximal independent set).

□

4.5.4 Proof of Proposition 30

Proposition 30. *Let Φ be a positive DNF formula and $G = (\Phi, E)$ be its clause-dependency graph. A formula Φ_L is an iDNF greatest lower bound of Φ with the largest probability among all iDNF greatest lower bounds of Φ if and only if Φ_L is a maximum-weight independent set in G.*

Proof. We show the statement by proving that the total orders on formula probabilities and on weighted independent sets in G are compatible under $w(\cdot)$. Let Φ_L and Φ'_L be two iDNF GLBs of Φ. We then have the following sequence of equivalences.

$$
\begin{aligned}
& P(\Phi_L) \geq P(\Phi'_L) \\
\text{iff} \quad & 1 - \prod_{\varphi \in \Phi_L} (1 - \prod_{x \in \varphi} P_x) \geq 1 - \prod_{\varphi \in \Phi'_L} (1 - \prod_{x \in \varphi} P_x) \\
\text{iff} \quad & \prod_{\varphi \in \Phi_L} (1 - \prod_{x \in \varphi} P_x) \leq \prod_{\varphi \in \Phi'_L} (1 - \prod_{x \in \varphi} P_x) \\
\text{iff} \quad & \sum_{\varphi \in \Phi_L} \ln(1 - \prod_{x \in \varphi} P_x) \leq \sum_{\varphi \in \Phi'_L} \ln(1 - \prod_{x \in \varphi} P_x) \\
\text{iff} \quad & \sum_{\varphi \in \Phi_L} -\ln(1 - \prod_{x \in \varphi} P_x) \geq \sum_{\varphi \in \Phi'_L} -\ln(1 - \prod_{x \in \varphi} P_x) \\
\text{iff} \quad & \sum_{\varphi \in \Phi_L} w(\varphi) \geq \sum_{\varphi \in \Phi'_L} w(\varphi) \\
\text{iff} \quad & W(\Phi_L) \geq W(\Phi'_L)
\end{aligned}
$$

It follows that the GLB with the largest probability is the independent set in the clause-dependency graph with maximum weight. □

4.5.5 Proof of Theorem 33

Theorem 33. *Given a positive DNF formula Φ with max-arity k and n clauses, Algorithm 5 constructs an iDNF greatest lower bound Φ_L for Φ in time $O(n^2)$ such that $P(\Phi'_L) \leq k \cdot P(\Phi_L)$ for every iDNF greatest lower bound Φ'_L for Φ.*

The following facts[2] are used to show Theorem 33.

Lemma 46. *Let $k, k_1, \ldots, k_m \in \mathbb{N}^+$ such that*

$$\sum_{i=1}^{l} k_i \leq lk \quad \text{for all } l = 1, \ldots, m.$$

Let $w_1, \ldots, w_m \in \mathbb{R}^+$ such that $w_1 \geq w_2 \geq \cdots \geq w_m$. Then

$$\sum_{i=1}^{m} k_i w_i \leq \sum_{i=1}^{m} k w_i.$$

[2]The proofs for Lemmata 46 and 47 are due to Graham Cormode and more elegant than our original proofs.

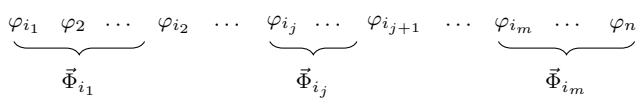


Figure 4.11: Notation used in the proof of Theorem 33.

Proof. Let $w_{m+1} = 0$. Then,

$$\begin{aligned}\sum_{i=1}^{m} k_i w_i &= \sum_{i=1}^{m}\sum_{j=i}^{m}(w_j - w_{j+1})k_i = \sum_{j=1}^{m}\sum_{i=1}^{j}(w_j - w_{j+1})k_i \\ &\leq \sum_{j=1}^{m}(w_j - w_{j+1})jk = k(\sum_{j=1}^{m} jw_j - \sum_{j=1}^{m}(j-1)w_j) = k\sum_{i=1}^{m} w_i.\end{aligned}$$

□

Lemma 47. *Let $k \in \mathbb{N}^+$ and $x \in [0,1)$. Then $\frac{1-x^k}{1-x} \leq k$.*

Proof.

$$1 - x^k = (1-x)\sum_{i=0}^{k-1} x^i \leq (1-x)k,$$

where $x \leq 1$ is used to bound the geometric series. □

Proof of Theorem 33. Let $\Phi = \varphi_1 \vee \cdots \vee \varphi_n$ be a positive DNF formula with max-arity k and assume without loss of generality that the clauses are ordered by descending probability, i.e., $P(\varphi_1) \geq \cdots \geq P(\varphi_n)$. Note that the mapping from clause probabilities to weights in Equation (4.1) is monotonic and thus the clause order $\varphi_1, \ldots, \varphi_n$ is also decreasing with respect to clause weights $w(\varphi_i)$, i.e., $w(\varphi_1) \geq \cdots \geq w(\varphi_n)$. In the light of the proof of Proposition 30, we can reason about the orders of weights and of probabilities equivalently.

Let Φ_L be the formula returned by Algorithm 5. By construction, Φ_L is an MLB and thus an iDNF GLB. Clearly, the algorithm runs in time $\mathcal{O}(|\Phi| \log(|\Phi|))$. It remains to be shown that for any other GLB Φ'_L the inequality $P(\Phi'_L) \leq k \cdot P(\Phi_L)$ holds.

We use the notation depicted in Figure 4.11: Let the lower bound Φ_L contain the m clauses indexed by $i_1, \ldots, i_m$:

$$\Phi_L = \varphi_{i_1} \vee \varphi_{i_2} \vee \cdots \vee \varphi_{i_m}.$$

For every $\varphi_{i_j} \in \Phi_L$ the set of clauses $\vec{\Phi}_{i_j}$ are the clauses between φ_{i_j} (included) and $\varphi_{i_{j+1}}$

(excluded) for $j < m$, and the clauses after φ_{i_j} (included) for $j = m$:

$$\vec{\Phi}_{i_j} = \begin{cases} \{\varphi_{i_j}, \ldots, \varphi_{i_{j+1}-1}\} & \text{if } j < m, \\ \{\varphi_{i_j}, \ldots, \varphi_n\} & \text{if } j = m. \end{cases}$$

The sets $\vec{\Phi}_{i_j}$ are a partition of Φ: Each Φ_{i_j} contains the clause φ_{i_j} picked by the algorithm, and all the following clauses up to (and excluding) the next clause φi_{j+1} picked by the algorithm. Define weights $\tilde{w}(\cdot)$ on the clauses of Φ by $\varphi \in \vec{\Phi}_{i_j} \Rightarrow \tilde{w}(\varphi) = w(\varphi_{i_j})$, i.e., the weights $\tilde{w}$ of the clauses in Φ are as large as possibly allowed by the decreasing order of weights. Clearly $\tilde{w}(\varphi) \geq w(\varphi)$ for all clauses $\varphi \in \Phi$.

Let Φ'_L be any GLB for Φ. Then $\Phi'_L \subseteq \Phi$ and

$$W(\Phi'_L) = \sum_{\varphi \in \Phi'_L} w(\varphi) \leq \sum_{\varphi \in \Phi'_L} \tilde{w}(\varphi).$$

Define integers k_j by

$$k_j = \left| \left\{ \varphi \mid \varphi \in \Phi'_L \text{ and } \varphi \in \vec{\Phi}_{i_j} \right\} \right|.$$

In words: Φ'_L contains k_1 clauses from $\vec{\Phi}_{i_1}$, k_2 clauses from $\vec{\Phi}_{i_2}$, ..., and k_m clauses from $\vec{\Phi}_{i_m}$. Then

$$\begin{aligned} \sum_{\varphi \in \Phi'_L} \tilde{w}(\varphi) &= \sum_{j=1}^{m} \sum_{\varphi \in \Phi'_L \text{ and } \varphi \in \vec{\Phi}_{i_j}} \tilde{w}(\varphi) \\ &= \sum_{j=1}^{m} \sum_{\varphi \in \Phi'_L \text{ and } \varphi \in \vec{\Phi}_{i_j}} w(\varphi_{i_j}) \\ &= \sum_{j=1}^{m} w(\varphi_{i_j}) \sum_{\varphi \in \Phi'_L \text{ and } \varphi \in \vec{\Phi}_{i_j}} 1 \\ &= \sum_{j=1}^{m} k_j w(\varphi_{i_j}). \end{aligned}$$

The existence of the GLB Φ_L imposes the following constraints on the integers k_j:

Constraint 1: For every $j \in [1, m]$, every clause $\varphi \in (\vec{\Phi}_{i_j} \setminus \{\varphi_{i_j}\})$ shares at least one variable with one of $\varphi_{i_1}, \ldots, \varphi_{i_j}$. This is due to the greedy selection in the algorithm that constructs Φ_L.

Constraint 2: For each $l \in [1, m]$, the set of clauses $\bigcup_{j=1}^{l} \vec{\Phi}_{i_j}$ contains at most $k \cdot l$ independent clauses. This can be seen as follows: The l independent clauses $\varphi_{i_1} \ldots \varphi_{i_l}$

contain at most kl different variables, since each clause has max-arity k. Let us identify one bucket with each variable and put each clause in Φ into the buckets corresponding to its variables. It is clear that any two clauses in a given bucket share a variable, i.e., are not independent. The maximal number of independent clauses we can find in $\bigcup_{j=1}^{l} \vec{\Phi}_{i_j}$ is thus kl — one clause from each bucket.

In terms of the k_j this means that for every $l \in [1, m]$ it holds that $\sum_{j=1}^{l} k_j \leq kl$.

By Lemma 46 it follows that

$$\sum_{j=1}^{m} k_j w(\varphi_{i_j}) \leq \sum_{j=1}^{m} k w(\varphi_{i_j})$$

By chasing back the inequalities we finally conclude that $W(\Phi'_L) \leq \sum_{j=1}^{m} k w(\varphi_{i_j}) = W(\Phi_L)$; using the correspondence between weights and probabilities from Equation (2.2), and the probability $P(\Phi_L) = 1 - \prod_{\varphi \in \Phi}(1 - P(\varphi))$ for an iDNF formula Φ_L, we can derive

$$\begin{aligned}
& W(\Phi'_L) \leq k W(\Phi_L) \\
\text{iff} \quad & \sum_{\varphi \in \Phi'_L} w(\varphi) \leq k \sum_{\varphi \in \Phi_L} w(\varphi) \\
\text{iff} \quad & \sum_{\varphi \in \Phi'_L} \ln(1 - P(\varphi)) \geq k \sum_{\varphi \in \Phi_L} \ln(1 - P(\varphi)) \\
\text{iff} \quad & \prod_{\varphi \in \Phi'_L} (1 - P(\varphi)) \geq \prod_{\varphi \in \Phi_L} (1 - P(\varphi))^k \\
\text{iff} \quad & 1 - \prod_{\varphi \in \Phi'_L} (1 - P(\varphi)) \leq 1 - \prod_{\varphi \in \Phi_L} (1 - P(\varphi))^k \\
\text{iff} \quad & \frac{P(\Phi'_L)}{P(\Phi_L)} = \frac{1 - \left[\prod_{\varphi \in \Phi_L}(1 - P(\varphi))\right]^k}{1 - \prod_{\varphi \in \Phi_L}(1 - P(\varphi))} \leq k
\end{aligned}$$

The last inequality is due to Lemma 47 with $0 \leq x = \prod_{\varphi \in \Phi_L}(1 - P(\varphi)) < 1$, which holds since if one assumes non-zero probabilities for variables and thus clauses, i.e., $P(\varphi) > 0$.

For any fixed max-arity k, the time-complexity of Algorithm 5 is indeed quadratic in the number of clauses of the input formula: The for-loop touches every clause exactly once, and the intersection condition $\text{vars}(\varphi_i) \cap \text{vars}(\Phi_L)$ inside the loop can be implemented by comparing the variables of φ_i with at most $k \cdot n$ variables of the clauses of Φ_L. □

4.5.6 Proof of Theorem 34

Theorem 34. *Let Φ be a positive DNF formula. An iDNF formula Φ_U is a minimal upper bound for Φ if and only if Φ_U is a least upper bound for Φ.*

Proof. We use the following graph representation of the witness relationships between clauses: The clauses are the nodes of the graph and there is a directed edge between

two clauses φ and ψ if and only if $\varphi \models \psi$, i.e., φ is a witness of ψ. Figure 4.12 gives an example of such a graph.

MUB $\Rightarrow$ LUB. Assume Φ_U is an MUB for Φ but not an LUB for Φ. Then there exists a better iDNF upper bound Φ'_U for Φ that satisfies $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Phi'_U) \subset \mathcal{M}(\Phi_U)$. Using Lemma 1, the second inclusion unfolds to:

$$\begin{aligned} & \mathcal{M}(\Phi'_U) \subset \mathcal{M}(\Phi_U) \\ \text{iff} \quad & \mathcal{M}(\Phi'_U) \subseteq \mathcal{M}(\Phi_U) \text{ and not } \mathcal{M}(\Phi_U) \subseteq \mathcal{M}(\Phi'_U) \\ \text{iff} \quad & \forall \varphi'_U \in \Phi'_U : \exists \varphi_U \in \Phi_U : \varphi'_U \supseteq \varphi_U \\ & \qquad \text{and} \quad \exists \varphi_U \in \Phi_U : \forall \varphi'_U \in \Phi'_U : \neg(\varphi_U \supseteq \varphi'_U). \end{aligned}$$

By collecting the assumptions and unfolding the definitions:

(i) $\forall \varphi \in \Phi : \exists \varphi'_U \in \Phi'_U : \varphi \supseteq \varphi'_U$ $(\Phi \models \Phi'_U)$

(ii) $\forall \varphi \in \Phi : \exists \varphi_U \in \Phi_U : \varphi \supseteq \varphi_U$ $(\Phi \models \Phi_U)$

(iii) $\forall \varphi'_U \in \Phi'_U : \exists \varphi_U \in \Phi_U : \varphi'_U \supseteq \varphi_U$ $(\Phi'_U \models \Phi_U)$

(iv) $\exists \varphi_U \in \Phi_U : \forall \varphi'_U \in \Phi'_U : \neg(\varphi_U \supseteq \varphi'_U)$ $(\Phi_U \not\models \Phi'_U)$

(v) There is no clause $\varphi_U \in \Phi_U$ that can be extended by a variable from vars(Φ) and the resulting formula is still in iDNF and implied by Φ

(vi) Every $\varphi \in \Phi$ is a witness for at least one clause $\varphi_U \in \Phi_U$, i.e., $\Phi \models \Phi_U$ according to Lemma 1

(vii) Every clause $\varphi_U \in \Phi_U$ has a critical witness in Φ.

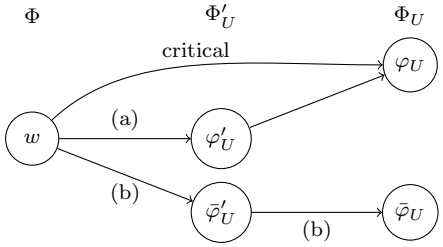
Sentences (i)–(iv) are due to the assumption that Φ_U is upper bound but no LUB for Φ and sentences (v)–(vii) are the syntactical characterisation of the MUB property of Φ_U.

Let $\varphi_U \in \Phi_U$ be as in (iv). Let $\varphi'_U \in \Phi'_U$ be a clause in Φ'_U that shares a variable with φ_U. If no such clause exists, then by transitivity of $\supseteq$ and $\Phi_U, \Phi'_U \in$ iDNF, Φ_U cannot have a witness in Φ which is a contradiction to (vii).

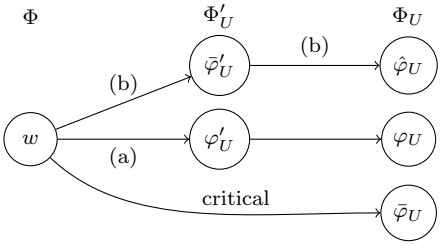
Then, since $\Phi_U, \Phi'_U \in$ iDNF, φ_U can be the only clause in Φ_U that satisfies sentence (iii) and together with $\neg(\varphi_U \supseteq \varphi'_U)$ from (iv) we can conclude $\varphi'_U \supset \varphi_U$. Let x be a variable that occurs in φ'_U, but not in φ_U. We show a contradiction to the above sentences (i) – (vii) by case differentiation:

Case 1: $x \notin \Phi_U$. According to (vii), φ_U has a critical witness $w \in \Phi$. Consider Figure 4.12a.

Case 1(a): $w \models \varphi'_U$. Then φ_U can be extended by x and the resulting formula is still in iDNF and implied by Φ. This is a contradiction to (v).



(a) Witness graph illustration of the witnesses in $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Phi'_U) \subset \mathcal{M}(\Phi_U)$ for the case $x \notin \Phi_U$. In case 1(b), the assumption that w is a critical witness for φ_U is disproved by clause $\bar{\varphi}_U$ which is also implied by w due to the transitivity of the implication relation.



(b) Witness graph illustration of the witnesses in $\mathcal{M}(\Phi) \subseteq \mathcal{M}(\Phi'_U) \subset \mathcal{M}(\Phi_U)$ for the case $x \in \Phi_U$. In cases 2(a) and 2(b), the assumption that w is a critical witness for $\bar{\varphi}_U$ is disproved by clause φ_U or $\hat{\varphi}_U$, respectively, which are also implied by w due to the transitivity of the implication relation.

Figure 4.12: Witness graphs used in the proof of Theorem 34.

Case 1(b): If $w \not\models \varphi'_U$. Then according to (i) there is a $\bar{\varphi}'_U \in \Phi'_U$ such that $w \models \bar{\varphi}'_U$. Since $\Phi'_U \in$ iDNF, φ'_U and $\bar{\varphi}'_U$ share no variables; due to (iii) there must be a $\bar{\varphi}_U \in \Phi_U$ which is implied by $\bar{\varphi}'_U$ and is different from φ_U due to the iDNF property of Φ_U. From the transitivity of the implication relation it follows $w \models \bar{\varphi}_U$ which is a contradiction to the assumption that w is a critical witness for φ_U.

Case 2: $x \in \Phi_U$. Let $\bar{\varphi}_U \in \Phi_U$ such that $x \in \bar{\varphi}_U$, and $w \in \Phi$ be a critical witness for $\bar{\varphi}_U$. Consider Figure 4.12b.

Case 2(a): $w \models \varphi'_U$. As above, transitivity of $\models$ implies $w \models \varphi_U$ which is a contradiction to the assumption that w is a critical witness for $\bar{\varphi}_U$.

Case 2(b): $w \not\models \varphi'_U$. Then according to (i) there is a $\bar{\varphi}'_U \in \Phi'_U$ such that $w \models \bar{\varphi}'_U$. Since $x \in \varphi'_U$ and $\Phi'_U \in$ iDNF, $\bar{\varphi}'_U$ does not contain the variable x; due to (iii) there must be a $\hat{\varphi}_U \in \Phi_U$ which is implied by $\bar{\varphi}'_U$ and does thus not contain the variable x and is hence different from $\bar{\varphi}_U$. Transitivity of $\models$ implies $w \models \hat{\varphi}_U$ which is a contradiction to the assumption that w is a critical witness for $\bar{\varphi}_U$.

This completes the proof for the direction MUB $\Rightarrow$ LUB.

LUB $\Rightarrow$ MUB. Assume Φ_U is no MUB for Φ. We need to show that whenever any of the three conditions in Definition 19 does not hold, then Φ_U is no LUB for Φ.

Case 1: If there is a clause in Φ which is not a witness of clauses in Φ_U, then $\Phi \not\models \Phi_U$ and thus Φ_U is no upper bound for Φ and in particular no least upper bound.

Case 2: Let x be a variable such that a clause $\varphi_U \in \Phi_U$ can be extended by x and the resulting formula is in iDNF and implied by Φ. It is clear that x does not occur in Φ_U, because otherwise it would not be possible to extend the formula without violating the iDNF property. Then the assignment with $x \leftarrow$ false and all other variables *true* is a model

of Φ_U but no model of the extended formula Φ_U^{ext}. Thus $\mathcal{M}(\Phi_U^{ext}) \subset \mathcal{M}(\Phi_U)$ and Φ_U is no LUB for Φ.

Case 3: Let $\varphi_U \in \Phi_U$ be a clause without a critical witness in Φ. Then removing φ_U from Φ_U creates a formula $\bar{\Phi}_U$ with the following properties:

(i) Since Φ_U is an iDNF formula, removing φ_U creates a formula with strictly fewer models than Φ_U.

(ii) Since all witnesses of φ_U are non-critical, they imply other clauses in Φ_U as well. Hence $\bar{\Phi}_U$ is still implied by Φ.

$\bar{\Phi}_U$ is a better upper bound for Φ and Φ_U is no LUB. □

4.5.7 Proof of Theorem 36

Theorem 36. *Algorithm 6 enumerates iDNF least upper bounds of a positive DNF formula with n clauses with $O(n^3)$ delay.*

Proof. The proof of Theorem 36 comprises three parts: (i) Every formula returned by Algorithm 6 is an MUB, (ii) no two formulas returned by the algorithm are equivalent, and (iii) the delay between returning consecutive MUBs is polynomial in the size of the input formula. Properties (i) and (ii) are shown below in Lemmata 48 and 49. Regarding (iii), let us consider the tree corresponding to the execution trace of the algorithm, where each iteration of the for-loop generates a new branch and each recursive call generates a child node; the leaves of the tree represent the MUBs discovered. The algorithm explores this tree in depth-first order. On each branch, the number of recursions and hence the height of the tree is bounded by the number n of clauses of the input formula. The formulas Φ_R, $\{\varphi \setminus \mathrm{vars}(\psi) \mid \varphi \in \Phi_R \wedge \varphi \neq \psi\}$, and $\Phi_U \cup \{\psi\}$ can be constructed in time at most quadratic in n (and linear in the max-arity of Φ). Hence, the time required to traverse the tree from the root to a leaf is bounded by $\mathcal{O}(n^3)$ and constitutes an upper bound for the delay between consecutive MUBs. □

Lemma 48. *Each formula returned by Algorithm 6 is an iDNF minimal upper bound for the input formula Φ.*

Proof. Proof by induction; base case: If PolyMub is called with a (possibly empty) clause Φ, then it returns Φ as its only MUB.

For the induction step, let ψ be a clause and Φ a DNF formula such that $\psi \vee \Phi$ is irreducible, and let $\Phi^{r(\psi)}$ be the formula obtained from Φ by removing all occurrences of variables of ψ, i.e., $\Phi^{r(\psi)} = \{\varphi \setminus \mathrm{vars}(\psi) \mid \varphi \in \Phi\}$; we prove the following property: If $\Phi_U^{r(\psi)}$ is an MUB for $\Phi^{r(\psi)}$, then $\varphi \vee \Phi_U^{r(\psi)}$ is an MUB for $\psi \vee \Phi$. This composition property is exactly the induction step showing that PolyMub constructs MUBs.

Let ψ, Φ, $\Phi^{r(\psi)}$ be as above, and let $\Phi_U^{r(\psi)}$ be an MUB for $\Phi^{r(\psi)}$. Then $\Phi_U^{r(\psi)}$ is an iDNF formula and satisfies the *upper bound, maximality, and criticality* conditions from

Definition 19 with respect to $\Phi^{r(\psi)}$. It remains to be shown that $\psi \vee \Phi_U^{r(\psi)}$ is an iDNF formula and satisfies those three conditions with respect to $\psi \vee \Phi$. First, for the iDNF property: Since $\Phi^{r(\psi)}$ does not contain variables from ψ, and since $\Phi_U^{r(\psi)}$ is an MUB for $\Phi^{r(\psi)}$, it follows that ψ and $\Phi_U^{r(\psi)}$ do not share variables, and thus $\psi \vee \Phi_U^{r(\psi)}$ is an iDNF formula.

Upper bound. We need to show that every clause in $\psi \vee \Phi$ implies a clause in $U = \psi \vee \Phi_U^{r(\psi)}$ (cf. Lemma 1). For ψ, this is evident. By transitivity of $\models$, and the fact that $\Phi^{r(\psi)} \models \Phi_U^{r(\psi)}$ by induction hypothesis, it suffices to show $\Phi \models \Phi^{r(\psi)}$. Let φ be a clause in Φ; then $\varphi \setminus \mathrm{vars}(\psi)$ is a clause in $\Phi^{r(\psi)}$ and is implied by φ (cf. Proposition 26).

Maximality. We show that every clause in $U = \psi \vee \Phi_U^{r(\psi)}$ is maximal. By induction hypothesis, $\Phi_U^{r(\psi)}$ is maximal with respect to $\Phi^{r(\psi)}$ and variables $\mathrm{vars}(\Phi^{r(\psi)})$; moreover, $\Phi_U^{r(\psi)}$ is also maximal with respect to Φ, since it cannot be extended by any of the remaining variables $\mathrm{vars}(\psi)$, since ψ is a clause in U and this extension would violate the iDNF property of U. ψ cannot be extended by a variable from $\mathrm{vars}(\Phi)$, as it would violate the upper bound property: Since $\Phi_U^{r(\psi)}$ does not contain any variable from $\mathrm{vars}(\psi)$, ψ only implies clause ψ in U. Any extension to ψ would invalidate this implication.

Criticality. It needs to be shown that every clause in $U = \psi \vee \Phi_U^{r(\psi)}$ has a critical witness in $\psi \vee \Phi$. Since $\psi \vee \Phi$ is irreducible, ψ is the only witness for $\psi \in U$. Now let φ be a clause in $\Phi_U^{r(\psi)}$; φ is not implied by ψ since they do not share any variables. We still need to show that φ has a critical witness in Φ. By induction hypothesis, φ has a critical witness $w \in \Phi^{r(\psi)}$; let $w^{e(\psi)} \in \Phi$ be the clause w extended by variables from $\mathrm{vars}(\psi)$. $w^{e(\psi)}$ is a critical witness for φ, since: (i) $w^{e(\psi)}$ cannot imply a different clause $\varphi' \in \Phi_U^{r(\psi)}$, since φ' does not contain variables from $\mathrm{vars}(\psi)$ and w does not imply φ'; (ii) $w^{e(\psi)}$ cannot imply ψ, since this would require $\psi \subset w^{e(\psi)}$ and hence $\psi \vee \Phi$ would not be irreducible which is a contradiction to our initial assumptions. □

Lemma 49. *Let Φ and Ψ be two formulas returned by Algorithm 6. Then $\Phi \not\equiv \Psi$.*

Proof. Φ and Ψ are irreducible since they are iDNF formulas by Lemma 48. By construction, Φ and Ψ contain two distinct clauses φ and ψ that share a variable x. Since Φ and Ψ are iDNF formulas, we thus have $\varphi \in \Phi$, $\varphi \notin \Psi$, $\psi \in \Psi$, and $\psi \notin \Phi$; it then follows from Corollary 2 that $\Phi \not\equiv \Psi$. □

4.5.8 Proof of Proposition 38

Proposition 38. *Let Φ be a positive formula, x be a variable in Φ, and Φ_L and Φ_U be formulas obtained from Φ by replacing one occurrence of x by $\bot$ and respectively $\top$. Then Φ_L and Φ_U are lower and respectively upper bounds of Φ.*

Proof. Let Φ be a positive formula, x be a variable in Φ, and Φ_L and Φ_U be formulas obtained from Φ by replacing one occurrence of x by $\bot$ and respectively $\top$. We show that

$\Phi_L \models \Phi \models \Phi_U$.

Assume that variable x occurs n times in Φ and that its different occurrences are labeled with an extra superscript, i.e., variable x appears as $x^1, \ldots, x^n$ in Φ. Without loss of generality, assume that occurrence x^1 is replaced by $\bot$ (or $\top$, respectively). Consider the DNF formulas equivalent to Φ_L, Φ, Φ_U. Since Φ is positive, all variables in those DNF formulas occur positively. Setting x^1 to $\bot$ in Φ_L corresponds to making the clauses in which x^1 appears in Φ_L's DNF representation unsatisfiable, or, equivalently, removing them from the formula. Hence Φ_L contains a subset of the clauses of Φ. Conversely, setting x^1 to $\top$ in Φ_U corresponds to removing x^1 from all clauses that it appears in. By Proposition 26 it follows that $\Phi_L \models \Phi \models \Phi_U$. □

4.5.9 Proof of Proposition 42

Proposition 42. *Let ϵ be a fixed error threshold, and Φ a formula with probability bounds $[L, U]$, i.e., $P_\Phi \in [L, U]$.*

- *If $U - \epsilon \leq L + \epsilon$, then any value in the interval $[U - \epsilon, L + \epsilon]$ is an absolute ϵ-approximation of P_Φ.*
- *If $(1 - \epsilon) \cdot U \leq (1 + \epsilon) \cdot L$, then any value in the interval $[(1 - \epsilon) \cdot U, (1 + \epsilon) \cdot L]$ is a relative ϵ-approximation of P_Φ.*

Proof. Let p be the probability of Φ. According to Proposition 41, $L \leq p \leq U$. For error ϵ it holds that $1 - \epsilon \geq 0$.

For absolute ϵ-approximation, choose some $\hat{p} \in [U - \epsilon, L + \epsilon]$. Then we have $p - \epsilon \leq U - \epsilon \leq \hat{p} \leq L + \epsilon \leq p + \epsilon$. For relative ϵ-approximation choose some $\hat{p} \in [(1-\epsilon) \cdot U, (1+\epsilon) \cdot L)]$. Then we have $(1 - \epsilon) \cdot p \leq (1 - \epsilon) \cdot U \leq \hat{p} \leq (1 + \epsilon) \cdot L \leq (1 + \epsilon) \cdot p$. □

4.5.10 Proof of Lemma 44

Lemma 44. *For a d-tree T, $L(T)$ is the pair of bounds $[L, U]$ that maximises $U - L$ for absolute approximation and $(1-\epsilon) \cdot U - (1+\epsilon) \cdot L$ for relative approximation, over the entire bounds space of T.*

Proof. Consider the point interval of each open leaf be $[x, x]$, where x is a distinct variable. The upper and lower bounds of T can be then expressed as functions f_U and f_L, respectively, of such variables. We show that for each such variable x, $\frac{\partial(f_U - f_L)}{\partial x} \leq 0$ and hence $f_U - f_L$ is maximised when x is minimised. That is, when $x = L$, where L is the lower bound of that open leaf.

We denote by f_U^n and f_L^n the lower and upper bound functions in variable x for a node at depth n. These functions are linear: $f_U^n = a_U^n \cdot x + b_U^n$ and $f_L^n = a_L^n \cdot x + b_L^n$.

Base case: Open leaf with variable x, depth n, and $f_U^n = f_L^n = x$. Then, $\frac{\partial(f_U^n - f_L^n)}{\partial x} = 1 - 1 = 0$.

Assume now the property holds at a node c at level $j+1$, and c is an ancestor of the open leaf with x or that open leaf. We show that the property also holds at the parent of c (and depth j).

Case 1: The parent of c is a $\sqcup$ node: $\sqcup(c_1, \ldots, c_k)$, where c is one of $c_1, \ldots, c_k$. Then,

$$f_U^j = f_U^{j+1} + \alpha_U = a_U^{j+1} \cdot x + b_U^{j+1} + \alpha_U$$
$$f_L^j = f_L^{j+1} + \alpha_L = a_L^{j+1} \cdot x + b_L^{j+1} + \alpha_L$$

where α_U and α_L represent the sum of the upper bounds, and lower bounds respectively, of all the siblings of c. We then immediately have that $\frac{\partial(f_U^j - f_L^j)}{\partial x} = a_U^{j+1} - a_L^{j+1} \leq 0$.

Case 2: The parent of c is a $\odot$ node: $\odot(c_1, \ldots, c_k)$, where c is one of $c_1, \ldots, c_k$. Recall that we only consider restricted $\odot$ nodes, where at most one child is not a clause and can have different values for lower and upper bounds. If this child is c, let q be the product of the (exact) probabilities of all other children. Then, $a_U^j = a_U^{j+1} \cdot q$ and $a_L^j = a_L^{j+1} \cdot q$ and thus the inequality $a_U^j - a_L^j \leq 0$ is preserved.

Case 3: The parent of c is a $\oplus$ node: $\oplus(c_1, \ldots, c_k)$, where c is one of $c_1, \ldots, c_k$. Let

$$\alpha_L = \prod_{i=1, c_i \neq c}^{k} (1 - L(c_i)) \qquad \alpha_U = \prod_{i=1, c_i \neq c}^{k} (1 - U(c_i))$$

where $L(c_i)$ and $U(c_i)$ represent the formulas for the lower and upper bounds, respectively, of node c_i. Given that $L(c_i) \leq U(c_i)$ for each node c_i, it holds that $\alpha_L \leq \alpha_U$. Then,

$$\begin{aligned} f_U^j &= 1 - \alpha_U \cdot (1 - f_U^{j+1}) \\ &= \alpha_U \cdot a_U^{j+1} \cdot x + 1 - \alpha_U + \alpha_U \cdot b_U^{j+1} \\ f_L^j &= 1 - \alpha_L \cdot (1 - f_L^{j+1}) \\ &= \alpha_L \cdot a_L^{j+1} \cdot x + 1 - \alpha_L + \alpha_L \cdot b_L^{j+1} \\ \frac{\partial(f_U^j - f_L^j)}{\partial x} &= \alpha_U \cdot a_U^{j+1} - \alpha_L \cdot a_L^{j+1} \leq 0. \end{aligned}$$

The latter inequality holds since $\alpha_U \leq \alpha_L$ (as discussed above) and $a_U^{j+1} \leq a_L^{j+1}$ (by hypothesis).

For relative approximation, we need to find x that maximises $(1-\epsilon) \cdot U - (1+\epsilon) \cdot L$. This holds by a straightforward extension of the previous proof: The coefficient of x is shown to be greater in L than in U for $U - L$. Since $1 - \epsilon \leq 1 + \epsilon$, this property is preserved. $\square$

Chapter 5

Queries with aggregation

The utility of database query languages that support aggregation has been argued for at length. In particular, aggregates are crucial for OLAP and decision support systems. All 22 TPC-H queries involve aggregation. This chapter considers the evaluation problem for queries with aggregates on probabilistic databases. The goal is to delevelop a coherent framework for computing the exact probability distributions that arise as answers to positive relational algebra queries with aggregation operators on probabilistic data. We follow the customary approach that lets users pose standard, deterministic queries that are unaware of the probabilistic nature of the data. The semantics of these queries, however, is probabilistic: Each possible answer is annotated with a probability measure that indicates its likelihood given the probabilities of the underlying data. While answers to queries without aggregates are Boolean random variables — each tuple is either an answer or it is not — the answers to queries with aggregates can be random variables over larger domains. For example, for a simple SUM aggregation over a tuple-independent probabilistic table whose tuples are positive integers, the sum over each subset of the tuples is a possible answer; the query result is thus a probability distribution over each possible subset-sum.

The query evaluation technique presented in this chapter is grounded in a novel representation system for probabilistic data called *pvc-tables* (probabilistic value-conditioned tables). It is based on the algebraic structures of semiring and semimodule to support a mixed representation of aggregated values and tuple annotations for different classes of annotations and aggregations [5]. Like pc-tables, pvc-tables can represent any finite probability distribution over relational databases. In addition, the results of queries with aggregates can be represented as pvc-tables of polynomial size. This contrasts with the pc-tables [79], which can require an exponential-size overhead [56].

Our approach considers the problem of exact probability computation for positive relational algebra queries with aggregates on pvc-tables. The core of our technique is a procedure that compiles arbitrary semimodule and semiring expressions over random variables into decomposition trees, for which the computation of the probability distribution can be done in polynomial time in the size of the tree and of the distributions at its nodes.

The decomposition trees proposed in this chapter are a generalisation of those defined in Chapter 4: While the latter operate on Boolean interpretations of the represented formula, the d-trees used for queries with aggregates represent probability distributions over arbitrary countable domains. To this end, the annotations of tuples are lifted from Boolean expressions to semiring expressions.

S

sid	shop	Φ
1	M&S	$\mathbf{x_1}$
2	M&S	$\mathbf{x_2}$
3	M&S	$\mathbf{x_3}$
4	Gap	$\mathbf{x_4}$
5	Gap	$\mathbf{x_5}$

(a)

PS

sid	pid	price	Φ
1	1	10	$\mathbf{y_{11}}$
1	2	50	$\mathbf{y_{12}}$
2	1	11	$\mathbf{y_{21}}$
2	2	60	$\mathbf{y_{22}}$
3	3	15	$\mathbf{y_{33}}$
3	4	40	$\mathbf{y_{34}}$
4	1	15	$\mathbf{y_{41}}$
4	3	60	$\mathbf{y_{43}}$
5	1	10	$\mathbf{y_{51}}$

(b)

P_1

pid	weight	Φ
1	4	$\mathbf{z_1}$
2	8	$\mathbf{z_2}$
3	7	$\mathbf{z_3}$
4	6	$\mathbf{z_4}$

(c)

P_2

pid	weight	Φ
1	5	$\mathbf{z_5}$

(d)

$Q_1 = \pi_{\text{shop, price}}[S \bowtie PS \bowtie (P_1 \cup P_2)]$

shop	price	Φ
M&S	10	$\mathbf{x_1y_{11}(z_1 + z_5)}$
M&S	50	$\mathbf{x_1y_{12}z_2}$
M&S	11	$\mathbf{x_2y_{21}(z_1 + z_5)}$
M&S	60	$\mathbf{x_2y_{22}z_2}$
M&S	15	$\mathbf{x_3y_{33}z_3}$
M&S	40	$\mathbf{x_3y_{34}z_4}$
Gap	15	$\mathbf{x_4y_{41}(z_1 + z_5)}$
Gap	60	$\mathbf{x_4y_{43}z_3}$
Gap	10	$\mathbf{x_5y_{51}(z_1 + z_5)}$

(e)

$Q_2 = \pi_{\text{shop}}\sigma_{P\leq 50}\varpi_{\text{shop;P}\leftarrow \text{MAX(price)}}[Q_1]$

Shop	Φ
M&S	$[\mathbf{x_1y_{11}(z_1 + z_5)} \otimes \mathbf{10} +_{\max} \mathbf{x_1y_{12}z_2} \otimes \mathbf{50} +_{\max}$ $\mathbf{x_2y_{21}(z_1 + z_5)} \otimes \mathbf{11} +_{\max} \mathbf{x_2y_{22}z_2} \otimes \mathbf{60} +_{\max}$ $\mathbf{x_3y_{33}z_3} \otimes \mathbf{60} +_{\max} \mathbf{x_3y_{34}z_4} \otimes \mathbf{15} \leq \mathbf{50}] \cdot \mathbf{\Psi_1}$
Gap	$[\mathbf{x_4y_{41}(z_1 + z_5)} \otimes \mathbf{15} +_{\max} \mathbf{x_4y_{43}z_3} \otimes \mathbf{60} +_{\max}$ $\mathbf{x_5y_{51}(z_1 + z_5)} \otimes \mathbf{10} \leq \mathbf{50}] \cdot \mathbf{\Psi_2}$

(f)

Where: $\Psi_1 = [x_1y_{11}(z_1 + z_5) + x_1y_{12}z_2 + x_2y_{21}(z_1 + z_5) + x_2y_{22}z_2 + x_3y_{33}z_3 + x_3y_{34}z_4 \neq 0_K]$
$\Psi_2 = [x_4y_{41}(z_1 + z_5) + x_4y_{43}z_3 + x_5y_{51}(z_1 + z_5) \neq 0_K]$

Figure 5.1: A database containing relations (a) Supplier S, (c) Products P_1, P_2, and (b) the many-to-many relation ProductSupplied PS, and the results of the positive query Q_1 and of the aggregate query Q_2 (d,e).

Example 37. Figure 5.1 shows six pvc-tables, amongst them the suppliers table S, the products tables P_1 and P_2, and the table PS pairing suppliers and products. They all have an annotation column Φ to hold expressions in a *semiring* K generated by a set of independent random variables, with operations sum ($+$) and product ($\cdot$), and neutral elements 0_K and 1_K. Each valuation of the random variables into a semiring (e.g. integers or Booleans) canonically maps semiring expressions into that semiring by interpreting $+$ and $\cdot$ as the corresponding operations of that semiring. Each such valuation defines a possible world of the database.

Figure 5.1e shows the result of query Q_1 that asks for prices of products available in

shops. The annotations of the result tuples are constructed as follows: The annotation of a join of two tuples is the product of their annotations, and the annotation obtained from projection or union is the sum of the annotations of the participating tuples [41]. For instance, the tuple $\langle \text{M\&S}, 10\rangle$ has the annotation $x_1 y_{11}(z_1 + z_5)$, whose probability distribution can be computed as a function of probability distributions of the random variables x_1, y_{11}, z_1, and z_5.

Consider the query Q_2 from Figure 5.1f that asks for shops in which the maximal price for the products in P_1 or P_2 is less than 50. Aggregation is expressed using the ϖ operator, which in this query groups by the column shop and applies the aggregation MAX on price within each group.

The annotations of result tuples are built using *semimodule* expressions of the form $\Psi \otimes v$, where Ψ is a semiring expression and v is a data value. Such expressions can be "summed up" with respect to aggregation operations: For MIN, the sum $\alpha +_{\min} \beta$ is $\min(\alpha, \beta)$; for MAX, $\alpha +_{\max} \beta = \max(\alpha, \beta)$; for SUM, $\alpha +_{\text{sum}} \beta = \alpha + \beta$. The sums correspond to operations in *commutative monoids*. The annotation Φ of M&S in Q_2's result is constructed as follows. This tuple represents a group of six tuples in the result of Q_1, all with the M&S shop value. The annotation Φ then expresses the conditions (1) that the sum of the price values of these six tuples in the MAX monoid is less than 50, and (2) that the group is not empty (as expressed by Ψ_1). Depending on the valuation of the variables in Φ, these conditions can be true ($\top$) or false ($\bot$), or, more generally, the additive or multiplicative neutral element of the semiring.

For instance, a valuation ν_1 that maps $x_1, x_2,\ y_{11}, y_{21}, z_1,\ z_2,\ z_5$ to $\top$ and all other variables to $\bot$ satisfies Φ, since

$$\begin{aligned}
\nu_1(\Phi) &\equiv [\top \otimes 10\ +_{\max}\ \bot \otimes 50\ +_{\max}\ \top \otimes 11\ +_{\max}\ \bot \otimes 60\ +_{\max}\ \bot \otimes 60\ +_{\max}\ \bot \otimes 15 \leq 50] \cdot \top \\
&\equiv [10\ +_{\max}\ 11 \leq 50] \\
&\equiv [_{\max}(10, 11) \leq 50] \\
&\equiv \top. \qquad \square
\end{aligned}$$

If the variables in such expressions are random variables, then the expressions themselves can be interpreted as random variables. Moreover, the probability distributions of the obtained expressions reflect the probabilities of query answers taking particular values in a randomly drawn world of the database. For example, the expression Φ in the preceding example is a random variable over two values $\{\top, \bot\}$ that it may take under different valuations of the variable symbols. Our technique allows to efficiently compute probabilities defined by such expressions by structural decomposition. For example, an expression $\alpha = ab \otimes 10 + xy \otimes 20$ can be decomposed in independent sub-expressions $ab \otimes 10$ and $xy \otimes 20$ that do not share variable symbols and hence constitute independent random variables.

$$
\begin{array}{rl}
\Phi ::= & x \mid \Phi + \Phi \mid \Phi \cdot \Phi \mid [\alpha\, \theta\, \alpha] \mid [\Phi\, \theta\, \Phi] \mid s \\
\alpha ::= & \Phi \otimes m \quad \{+_{\mathrm{op}} \Phi \otimes m\} \mid m \\
\mathrm{op} ::= & \min \mid \max \mid \mathrm{count} \mid \mathrm{sum} \mid \mathrm{prod} \\
\theta ::= & = \mid \neq \mid \leq \mid \geq \mid < \mid > \\
x ::= & \text{A variable symbol } x \in \mathbf{X} \\
m ::= & \text{A value from an aggregation monoid } M \\
s ::= & \text{A value from the semiring } S
\end{array}
$$

Figure 5.2: Grammar for semiring expressions $\Phi \in K$ and semimodule expressions $\alpha \in (K \cup S) \otimes M$.

Organisation of this chapter

The contributions and the structure of this chapter are as follows:

- We present an evaluation framework for queries with aggregates (MIN, MAX, SUM, PROD, COUNT) on pvc-tables, a representation system for probabilistic data based on semirings and semimodules (Section 5.1).
- We devise a technique for computing the exact probability distribution of query results based on a generic compilation procedure of *arbitrary* semimodule and semiring expressions into decomposition trees, for which the computation of the probability distribution can be done in polynomial time in the size of the decomposition tree and of the distribution (Sections 5.2 and 5.3).
- We give a syntactic characterisation of a class of aggregate queries that are tractable on tuple-independent databases (Section 5.4). Our query tractability result follows from the observation that the semiring and semimodule expressions in the result of our tractable queries admit polynomial size decomposition trees and polynomial size probability distributions at their nodes.
- A prototype of our technique is incorporated into the probabilistic database engine SPROUT. Extensive performance experiments using our own synthetic datasets and TPC-H data are discussed (Section 5.5).

5.1 pvc-tables: A representation system for probabilistic data

In this section we introduce a succinct and complete representation system for probabilistic data, which we call *probabilistic value-conditioned tables* or pvc-tables for short. This system is based on work in databases with provenance information [5, 41, 14]. The reason for using pvc-tables, as opposed to main-stream representation systems such as pc-tables (and

special cases such as tuple-independent or BID tables) [79], is that pvc-tables fit naturally with aggregate queries: answers to aggregate queries on pvc-tables or even on pc-tables are representable as pvc-tables of size polynomial in the size of the input tables, while they may require an exponential overhead when modelled as pc-tables [56].

The first sub-section introduces the algebraic foundations for modelling the results of queries with aggregates. The second sub-section formalises the pvc-tables representation system.

5.1.1 Algebraic structures for representing aggregates

Our representation system for probabilistic data is based on the notions of monoid, semiring, and semimodule.

Definition 25. *A* monoid *is a set M with a binary operation $+ : M \times M \to M$ and a neutral element $0 \in M$ that satisfy the following axioms for all $m_1, m_2, m_3 \in M$:*

Commutativity:	$(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)$
Neutral element:	$0 + m_1 = m_1 + 0 = m_1$

A monoid is commutative *if $m_1 + m_2 = m_2 + m_1$.*

Monoids naturally describe many aggregation operations. Aggregation over a column of a relation fixes a domain of values, usually $\mathbb{R}$ or $\mathbb{N}$, and a binary operation. For instance, the MIN of a column with values $v_1, \cdots, v_n \in \mathbb{R}$ is

$$\min(v_1, \cdots, v_n) = \min(v_1, \min(v_2, \cdots \min(v_{n-1}, v_n) \cdots)).$$

The binary operation is *commutative* and *associative*, i.e., the value of the aggregation is invariant under the order in which it is computed. Furthermore, each aggregation operation has a *neutral element*, i.e., a value that does not contribute to the aggregation. For example, $0 \in \mathbb{R}$ is the neutral element for SUM, and ∞ is the neutral element for MIN. These aggregations correspond to commutative monoids, in particular: SUM $= (\mathbb{N}, +, 0)$, MIN $= (\mathbb{N}^{\pm\infty}, \min, +\infty)$, MAX $= (\mathbb{N}^{\pm\infty}, \max, -\infty)$. COUNT is a special case of SUM. More complicated aggregations (e.g., AVG) can conceptually be composed from simpler ones (e.g., SUM and COUNT), but their treatment is out of the scope of this work.

Definition 26. *A* commutative semiring *is a set S together with operations $+, \cdot : S \times S \to S$ and neutral elements $0, 1 \in S$ such that $(S, +, 0)$ and $(S, \cdot, 1)$ are commutative monoids and*

the following holds for all $s_1, s_2, s_3 \in S$*:*

Left distributivity:	$s_1 \cdot (s_2 + s_3) = (s_1 \cdot s_2) + (s_1 \cdot s_3)$
Right distributivity:	$(s_1 + s_2) \cdot s_3 = (s_1 \cdot s_3) + (s_2 \cdot s_3)$
Neutral element:	$0 \cdot s_1 = s_1 \cdot 0 = 0.$

Commutative semirings are the canonical algebraic structure for tuple annotations [41]. Annotations from the Boolean semiring yield set semantics, annotations from $\mathbb{N}$ correspond to bag semantics, and annotations from the security semiring can be used to constrain access to query results depending on access rights to database tuples that contributed to the result [5]. The most general semirings are those generated over a set of variables. Intuitively, the carrier of such semirings are syntactic expressions built from the variables and the multiplication and sum symbols, where elements are identified via the semiring laws. For example, given a set $\mathbf{X} = \{x_1, x_2, x_3\}$ of variables, the elements of the semiring PosBool($\mathbf{X}$) are positive Boolean expressions, e.g., $x_1 + x_2$ or $x_1(x_2 + x_3)$. By the distributivity law in semirings, the expressions $x_1(x_2 + x_3)$ and $x_1x_2 + x_1x_3$ are equal. A more general freely generated semiring is the ring of polynomials (also called free commutative algebra) over a set $\mathbf{X}$ of variables. Elements of generated semirings are called *semiring expressions* which we denote by K throughout this chapter.

Definition 27. *Let* $(S, +_S, 0_S, \cdot_S, 1_S)$ *be a commutative semiring. An* S-semimodule M *consists of a commutative monoid* $(M, +_M, 0_M)$ *and a binary operation* $\otimes : S \times M \to M$ *such that for all* $s_1, s_2 \in S$ *and* $m_1, m_2 \in M$ *we have*

Left distributivity:	$s_1 \otimes (m_1 +_M m_2) = s_1 \otimes m_1 +_M s_1 \otimes m_2$
Right distributivity:	$(s_1 +_S s_2) \otimes m_1 = s_1 \otimes m_1 +_M s_2 \otimes m_1$
Commutativity:	$(s_1 \cdot_S s_2) \otimes m_1 = s_1 \otimes (s_2 \otimes m_1)$
Neutral element:	$s_1 \otimes 0_M = 0_S \otimes m_1 = 0_M$
Neutral element:	$1_S \otimes m_1 = m_1.$

We write $S \otimes M$ to denote a S-semimodule M, and write $\cdot$ for $\cdot_S$ and $+$ for $+_S, +_M$ whenever the meaning is unambiguous. Semimodules combine monoids with semirings to represent aggregation values *conditioned* on the value of a semiring expression. Analogous to the case of semiring expressions that correspond to freely generated semirings, we denote by *semimodule expressions* the elements of the K-semimodule generated by a given monoid. The semimodules we use frequently are $\mathbb{N} \otimes \mathbb{N}$ and $\mathbb{B} \otimes \mathbb{N}$ for MIN, MAX, SUM, and PROD monoids. A semimodule $\mathbb{B} \otimes \mathbb{N}$ over SUM would not have the expected semantics of additing and multiplying integers, as illustrated by the attempt to take the sum of two tuples $\langle 1 \rangle$ that exist in the database with certainty, i.e. have annotation $\top$. We would expect the

result to be 1+1=2, yet the application of the semi-module axioms of Definition 27 yields:

$$\top \otimes 1 +_{\text{SUM}} \top \otimes 1 = (\top \vee \top) \otimes 1 = \top \otimes 1 = 1$$

This example reflects the well-known incompatibility of SUM aggregation with set semantics.

5.1.2 pvc-tables

The pvc-tables representation system generalises pc-tables by allowing for valuations of the annotations of formulas beyond the Boolean true/false: Intuitively, a tuple with annotation Φ in a pc-table is in the database in worlds ν for which $\nu(\Phi) = \top$, and is not in the database in worlds with $\nu(\Phi) = \bot$; this corresponds to set semantics. In pvc-tables, annotations can, for example, be evaluated to positive integers, thus yielding bag semantics: a tuple with $\nu(\Phi) = 0$ is in the database with multiplicity 0 (i.e., it is not in the database when defaulting to set semantics), and a tuple with $\nu(\Phi) = n$ is in the database with multiplicity n.

The starting point of the formalisation of the above intuition is the generalisation of the *induced probability space* over Boolean random variables (cf. Section 2) to random variables over countable domains. Let S be a countable set and $\mathbf{X}$ be a finite set of S-valued independent random variables. As before, we denote by P_x the discrete probability distribution of a variable $x \in \mathbf{X}$, and by $P_x[s]$ the probability that x takes value $s \in S$; we often specify P_x by the set of pairs of unique values with their non-zero probabilities, $\{(s, P_x[s]) \mid s \in S \text{ and } P_x[s] > 0\}$. The *size* of a probability distribution is the size of its set representation.

Definition 28. *Let $\Omega = \{\nu : \mathbf{X} \to S\}$ be the set of mappings from $\mathbf{X}$ into S. A probability mass function*

$$\Pr(\nu) = \prod_{x \in \mathbf{X}} P_x[\nu(x)]$$

for every sample $\nu \in \Omega$, and a probability measure

$$\Pr(E) = \sum_{\nu \in E} \Pr(\nu) \qquad \text{for all } E \subseteq \Omega$$

define a probability space $(\Omega, 2^\Omega, \Pr)$ that we call the probability space induced by $\mathbf{X}$.

Aggregation on pvc-tables is handled using a mixed representation of tuple annotations and aggregated values using the algebraic structures of semirings and semimodules.

Definition 29. *A* pvc-table *T over a probability space Ω induced by a set $\mathbf{X}$ of variables is a relation with an annotation column Φ holding semiring expressions over $\mathbf{X}$, and where the tuple values can be constants or semimodule expressions over $\mathbf{X}$. A pvc-database $D = \{T_1, \ldots, T_n\}$ is a set of pvc-tables over the same probability space Ω.*

$S_{\mathbb{B}}$

sid	shop	Φ
1	M&S	$\bot$
2	M&S	$\top$
3	M&S	$\bot$
4	Gap	$\bot$
5	Gap	$\top$

(a)

$S_{\mathbb{N},1}$

sid	shop	Φ
1	M&S	**2**
2	M&S	**1**
3	M&S	**1**
4	Gap	**7**
5	Gap	**2**

$S_{\mathbb{N},2}$

sid	shop	Φ
1	M&S	**0**
2	M&S	**1**
3	M&S	**3**
4	Gap	**7**
5	Gap	**2**

(b)

Figure 5.3: Three possible worlds of pvc-table S in Figure 5.1 under two different semirings: $S_{\mathbb{B}}$ is annotated with expressions from the Boolean semiring, and $S_{\mathbb{N},1}$ and $S_{\mathbb{N},2}$ with integers.

The semantics of D is given by its possible worlds:

$$\Big\{\big\{\{\nu(t) \mid t \in T_1\}, \ldots, \{\nu(t) \mid t \in T_n\}\big\} \;\Big|\; \nu \in \Omega\Big\}.$$

where the mapping ν is applied to all expressions in tuples t and is identity for constants. Each mapping $\nu \in \Omega$ defines a possible world.

Columns that hold semimodule expressions are called *aggregation columns*. Semimodule expressions over different monoids can co-exist in a pvc-table. The annotation column Φ hosts semiring expressions that are generated by the variable set $\mathbf{X}$, and conditional expressions representing comparisons of expressions and constants. The other columns of a pvc-table host constants and semimodule expressions. A grammar for such expressions is given in Figure 5.2. All these types of expressions can occur in pvc-tables representing the result of aggregate queries; we show how to compute these expressions for positive relational algebra queries with aggregation operators in Section 5.2.

Example 38. Figure 5.1 shows six pvc-tables. The pvc-table S has annotations from a semiring over the set $\mathbf{X}$ of variables $x_1, \ldots, x_5$. Figure 5.3 shows a few possible worlds for this pvc-table for different semirings. First, consider valuations of the variables $\mathbf{X}$ into the Boolean semiring $\mathbb{B}$. Possible world $S_{\mathbb{B}}$ in Figure 5.3a has probability $P_{x_1}[\bot] \cdot P_{x_2}[\top] \cdot P_{x_3}[\bot] \cdot P_{x_4}[\bot] \cdot P_{x_5}[\top]$. Since $\mathbb{B}$ has only the two elements $\bot, \top$, the number of possible worlds is 2^5. Secondly, consider valuations of $\mathbf{X}$ into $\mathbb{N}$. Figure 5.3b shows two possible worlds, with respective probabilities $P_{x_1}[2] \cdot P_{x_2}[1] \cdot P_{x_3}[1] \cdot P_{x_4}[7] \cdot P_{x_5}[2]$ and $P_{x_1}[0] \cdot P_{x_2}[1] \cdot P_{x_3}[3] \cdot P_{x_4}[7] \cdot P_{x_5}[2]$. □

The pvc-tables PS, P_1, P_2, and Q_1 in Figure 5.1 contain no semimodule expressions and are in fact standard pc-tables as introduced in Chapter 2.3. In pvc-tables, however, semimodule expressions can appear as tuple values.

Example 39. Consider an aggregation AGG on the weight column of relation P_1 in Figure 5.1. The semimodule expression representing the aggregated value is $\alpha = z_1 \otimes 4 \;+_{\text{AGG}}\; z_2 \otimes 8 \;+_{\text{AGG}}\; z_3 \otimes 7 \;+_{\text{AGG}}\; z_4 \otimes 6$, where the $+_{\text{AGG}}$ operator depends on the particular aggregation monoid AGG. □

We next discuss the semantics of expressions under valuations of variables; this will be extended to a probabilistic interpretation in Section 5.3.

Semiring, monoid, and semimodule homomorphism. Let K be the semiring generated by $\mathbf{X}$; the variables in $\mathbf{X}$ are themselves elements of K, i.e., $\mathbf{X} \subseteq K$. Given another semiring S, a mapping $\nu : \mathbf{X} \to S$ of the variables can be uniquely extended to a semiring homomorphism $\nu : K \to S$ that "evaluates" semiring expressions from K to elements in S. A similar construction holds for semimodule: For semimodule expressions W and a monoid M, a mapping $\nu : \mathbf{X} \to S$ is extended to a monoid homomorphism $\nu : W \to M$.

Example 40. Consider the semimodule expression

$$\alpha = xy \otimes 5 \ +_{\min} \ (x + z) \otimes 10$$

over the semiring generated by $\mathbf{X} = \{x, y, z\}$ and the monoid $(\mathbb{N}, +_{\min}, \infty)$. The map $\nu : \mathbf{X} \to \mathbb{N}$ evaluates variable symbols to the semiring of positive integers and induces a monoid homomorphism that maps α to positive integers. For instance, the mapping $\nu : x \mapsto 2, y \mapsto 3, z \mapsto 0$ yields

$$\begin{aligned}
\nu(\alpha) &= \nu(x)\nu(y) \otimes 5 \ +_{\min} \ (\nu(x) + \nu(z)) \otimes 10 \\
&= 6 \otimes 5 \ +_{\min} \ 2 \otimes 10 \\
&= 1 \otimes 5 \ +_{\min} \ \ldots \ +_{\min} \ 1 \otimes 5 +_{\min} 1 \otimes 10 \ +_{\min} \ \ldots \ +_{\min} \ 1 \otimes 10 \\
&= 5 \ +_{\min} \ 10 \\
&= 5.
\end{aligned}$$

Similarly, the value of α in Example 39 depends on the particular target semiring S and the valuation of the variables. For instance, $\alpha \mapsto 24$ for SUM aggregation and semiring $\mathbb{N}$ with $z_1, z_2 \mapsto 2$ and $z_3, z_4 \mapsto 0$. Also, $\alpha \mapsto 6$ for MIN aggregation and the Boolean semiring with $z_1 \mapsto \bot$ and $z_2, z_3, z_4 \mapsto \top$. If all variables are mapped to 0_S, the query answer is 0_M, i.e., 0 in case of SUM and $+\infty$ for MIN. □

Conditional expressions have the form $[\alpha\theta\beta]$, where α and β are semimodule expressions for (possibly different) aggregation monoids and semirings generated by $\mathbf{X}$, and θ is a binary relation. Such conditional expressions can appear as tuple annotations in pvc-tables. Given a valuation $\nu : \mathbf{X} \to S$ and (semiring or semimodule) expressions Φ, Ψ, the semantics of $[\Phi\theta\Psi]$ is defined by extending ν via

$$\nu([\Phi\theta\Psi]) = \begin{cases} 1_S, & \text{if } \nu(\Phi) \ \theta \ \nu(\Psi) \\ 0_S, & \text{otherwise} \end{cases} \tag{5.1}$$

Comparisons operators $\leq, \geq$ for θ only make sense for ordered carriers. In the light

Database Semantics		S	Probability Distributions
Deterministic	Set	$\mathbb{B}$	$P_x[\top] = 1$ or $P_x[\bot] = 1$
Deterministic	Bag	$\mathbb{N}$	$\exists n \in \mathbb{N} :\ P_x[n] = 1$
Probabilistic	Set	$\mathbb{B}$	$P_x[\top], P_x[\bot] \in [0, 1]$
Probabilistic	Bag	$\mathbb{N}$	$\forall n \in \mathbb{N} :\ P_x[n] \in [0, 1]$

Figure 5.4: Database semantics for different semirings S and probability distributions.

of Equation (5.1), we can equivalently see $[\Phi\theta\Psi]$ as a binary operation $S \times S \to S$ or $M \times M \to S$.

Example 41. The annotations in the pvc-table Q_2 in Figure 5.1 contain conditional expressions representing comparisons between semimodule expressions and a constant from M and semiring expressions and the constant 0_K. □

Set semantics vs. bag semantics. The pvc-tables system is generic enough to encode both deterministic and probabilistic databases with set and bag semantics. Figure 5.4 summarises how different choices for the semiring S and probability distributions for variables $x \in \mathbf{X}$ give rise to those semantics.

Under the Boolean semiring, annotation expressions are evaluated to $\bot$ or $\top$, denoting tuples *in* or *not in* the database instance, respectively. If in addition every variable $x \in \mathbf{X}$ has either $P_x[\bot] = 1$ or $P_x[\top] = 1$, then we have the case of deterministic databases with set semantics, i.e., there is exactly one possible world with non-zero probability.

For deterministic bag semantics, tuple annotations are evaluated to $\mathbb{N}$ and represent tuple multiplicities and for each variable $x \in \mathbf{X}$ there is exactly one $n \in \mathbb{N}$ for which $P_x[n] = 1$. For probabilistic databases with set semantics, every answer tuple has a probability for being true or false, while in the case of bag semantics, the pvc-table encodes a probability distribution over the multiplicity of its tuples.

Existing representation systems for probabilistic data are over the Boolean semiring, i.e., their semantics is set-based. While this choice is justified in the absence of aggregation (or for MIN/MAX), it is *not* for COUNT/SUM/PROD aggregates, which require bag semantics. In the latter case, we need a semiring for which there is a homomorphism into $\mathbb{N}$ [5]. The set-based probabilistic database model is subsumed by the probabilistic bag-based model: Every $x \in \mathbf{X}$ is an $\mathbb{N}$-valued random variable, yet the probability distributions P_x are non-zero only for $0, 1 \in \mathbb{N}$.

5.2 Annotation propagation

As in the previous chapters, we take the *intensional* approach to query evaluation of positive relational algebra queries with aggregates on pvc-tables; this approach has two steps: In the first step we compute the tuples and their annotation expressions in the query result,

and in the second step we compute their probability distributions. We discuss the first step in this section and the second step in Section 5.3.

Query language. The query language under consideration is a restriction of positive relational algebra extended by an operator ϖ for aggregation and grouping and is denoted by RA^{ϖ}. Given a relation R over schema Σ, the operator ϖ in the query

$$\varpi_{\bar{A};\alpha_1 \leftarrow \mathrm{AGG}_1(B_1),\ldots,\alpha_l \leftarrow \mathrm{AGG}_l(B_l)}(R),$$

where $(\bar{A} \cup \{B_1, \ldots, B_l\}) \subseteq \Sigma$, groups by the attributes $\bar{A}$ and takes the aggregations AGG_1 to AGG_l over the attributes B_1 to B_l respectively [2]. The result has schema $\bar{A} \cup \{\alpha_1, \ldots, \alpha_l\}$, where α_1 to α_l are *aggregation attributes*. We consider SUM, PROD, COUNT, MAX, and MIN aggregations, and restrict the queries such that projection, union and grouping are never applied to aggregation attributes. This restriction simplifies query rewriting, but is not requisite for our query evaluation method. It is known that queries beyond this restriction can be treated within the same formal algebraic framework of query annotations [5]. Out of the 22 TPC-H queries, only query Q13 violates the restriction.

Definition 30. *The query language RA^{ϖ} considered in this chapter consists of queries that are built using the relational operators $\delta, \sigma, \pi, \times, \cup, \varpi$ and satisfy the following constraints:*

1. *In $\pi_{\bar{A}}(Q)$ and $\varpi_{\bar{A};\alpha_1 \leftarrow AGG_1(B_1),\ldots,\alpha_l \leftarrow AGG_l(B_l)}(Q)$, the attributes in $\bar{A}$ are not aggregation attributes.*
2. *In $Q_1 \cup Q_2$, the attributes of Q_1 and Q_2 are not aggregation attributes.*

Example 42. The TPC-H query Q1 has the structure

```
SELECT A,SUM(B) FROM R GROUP BY A
```

which is equivalent to $\varpi_{A;\beta \leftarrow \mathrm{SUM}(B)}(R)$ when written as a RA^{ϖ} query. TPC-H query Q2 has the structure

```
SELECT A FROM R WHERE B = (SELECT MIN(C) FROM S),
```

or, equivalently, $\pi_A \sigma_{B=\gamma}\big(R \times \varpi_{\emptyset;\gamma \leftarrow \mathrm{MIN}(C)}(S)\big)$. The query $R \cup \varpi_{A;\beta \leftarrow \mathrm{SUM}(B)}(S)$ is not in RA^{ϖ}, since the second union term is a relation with the aggregation attribute β and hence violates constraint 2 in Def. 30. However, $\pi_A(R) \cup \pi_A \sigma_{\beta \geq 5}\big(\varpi_{A;\beta \leftarrow \mathrm{SUM}(B)}(S)\big)$ is a valid RA^{ϖ}-query. □

Propagation of annotations. The tuples in the result of a query Q can have semimodule expressions as values and semiring expressions as annotations. The construction of such

$$
\begin{aligned}
[\![R]\!] &= \text{ select R.*, R.}\Phi \text{ from R}\\
[\![\delta_{B\leftarrow A}(Q)]\!] &= \text{ select R.*, R.A as B, R.}\Phi \text{ as } \Phi \text{ from } \big([\![Q]\!]\big) \text{ R}\\
[\![\sigma_{A\theta B}(Q)]\!] &= \text{ select R.*, R.}\Phi \cdot_K [A\theta B] \text{ as } \Phi \text{ from } \big([\![Q]\!]\big) \text{ R}\\
[\![\pi_{A_1,\ldots,A_n}(Q)]\!] &= \text{ select R.A}_1, \ldots, \text{R.A}_n, \textstyle\sum_K\big(R.\Phi\big) \text{ as } \Phi \text{ from } \big([\![Q]\!]\big) \text{ R}\\
&\qquad\qquad \text{group by R.A}_1, \ldots, \text{R.A}_n\\
[\![Q_1 \times Q_2]\!] &= \text{ select R.*, S.*, R.}\Phi \cdot_K \text{S.}\Phi \text{ as } \Phi \text{ from } \big([\![Q_1]\!]\big) \text{ R}, \big([\![Q_2]\!]\big) \text{ S}\\
[\![Q_1 \cup Q_2]\!] &= \text{ select R.*, } \textstyle\sum_K\big(R.\Phi\big) \text{ as } \Phi \text{ from}\\
&\qquad \Big(\text{select * from } \big([\![Q_1]\!]\big) \text{ union all select * from } \big([\![Q_2]\!]\big)\Big) \text{ R group by R.*}\\
[\![\varpi_{A_1,\ldots,A_n;\alpha_1\leftarrow \mathrm{AGG}_1(B_1),\ldots,\alpha_l\leftarrow \mathrm{AGG}_l(B_l)}(Q)]\!] &= \text{ select R.A}_1, \ldots, \text{R.A}_n, \Gamma_1 \text{ as } \alpha_1, \ldots, \Gamma_l \text{ as } \alpha_l,\\
&\qquad \Big[\Big(\textstyle\sum_K \text{R.}\Phi\Big) \neq 0_K\Big] \text{ as } \Phi \text{ from } \big([\![Q]\!]\big) \text{ R group by R.A}_1, \ldots, \text{R.A}_n\\
[\![\varpi_{\emptyset;\alpha_1\leftarrow \mathrm{AGG}_1(B_1),\ldots,\alpha_l\leftarrow \mathrm{AGG}_l(B_l)}(Q)]\!] &= \text{ select } \Gamma_1 \text{ as } \alpha_1, \ldots, \Gamma_l \text{ as } \alpha_l, 1_K \text{ as } \Phi \text{ from } \big([\![Q]\!]\big) \text{ R}\\
&\text{where } \Gamma_i = \begin{cases} \sum_{\mathrm{AGG}_i}\big(\text{R.}\Phi \otimes \text{R.B}_i\big) & \text{if AGG}_i = \text{MIN,MAX,SUM,PROD}\\ \sum_{\mathrm{SUM}}\big(\text{R.}\Phi \otimes 1\big) & \text{if AGG}_i = \text{COUNT}\end{cases}
\end{aligned}
$$

Algorithm 11: Recursive algorithm $[\![\cdot]\!]$ for rewriting a RA^{ϖ} query to account for computation of semiring (K) and semimodule expressions. We assume that R.*, S.* do not select column Φ.

expressions can be done by a query that can be statically inferred from Q [5]. Algorithm 11 is a translation $[\![\cdot]\!]$ of RA^{ϖ} queries into SQL queries that compute the tuples in the query results. The rewriting is similar to the rewriting for queries without aggregation (cf. Algorithm 1 in Section 2) in that it accounts for joint and alternative use of data: Joint use of data (as in join) corresponds to multiplication in the annotation semiring and alternative use of data (as in union or projection) corresponds to summation in the annotation semiring. Semimodule expressions are constructed for the aggregation and grouping operator ϖ. The translation of the operators rename, selection, projection, product, and union is equivalent to the case of pc-tables (cf. Algorithm 1). In case of ϖ, the translation depends on the presence of group-by attributes. The translation uses the custom SQL operators $\sum_{\mathrm{AGG}}, \sum_K, \cdot_K$ and $\otimes$ to construct annotation expressions.

Example 43. The query $\varpi_{\emptyset;\alpha\leftarrow \mathrm{AGG(weight)}}(P_1)$ from Example 39 is rewritten into

$$\texttt{select } \sum_{\texttt{AGG}}\big(\texttt{R.}\Phi \otimes \texttt{R.weight}\big) \texttt{ as } \alpha\texttt{, } 1_K \texttt{ as } \Phi \texttt{ from (select * from P}_1\texttt{) R}$$

The answer to the rewritten query is one tuple with one attribute α that takes value $\alpha = z_1 \otimes 4 \ +_{\mathrm{AGG}} \ z_2 \otimes 8 \ +_{\mathrm{AGG}} \ z_3 \otimes 7 \ +_{\mathrm{AGG}} \ z_4 \otimes 6$. The annotation of this tuple is the constant 1_K, stating that the tuple $\langle\alpha\rangle$ is the query answer in all possible worlds. The evaluation of α may however yield different outcomes in different worlds.

The query $\pi_{\emptyset}\sigma_{5\leq\alpha}\big(\varpi_{\emptyset;\alpha\leftarrow \mathrm{MIN(weight)}}(P_1)\big)$ is rewritten to

$$\texttt{select S.}\Phi \cdot [5 \leq \texttt{S.}\alpha] \texttt{ as } \Phi \texttt{ from } \Big(\texttt{select } \sum_{\texttt{MIN}} (\texttt{R.}\Phi \otimes \texttt{R.weight}) \texttt{ as } \alpha\texttt{, } 1_K \texttt{ as } \Phi \texttt{ from (select * from P}_1\texttt{) R}\Big) \texttt{ S}$$

This Boolean query asks for the probability of the minimum weight of articles being larger than 5. The result of this query is a single empty tuple with annotation $\Phi = 1_K \cdot [z_1 \otimes 4 \ +_{\min} \ z_2 \otimes 8 \ +_{\min} \ z_3 \otimes 7 \ +_{\min} \ z_4 \otimes 6 \leq 5]$. □

As seen in the above example, for aggregation without grouping the resulting semimodule value is annotated with 1_K, i.e., it "exists" in every possible world. In case of grouping, the resulting tuples are additionally annotated with a conditional expression to enforce that the group is not empty and thus at least one tuple in the group has an annotation different from 0_K in a possible world. This has been exemplified in the introduction to this chapter for the query Q_2. However, this conditional expression is not always necessary:

Example 44. Consider the query

$$Q_2' = \pi_{\text{shop}} \sigma_{P \leq 50} \varpi_{\text{shop;P} \leftarrow \text{MIN(price)}}[Q_1]$$

which is similar to Q_2 on the database in Figure 5.1. Under the Boolean semiring K, the variables can take values $\bot = 0_K$ and $\top = 1_K$. In a possible world with $x_1, x_2, x_3 \mapsto \bot$, $\langle$M&S$\rangle$ is not an answer since there is no supplier for the shop M&S in the input instance S. Its annotation evaluates to $[\infty \leq 50] \cdot \bot = \bot \cdot \bot = \bot$. Here, the conditional expression Ψ_1 from Figure 5.1 would not be necessary, since it is implied by the first conditional expression in the expression Φ. In case of other monoids, such as MAX (as discussed in the introduction), SUM, or COUNT, Ψ_1 is necessary. □

The constraints imposed on the query language RA^{ϖ} by Definition 30 simplify the rewriting $[\![\cdot]\!]$: Since projections and unions are only done on non-aggregate columns, the rewriting rules for those operators can assume the input tuples to be free of semimodule expressions.

Closure of pvc-tables under RA^{ϖ} queries. Since pvc-tables generalise pc-tables and the latter are complete in the sense that any finite probability distribution over relational databases can be represented using pc-tables [79], it follows that pvc-tables also form a complete representation system. In particular, they are closed under RA^{ϖ} queries, in the sense that for any input pvc-database, the result of any RA^{ϖ} query can be represented as a pvc-table. The pvc-tables representing query results are also succinct (for a fixed RA^{ϖ} query), since the translation $[\![\cdot]\!]$ only constructs SQL queries of size linear in the size of the input RA^{ϖ} queries and the evaluation of such SQL queries is in polynomial time data complexity.

Theorem 50 (Completeness and Succinctness)**.**

1. *Any finite probability distribution over relational database instances can be represented by pvc-databases.*
2. *Given a pvc-database $\mathcal{D}$ and a fixed RA^{ϖ} query Q, the query result $[\![Q]\!](\mathcal{D})$ can be represented as a pvc-table with size polynomial in the size of $\mathcal{D}$.*

Proof. The first part follows from the facts that pc-tables are a complete representation system [79] and that pvc-tables are a generalisation of pc-tables. The second part was shown in [5]. □

In particular, pvc-tables can be exponentially more succinct than pc-tables. The key difference is that pvc-tables allow for values and annotations to be intertwined in semimodule expressions, which can encode exponentially many possible outcomes of an aggregation in polynomial space. In pc-tables, we would need to enumerate all these outcomes [56]. The latter result together with the succinctness result from Theorem 50 explains the exponential gap in succinctness between pc-tables and pvc-tables.

5.3 Probability computation by decomposition

This section presents a novel approach to computing probability distributions for query results on pvc-tables; this problem is equivalent to computing probability distributions of the results' semiring and semimodule expressions. Our approach is based on compiling expressions into a tractable form called decomposition trees, for which distributions can be computed efficiently. A key property of our approach is its generality, as it applies to different semirings and semimodules.

5.3.1 Expressions and random variables

We first show how semiring and semimodule expressions — including conditional expressions — in pvc-tables give rise to random variables. Let S be a countable semiring and M a monoid, $\mathbf{X}$ a set of S-valued random variables, and Ω the probability space induced by $\mathbf{X}$; K and $K \otimes M$ are the sets of semiring and semimodule expressions over $\mathbf{X}$. A semiring expression $\Phi \in K$ can be seen as a random variable $\Phi : \Omega \to S$ by letting $\Phi : \nu \mapsto \nu(\Phi)$ and with probability distribution

$$P_\Phi[s] = \Pr\big(\{\nu \in \Omega \mid \nu(\Phi){=}s\}\big) = \sum_{\substack{\nu\in\Omega:\\ \nu(\Phi)=s}} \Pr(\nu). \tag{5.2}$$

A semimodule expression $\alpha \in K \otimes M$ is a random variable $\alpha : \Omega \to M$ with $\alpha : \nu \mapsto \nu(\alpha)$ and probability distribution P_α defined similar to Equation (5.2) for all $m \in M$. If M is over $\mathbb{R}$, then α is an $\mathbb{R}$-valued random variable. However, since the random variables $\mathbf{X}$ and hence the probability space Ω are countable, the set of values that α may take is countable.

Probability distributions of independent expressions. Two semiring or semimodule expressions are *(syntactically) independent* if their sets of variables are disjoint; it is clear that independent expressions are independent random variables.

Example 45. Let $\mathbf{X} = \{x, y, a, b, c\}$, $M = \mathbb{N}$, $\Phi = x + y$ and $\alpha = a(b + c) \otimes 10 + c \otimes 20$. Then Φ and α are independent random variables since their sets of variables are disjoint. □

The probability distribution of the sum of two independent random variables is the convolution of their individual distributions [42]. For instance, given two random variables x, y over positive integers, the probability that the sum of the random variables equals to 4 is the sum of the probabilities of x being 0 and y being 4, of x being 1 and y being 3, and so on. The applicability of convolution to determine the probability distribution of a function of independent random variables is not limited to sums of integers:

Proposition 51 ([42])**.** *Given sets A, B, C, an A-valued random variable x with probability distribution P_x, a B-valued random variable y with probability distribution P_y, x independent of y, and an operation $\bullet : A \times B \to C$, the probability distribution $P_{x \bullet y}$ of the C-valued random variable $z = x \bullet y$ is the convolution of P_x and P_y with respect to $\bullet$:*

$$P_{x \bullet y}[c] = \sum_{\substack{(a,b) \in A \times B: \\ c = a \bullet b}} P_x[a]\, P_y[b] \qquad \text{for all } c \in C \tag{5.3}$$

Example 46. Equation (2.5) for the probability $P_{\Phi \vee \Psi}$ of the disjunction $\Phi \vee \Psi$ of two independent Boolean random variables is a special case of Equation (5.3):

$$\begin{aligned} P_{\Phi \vee \Psi}[\top] &= \sum_{\substack{(a,b) \in \mathbb{B} \times \mathbb{B}: \\ \top = a \vee b}} P_\Phi[a]\, P_\Psi[b] \\ &= P_\Phi[\top] P_\Psi[\top] + P_\Phi[\bot] P_\Psi[\top] + P_\Phi[\top] P_\Psi[\bot] \\ &= 1 - (1 - P_\Phi[\top])(1 - P_\Psi[\top]) \end{aligned}$$ □

Remark 9. The sum in Equation (5.3) is invariant under the restriction to those a and b for which $P_x[a] > 0$ and $P_y[b] > 0$. Hence, even if the cardinality of $A \times B$ is infinite, the convolution is a finite sum whenever only finitely many elements in A and B have non-zero probability.

Since expressions can be interpreted as random variables (cf. Equation 5.2), we can apply convolution as per Proposition 51 to obtain equations for the probability distribution of the

sum or product of independent semiring or semimodule expressions. Given independent expressions $\Phi, \Psi \in K$ and $\alpha, \beta \in K \otimes M$ with probability distributions $P_\Phi, P_\Psi, P_\alpha, P_\beta$, we have for all $s \in S, m \in M$:

$$P_{\Phi+\Psi}[s] = \sum_{s_1,s_2 \in S : s_1+s_2=s} P_\Phi[s_1] \cdot P_\Psi[s_2] \tag{5.4}$$

$$P_{\Phi.\Psi}[s] = \sum_{s_1,s_2 \in S : s_1 \cdot s_2=s} P_\Phi[s_1] \cdot P_\Psi[s_2] \tag{5.5}$$

$$P_{\alpha+\beta}[m] = \sum_{m_1,m_2 \in M : m_1+m_2=m} P_\alpha[m_1] \cdot P_\beta[m_2] \tag{5.6}$$

$$P_{\Phi\otimes\alpha}[m] = \sum_{s_1 \in S, m_1 \in M : s_1 \otimes m_1=m} P_\Phi[s_1] \cdot P_\alpha[m_1] \tag{5.7}$$

$$P_{[\alpha\theta\beta]}[s] = \sum_{m_1 \in M, m_2 \in M : [m_1 \theta m_2]=s} P_\alpha[m_1] \cdot P_\beta[m_2] \tag{5.8}$$

$$P_{[\Phi\theta\Psi]}[s] = \sum_{s_1 \in S, s_2 \in S : [s_1 \theta s_2]=s} P_\Phi[s_1] \cdot P_\Psi[s_2] \tag{5.9}$$

Following Remark 9, the sums in the above equations are restricted to summands of non-zero probabilities.

Example 47. Let S and M be the semiring and the monoid of natural numbers with standard addition and multiplication. Let $\Phi = x$ with $P_x = \{(0, 0.3), (1, 0.3), (2, 0.4)\}$, and $\alpha = y \otimes 5$ be a semimodule expression with $P_y = \{(1, 0.4), (2, 0.4), (3, 0.2)\}$. Then α is a M-valued random variable with $P_\alpha = \{(5, 0.4), (10, 0.4), (15, 0.2)\}$. The probability distribution of $\Phi \otimes \alpha = x \otimes (y \otimes 5) = (x \cdot_S y) \otimes 5$ is given by Equation (5.7). For example, $P_{\Phi\otimes\alpha}[10] = P_x[1]P_\alpha[10] + P_x[2]P_\alpha[5]$. The convolution is finite as the probability distributions are non-zero for finitely many elements. Further possible outcomes for $\Phi \otimes \alpha$ are 0, 5, 15, 20, 30. In case of the Boolean semiring, $S{=}\mathbb{B}$, possible outcomes are 0 and 5 with $P_{\Phi\otimes\alpha}[5] = P_x[\top]P_y[\top]$ and $P_{\Phi\otimes\alpha}[0] = 1 - P_{\Phi\otimes\alpha}[5]$. □

Partitioning into mutually exclusive expressions. Given an expression Φ and a variable $x \in \mathbf{X}$ that occurs in Φ, the probability distribution of Φ can be expressed using probability distributions of sub-expressions under valuations of x; this decomposition can be understood as generalised Shannon expansion. For any $s' \in S$, we denote by $\Phi|_{x \leftarrow s'}$ the expression obtained from Φ by replacing every occurrence of x by s'. Then the probability distribution of Φ can be partitioned by the probability distributions of expressions $\Phi|_{x \leftarrow s'}$:

$$P_\Phi[s] = \sum_{s' \in S} P_x[s'] \cdot P_{\Phi|_{x \leftarrow s'}}[s]. \tag{5.10}$$

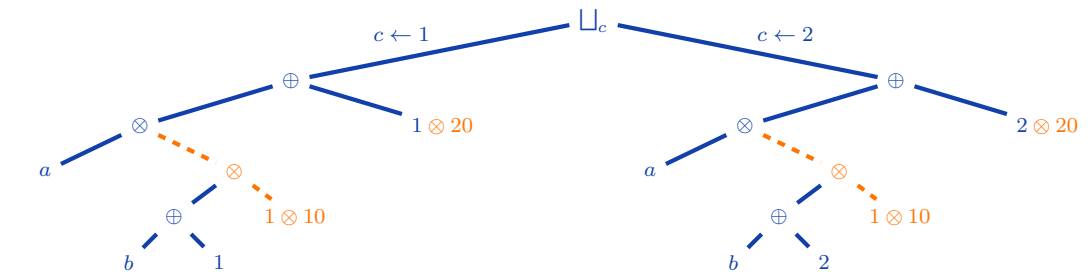


Figure 5.5: A d-tree for $\alpha = a(b + c) \otimes 10 + c \otimes 20$ over the semimodule $\mathbb{N} \otimes \mathbb{N}$. The node $\bigsqcup_c$ has one child for every $k \in \mathbb{N}$ with non-zero probability for $c \leftarrow k$, i.e., two children for $c = 1, 2$ in the setting of Ex.48. The solid (blue) part is a d-tree for the semiring component $a(b + c) + c$, where $\otimes$ is replaced by $\odot$. The dashed (orange) part shows the semimodule component.

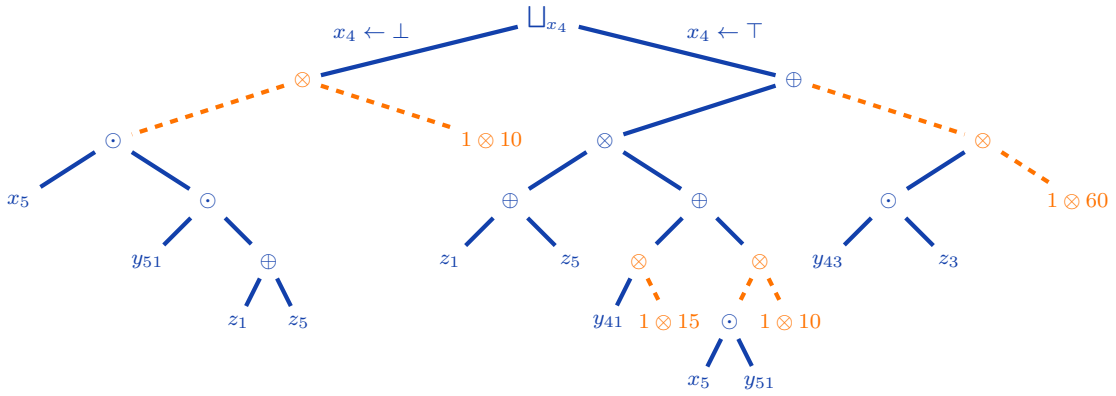


Figure 5.6: D-tree for the semimodule expression $x_4y_{41}(z_1 + z_5) \otimes 15 +_{\max} x_4y_{43}z_3 \otimes 60 +_{\max} x_5y_{51}(z_1 + z_5) \otimes 10$ (part of the annotation of tuple $\langle \text{Gap} \rangle$ in Figure 5.1f) over the semimodule $\mathbb{B} \otimes \mathbb{N}$. The solid (blue) part is a d-tree for the semiring component of the expression, where $\otimes$ is replaced by $\odot$. The dashed (orange) part shows the semimodule component.

5.3.2 Decomposition trees

Decomposition trees (d-trees) are a normal form for semiring and semimodule expressions and represent their probability distributions. We next define them and then show how to compile arbitrary expressions into d-trees.

Definition 31. *Let M be a monoid and K be a semiring generated by an S-valued set $\mathbf{X}$ of variables and constants from S. A* decomposition tree*, or d-tree, is a tree, where each inner node is one of $\oplus$, $\odot$, $\otimes$, $\bigsqcup$, or $[\theta]$ and each leaf node is a variable in $\mathbf{X}$ or a constant in S, M, or $S \otimes M$. The five types of inner nodes have the following meaning.*

1. *A node $\oplus$ with children representing Φ and Ψ represents the expression $\Phi + \Psi$, where Φ and Ψ are either independent expressions in K, or independent expressions in $K \otimes M$.*

2. *A node $\odot$ with children representing Φ and Ψ represents the expression $\Phi \cdot \Psi$, where Φ and Ψ are independent semiring expressions in K.*

3. *A node $\otimes$ with children representing Φ and α represents the expression $\Phi \otimes \alpha$, where Φ and α are independent expressions in K and $K \otimes M$, respectively.*

4. *A node $[\theta]$ with children representing Φ and Ψ represents the expression $[\Phi\theta\Psi]$, where Φ and Ψ are independent expressions in K or $K \otimes M$.*

5. *Given a variable $x \in \mathbf{X}$, an inner node $\bigsqcup_x$ with children representing $\Phi|_{x \leftarrow s_1}, \ldots, \Phi|_{x \leftarrow s_n}$ for all those $s_i \in S$ with $P_x[s_i] \neq 0$ represents the expression Φ.*

Just like semiring or semimodule expressions, d-trees represent probability distributions. Direct implementations of Equations (5.4) through (5.10) are efficient procedures to compute the probability distribution at any inner node of a d-tree, given the probability distributions at its children: Equations (5.4) and (5.6) apply to $\oplus$ nodes, Equation (5.5) applies to $\odot$ nodes, Equation (5.7) applies to $\otimes$ nodes, Equations (5.8) and (5.9) apply to $[\theta]$, and Equation (5.10) applies to $\bigsqcup_x$ nodes. For leaves, we only have the trivial cases of a variable x with probability distribution P_x, of constant $s \in S$ or $m \in M$ with distribution $\{(s, 1)\}$ and $\{(m, 1)\}$, and of constant semimodule expressions of the form $s \otimes m$ with probability distribution $\{(s \otimes m, 1)\}$. The probability distribution of the entire d-tree is the distribution of its root and can be computed bottom-up in one pass over the d-tree.

Example 48. Consider the d-tree from Figure 5.5 in which it is assumed that each variable $x \in \{a, b, c\}$ has non-zero probabilities p_x and $\bar{p}_x = 1 - p_x$ for values 1 and 2, respectively. We first compute the probability distribution for the left branch of the d-tree under the aggregation monoid SUM. The distributions are:

$$\begin{aligned}
\{(1, p_b), (2, \bar{p}_b)\} &\quad \text{for } b \\
\{(1, 1)\} &\quad \text{for } 1 \\
\{(2, p_b), (3, \bar{p}_b)\} &\quad \text{for } b \oplus 1 \\
\{(10, 1)\} &\quad \text{for } 1 \otimes 10 \\
\{(20, p_b), (30, \bar{p}_b)\} &\quad \text{for } (b \oplus 1) \otimes 10 \\
\{(1, p_a), (2, \bar{p}_a)\} &\quad \text{for } a \\
\{(20, p_a p_b), (30, p_a \bar{p}_b), (40, \bar{p}_a p_b), (60, \bar{p}_a \bar{p}_b)\} &\quad \text{for } a(b + 1) \otimes 10 \\
\{(20, 1)\} &\quad \text{for } 1 \otimes 20 \\
\{(40, p_a p_b), (50, p_a \bar{p}_b), (60, \bar{p}_a p_b), (80, \bar{p}_a \bar{p}_b)\} &\quad \text{for } a(b + 1) \otimes 10 + 1 \otimes 20
\end{aligned}$$

and finally

$$\{(40, p_a p_b p_c), (50, p_a \bar{p}_b p_c), (60, \bar{p}_a p_b p_c), (80, \bar{p}_a \bar{p}_b p_c)\} \quad \text{for the entire left branch.}$$

For the right branch one obtains

$$\{(70, p_a p_b \bar{p}_c), (80, p_a \bar{p}_b \bar{p}_c), (100, \bar{p}_a p_b \bar{p}_c), (120, \bar{p}_a \bar{p}_b \bar{p}_c)\}.$$

The probability distribution of the entire d-tree is:

$$\{(40, p_a p_b p_c), (50, p_a \bar{p}_b p_c), (60, \bar{p}_a p_b p_c), (70, p_a p_b \bar{p}_c),$$
$$(80, \bar{p}_a \bar{p}_b p_c + p_a \bar{p}_b \bar{p}_c), (100, \bar{p}_a p_b \bar{p}_c), (120, \bar{p}_a \bar{p}_b \bar{p}_c)\}$$

In case of MIN aggregation, the distribution for both branches as well as for the entire d-tree is $\{(10, 1)\}$. In case of the Boolean semiring and MIN aggregation, the distribution for the left branch ($c \leftarrow \bot$) is $\{(10, p_a p_b \bar{p}_c), (\infty, p_a \bar{p}_b \bar{p}_c + \bar{p}_a p_b \bar{p}_c + \bar{p}_a \bar{p}_b \bar{p}_c)\}$, for the right branch ($c \leftarrow \top$) is $\{(10, p_a p_c), (20, \bar{p}_a p_c)\}$, and for the overall d-tree $\{(10, p_a p_b \bar{p}_c + p_a p_c)$, $(20, \bar{p}_a p_c)$, $(\infty, p_a \bar{p}_b \bar{p}_c + \bar{p}_a p_b \bar{p}_c + \bar{p}_a \bar{p}_b \bar{p}_c)\}$. □

We now turn to analysing the complexity of computing the probability distribution for a given d-tree; the following theorem provides an upper bound on the time required to evaluate the probability of a d-tree:

Theorem 52. *The probability distribution P_d of a d-tree d whose nodes have probability distributions $p_1, \ldots, p_n$ can be computed in time $\mathcal{O}(\prod |p_i|)$.*

Proof. Recall that the size $|p|$ of a probability distribution is the size of its set representation, cf. page 137. Since d-trees are binary, the distribution of a convolution node can be computed in time linear in each of the sizes of the distributions of the children according to Equation (5.3). The distribution of a $\bigsqcup$-node is computed in time linear in the number of its children and their probability distributions. The overall time required is thus bounded by the product of the sizes of the probability distributions in the tree, $\mathcal{O}(\prod |p_i|)$. □

For the SUM monoid, the size of P_d can be exponential in the number n of leaves, since there may be exponentially many distinct sums out of n numbers. We analyse two classes of d-trees for which we can obtain a better upper bound on the time complexity of computing P_d.

Firstly, observe that the sum of two elements in the MIN or MAX monoid is one of the elements itself (e.g., $a +_{\min} b$ is either a or b). This implies that the size of the probability distribution of a semimodule expression α is bounded by the number of monoid values occurring at the leaves of d. Moreover, if α's variables are $\mathbb{N}$-valued, α's probability distribution is unaltered under the following *reduction* of its variables to $\mathbb{B}$-valued variables: $P_x[\bot] = P_x[0]$ and $P_x[\top] = 1 - P_x[\bot]$. Hence, one may equivalently consider a d-tree for $\mathbb{B}$-valued variables obtained from this reduction:

Proposition 53. *Given a semimodule expression α over MIN or MAX, the distribution P_α can be computed in time linear in the size of α's d-tree in which all variables are considered by their reduction to $\mathbb{B}$-valued variables.*

COMPILE (Expression Φ)
  if Φ *has no variables* **then**
    return Φ
  if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 + \Phi_2 = \Phi$ **then**
    return COMPILE(Φ_1) $\oplus$ COMPILE(Φ_2)
  if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 \cdot \Phi_2 = \Phi$ **then**
    return COMPILE(Φ_1) $\odot$ COMPILE(Φ_2)
  if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $\Phi_1 \otimes \Phi_2 = \Phi$ **then**
    return COMPILE(Φ_1) $\otimes$ COMPILE(Φ_2)
  if $\exists$ *independent* Φ_1, Φ_2 *s.t.* $[\Phi_1 \theta \Phi_2] = \Phi$ **then**
    return COMPILE(Φ_1) $[\theta]$ COMPILE(Φ_2)
  Choose variable $x \in \mathbf{X}$ occurring in Φ
  return $\bigsqcup_x$ $(\forall s \in S, P_x[s] \neq 0$: COMPILE($\Phi_s$))

Algorithm 12: Compilation of semimodule or semiring expressions into d-trees.

Proof. Since all variables are $\mathbb{B}$-valued, the probability distribution at each node of the d-tree is at most of size 2. The convolution at each node can therefore be computed in constant time. This yields an overall time complexity linear in the number of nodes, i.e. linear in the size of the d-tree. □

Secondly, we analyse SUM aggregation. Evaluating a SUM expression $\sum x_i \otimes m_i$ in which all x_i are independent variables is NP-hard [72]. Yet, instances in which the values m_i are constrained are tractable: we say that a semimodule expression $\sum \varphi_i \otimes m_i$ over the SUM monoid $\mathbb{N}$ is *m-bounded* if there is a constant $m \in \mathbb{N}$ such that $\forall i : 0 \leq m_i \leq m$. Aggregation over rationals from a bounded set can be regarded as bounded if the number of decimal places is fixed, e.g., for currencies with 2 decimal places.

Proposition 54. *Let $\alpha = \sum_{i=1}^{n} \varphi_i \otimes m_i$ be an m-bounded semimodule expression over the SUM monoid $\mathbb{N}$ where each φ_i is a product of variables and all variables have non-zero probability only for 0_S and 1_S. The probability distribution of a d-tree d of size $|d|$ for α can be computed in time $\mathcal{O}(n^2 m^2 |d|)$.*

Proof. First note that every φ_i may only evaluate to 0_S or 1_S and hence the sum is bounded by the product of the number of terms, n, and m; the product $n \cdot m$ is thus an upper bound for the size of the probability distribution at each node and the convolution at each node can be computed in time $\mathcal{O}(n^2 m^2)$. Since there are $|d|$ nodes in the d-tree, this yields an overall time complexity of $\mathcal{O}(n^2 m^2 |d|)$. □

The special case of COUNT aggregation corresponds to $m_i = 1$ for all i. Then the resulting probability distribution has at most size n and can be computed in time $\mathcal{O}(n^2|d|)$. In a d-tree that contains semimodule expressions α, β over different aggregation monoids, the complexity of the sub-trees of α and β is each according to Propositions 53 and 54.

Compiling expressions into d-trees. Algorithm 12 constructs a d-tree for an input expression Φ by repeatedly applying six decomposition rules. The obtained d-tree is equivalent to the input expression in the sense that they represent the same probability distribution.

The first four rules check whether the input expression can be partitioned and decomposed into two independent expressions Φ_1 and Φ_2. In the first rule, Φ_1 and Φ_2 are either both semimodule or both semiring expressions; in the second rule, they are semiring expressions. In the third rule, Φ_1 is a semiring, and Φ_2 is a semimodule expression.

In order to find independent decompositions in polynomial time, expressions are analysed at a syntactic level: We attempt the first rule only if Φ is a sum and partition it by the connected components in its clause-dependency graph. In addition to such syntactic manipulations, the decomposition in the second and third rules uses known polynomial-time algorithms to recognise *read-once expressions*, i.e., expressions where each variable occurs once, and hence factorise expressions based on algebraic rewritings such as the associativity and commutativity laws, e.g., [37, 63]. In particular, this approach allows to factor expressions into complex sub-expressions and not only into one variable and the residual.

The last rule decomposes Φ into sub-expressions Φ_s, for each $s \in S$. As explained for Equation (5.10), each expression Φ_s is obtained in linear time by replacing x with the constant s in Φ. Many heuristics have been proposed for choosing the variable x, since good choices can make the difference between polynomial and exponential size decision diagrams (such as ordered binary decision diagrams, d-DNNFs, or d-trees). In our implementation, we choose a variable with most occurrences [63].

Remark 10. The requirement that the underlying algebraic structures be commutative and associative is crucial for structural decomposition: without these properties, expression decomposition would be constrained to the fixed order defined by the order of symbols and the nesting of the expression.

Example 49. Figure 5.5 depicts a d-tree for the semimodule expression $\Phi = a(b+c) \otimes 10 + c \otimes 20$. Φ cannot be split into independent sub-expressions since variable c occurs in both summands. We choose to eliminate variable c to create mutually exclusive events. (This is a good choice since it leads to independent sub-expressions.) We thus create a node $\bigsqcup_c$ with as many children as valuations of c that have non-zero probability. Consider the case $c \leftarrow 1$ and hence $\Phi|_{c \leftarrow 1} = a(b+1) \otimes 10 + 1 \otimes 20$. This corresponds to the left branch in the d-tree, and can be decomposed using the first rule into its independent summands $a(b+1) \otimes 10$ and $1 \otimes 20$. The former expression can be decomposed using the third rule into independent expressions a and $(b+1) \otimes 10$. The procedure continues until we completely decompose the expressions into variables and semimodule constants.

Figure 5.6 shows a d-tree for the semimodule expression in the first conditional expression in the annotation of the result tuple $\langle \text{Gap} \rangle$ in Figure 5.1f over semimodule $\mathbb{B} \otimes \mathbb{N}$:

$$x_4 y_{41}(z_1 + z_5) \otimes 15 + x_4 y_{43} z_3 \otimes 60 + x_5 y_{51}(z_1 + z_5) \otimes 10$$

It cannot be partitioned into independent sub-expressions. Choosing variable x_4 in the last rule, we create a node $\bigsqcup_{x_4}$. Its left child $\Phi|_{x_4 \leftarrow \bot} = x_5 y_{51}(z_1 + z_5) \otimes 10$ can be decomposed using the third rule into $x_5 y_{51}(z_1 + z_5)$ and $1 \otimes 10$. The former is a read-once expression, on which the second rule can be applied twice to decompose it into x_5 and the rest, then the rest into y_{51} and $z_1 + z_5$. The latter is decomposed using the first rule into z_1 and z_5. The right child

$$\Phi|_{x_4 \leftarrow \top} = y_{41}(z_1 + z_5) \otimes 15 + y_{43} z_3 \otimes 60 + x_5 y_{51}(z_1 + z_5) \otimes 10$$

can be decomposed using a sequence of the first three rules that exploit the independence of sub-expressions.

By construction of query results using $[\![\cdot]\!]$, the semiring expression in the second conditional expression in Φ is precisely the semiring part of the above semimodule expression:

$$x_4 y_{41}(z_1 + z_5) + x_4 y_{43} z_3 + x_5 y_{51}(z_1 + z_5),$$

We can thus use the very same compilation steps as above, with the simplification that only the first two rules, which work on semirings, need to be applied. The d-tree is the same as for the semimodule expression, but where $\otimes$ nodes are replaced by $\odot$ and without semimodule expressions at leaves. Figure 5.6 shows this d-tree with solid (blue) edges. □

Proposition 55. *For any semimodule or semiring expression Φ, Algorithm 12 constructs a d-tree d such that $P_\Phi = P_d$. The time complexity of the compilation is bounded by $\mathcal{O}(|\Phi|^{|vars(\Phi)|})$ where $|vars(\Phi)|$ denotes the number of distinct variables occurring in Φ.*

Proof. The equivalence of P_Φ and P_d follows directly from the definition of decomposition trees (Definition 31) and the convolution equations that define the represented probability distributions (Equations (5.4) – (5.9)).

The algorithm is complete since the last rule is always applicable and the number of variables in the expression is strictly decreasing between successive recursive invocations. Decomposing Φ by the last rule only yields the upper bound for the time complexity: The algorithm constructs a binary tree whose depth is given by the number of distinct variables in Φ, and each decomposition step can be performed in time linear in the size of Φ. □

By virtue of this equivalence, we can compute the probability distribution of any expression by first compiling it into a d-tree d and then computing its probability distribution P_d. The algorithm is complete since the last rule is always applicable until all variables are evaluated. However, exclusive application of this rule is not desirable: compiling arbitrary expressions by applying the last rule only can lead to d-trees of size exponential in the number of variables. This limitation seems necessary: if we could compile any expression Φ into a d-tree in polynomial time, then hard problems such as satisfiability, tautology, and

even probability computation can be solved in polynomial time for Φ. In practice, however, this rule can be quite effective, as we show in the experiments. In case we need only apply the first four rules, the compilation finishes in polynomial time. This is already known for semiring expressions occurring in the results of tractable relational algebra queries without repeating symbols [63]. In Section 5.4, we define a fragment of our query language RA^{ϖ} consisting of tractable queries that only create expressions compilable using the first four rules.

We end this section with two observations on our compilation approach.

Pruning conditional expressions. The evaluation of $[\alpha\theta\beta]$-expressions can be considerably improved by employing pruning rules that prove parts of α or β redundant. Consider the expression $\Phi = [\alpha\theta\beta] = [x \otimes 10 +_{\min} y \otimes 20 \leq 15]$ which can be decomposed into subexpressions α and β. The probability $P_\Phi[1_S] = 1 - P_x[0_S]$ is independent of y; indeed, when computing P_α one may safely ignore computing $P_\alpha[20]$ since it cannot contribute to $P_\Phi[1_S]$ in the convolution of $[\theta]$. Equivalently, we may replace Φ with a simpler yet equivalent expression in which redundant terms are pruned. Examples of pruning rules are:

$$\text{MIN}: \qquad \left[\sum_i \Phi_i \otimes m_i \leq m\right] \equiv \left[\sum_{i:m_i \leq m} \Phi_i \otimes m_i \leq m\right]$$

$$\text{SUM}: \qquad \left[\sum_i \Phi_i \otimes m_i \leq m\right] \equiv 1_S \qquad \text{if } \sum_i m_i \leq m$$

Pruning is particularly effective when the probability distributions of α and β have exponential size, such as in case of the SUM monoid; there, early pruning can avoid the full materialisation of such probability distributions.

Compiling joint probability distributions. A tuple in the result of an aggregate query in RA^{ϖ} may have several semimodule expressions and is annotated with a conditional expression; in such cases we are interested in finding the joint probability distribution of these expressions. This can be accomplished by compiling the expressions into a single d-tree by applying mutex decomposition until some of the expressions become independent; the joint probability distribution of two independent random variables is simply their product. For instance, given integer variables a, b, c with non-zero probabilities for 1,2 only, the mutex decomposition on a decomposes the joint expression $\langle a + b, a \cdot c\rangle$ into two branches $\langle 1 + b, 1 \cdot c\rangle$ and $\langle 2 + b, 2 \cdot c\rangle$. In each branch, the two expressions are independent and can be considered separately. For example, the probability for the value $\langle 3, 2\rangle$ can be worked out to be $P_a[2]P_b[1]P_c[1] + P_a[1]P_b[2]P_c[2]$.

5.4 Tractable queries

This section describes the classes $\mathrm{RA}^{\varpi}_{\mathrm{ind}}$ and $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$ of tractable relational algebra queries with aggregation. The characterisation uses as building blocks queries that return tuple-independent relations and queries reminiscent of hierarchical queries [79]. Class $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$ uses $\mathrm{RA}^{\varpi}_{\mathrm{ind}}$ as a building block, i.e., $\mathrm{RA}^{\varpi}_{\mathrm{ind}} \subset \mathrm{RA}^{\varpi}_{\mathrm{hie}}$. Our tractability result follows from the discussion in this section:

Theorem 56 (Tractable Queries)**.**
Every query in RA^{ϖ}_{hie} has polynomial-time data complexity.

The proof is deferred to the end of this section. The characterisation of tractable queries is based on a generalisation of the *hierarchical* property for non-repeating conjunctive queries, viz. Definition 6. Definitions 32 and 33 rely on the following notions and assumptions. Every relation that does not contain semimodule expressions and whose tuples are annotated with distinct variable symbols with given probability distributions is called *tuple-independent*. We allow MIN and MAX aggregation as well as bounded SUM and COUNT aggregation, see Proposition 54. Additionally, we assume that for every aggregation monoid M occurring in a query, the constant 0_M does not occur in the database. Our tractability result depends on this assumption in the case of queries with grouping where it ensures that the probability distributions of the aggregate value and the existence conditions for the group are not correlated — cf. the reasoning following Definition 32. For MIN and MAX aggregation, this assumption is usually no significant limitation, since it disallows the values $+\infty$ and $-\infty$ in the data and these values arguably appear rarely in real data. For SUM aggregation, the value 0 may not appear in the database in any attribute that SUM aggregation is carried out on; this can constitute a limitation for applications in domains where 0 is a valid value, for example money, distances, times, etc.

We assume that in a selection σ_ρ, ρ is a conjunction of (1) equality predicates of the form $A{=}B$ or $A{=}c$ where A and B are non-aggregation attributes and c is a constant, and (2) θ-comparisons $\alpha\theta\beta$, $\alpha\theta c$, or $\alpha\theta A$ where α and β are aggregation attributes and A is a non-aggregation attribute.

We give separate recursive definitions of the classes $\mathrm{RA}^{\varpi}_{\mathrm{ind}}$ of queries whose result tuples are independent, and $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$ whose result tuples may be correlated.

Definition 32 (Class $\mathrm{RA}^{\varpi}_{\mathrm{ind}}$)**.**

1. *Every tuple-independent relation is a RA^{ϖ}_{ind}-query.*
2. *Let $Q_1, \ldots, Q_n \in RA^{\varpi}_{ind}$, and $\tilde{Q}_i = \varpi_{\bar{A}_i;\gamma_i \leftarrow AGG_i(C_i)}(Q_i)$. Then the following are RA^{ϖ}_{ind}-queries:*
   (a) $Q = \pi_{\bar{A}}\sigma_\rho(\tilde{Q}_1)$, such that $\gamma_1 \notin \bar{A}$

(b) $Q = \pi_{\bar{A}}\sigma_{\rho}(Q_1 \times \cdots \times Q_n)$*, such that* Q *is hierarchical and all attributes in* $\bar{A}$ *are root variables in* $Q_1, \ldots, Q_n$

(c) $Q = \pi_{\emptyset}\sigma_{\gamma_1\theta\gamma_2}\big(\tilde{Q}_1 \times \tilde{Q}_2\big)$*, such that* $\bar{A}_1 = \bar{A}_2 = \emptyset$*.*

Lemma 57. *For any tuple-independent database* $\mathcal{D}$ *and any* RA^{ϖ}_{ind}*-query* Q*, the relation* $Q(\mathcal{D})$ *is tuple-independent.*

Proof. Regarding the queries under Definition 32.2, first observe that the tuples in the result of $\tilde{Q}_i$ are independent and that the expressions γ_i have the form $\gamma_i = \sum_{\mathrm{AGG}} x_i \otimes v_i$ where the x_i are independent random variables; the annotation of each tuple in $\tilde{Q}_i$ is $\Phi = 1_K$ in case of $\bar{A}_i = \emptyset$ and a sum $\Phi = \sum x_j$ over the annotations of the tuples participating to this group, otherwise. Consider the case 32.2(a): The selection σ_ρ may compare γ_1 with a constant c to yield the annotation $\Phi_t = \Phi \cdot [\gamma_1\theta c]$. Under the assumption that the constant 0_M is not in the database, we have that for any valuation $\nu \in \Omega$ it holds that $\nu(\Phi) = 0_S$ if and only if $\nu(\gamma_1) = 0_M$, i.e., the correlation of the distributions of Φ and γ_1 is independent of the data. Starting from this observation, it follows that the probability distribution of $\Phi_t = \Phi \cdot [\gamma_1\theta c]$ can by expressed by considering the probabilities of Φ and $[\gamma_1\theta c]$ separately: For instance, for MIN aggregation and θ is $\leq$: $P_{\Phi_t}[0_S] = P_{[\gamma_1 \leq c]}[0_S]$ and $P_{\Phi_t}[1_S] = P_{[\gamma \leq c]}[1_S]$. For MIN aggregation and θ is $\geq$: $P_{\Phi_t}[0_S] = P_{\Phi}[0_S] + P_{[\gamma_1 \geq c]}[0_S]$ and $P_{\Phi_t}[1_S] = P_{\Phi}[1_S] + P_{[\gamma \geq c]}[1_S] - 1$. Similar results are obtained for MAX and SUM aggregation. Since the resulting tuples are again independent, the final projection $\pi_{\bar{A}}$ creates as annotations sums of independent expressions; furthermore, as $\gamma_1 \notin \bar{A}$, the only expressions in result tuples are the annotation expressions.

For 32.2(b), first recall that non-repeating hierarchical queries are tractable on tuple-independent relations [79]; since in addition all variables in the projection list are root variables, the resulting relation is tuple-independent. In 32.2(c), the conditional expression obtained for the result tuple compares two tractable and independent semimodule expressions. □

Secondly, we define $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$ as follows.

Definition 33 (Class $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$)**.** *The following are* RA^{ϖ}_{hie}*-queries:*

1. $Q = \pi_{\bar{A}}\varpi_{\bar{A};\gamma \leftarrow AGG(C)}\big[\sigma_\rho(Q_1 \times \cdots \times Q_n)\big]$
   if $Q_1, \ldots, Q_n \in RA^{\varpi}_{ind}$*, and* $\pi_{\bar{A}}\sigma_\rho(Q_1 \times \cdots \times Q_n)$ *is hierarchical*

2. $Q = \pi_{\bar{A}}\sigma_\rho(Q_1 \times \cdots \times Q_n)$ *if* $Q_1, \ldots, Q_n \in RA^{\varpi}_{ind}$*, and* Q *is hierarchical.*

We are now in the position to prove Theorem 56:

Proof of Theorem 56. By Lemma 57, the $\mathrm{RA}^{\varpi}_{\mathrm{ind}}$-sub-queries $Q_1, \ldots, Q_n$ can be materialised into tuple-independent relations. It follows that the queries in 33.2 are the well-known non-repeating hierarchical conjunctive queries [79] and have polynomial-time data complexity. For queries in 33.1, first consider the query $Q' = \pi_{\bar{A}}\sigma_{\rho}(Q_1 \times \cdots \times Q_n)$ which is by assumption hierarchical. It follows that, given a tuple $t \in Q'$, its annotation Φ is a read-once expression [60, 79]; moreover, Φ can be compiled into a d-tree whose size is bounded by the number of its variables. By Propositions 53 and 54, computing the probability distribution of such a d-tree – and thus the probability distribution of Φ – can be done in time polynomial in the size of Φ. By Theorem 50, the size of the input annotation is polynomial in the size of the input database, hence query evaluation is in polynomial-time.

Now consider the query Q as per Definition 33.1. Aggregation yields expressions of the form $\alpha = \sum_{\mathrm{AGG}} \varphi_i \otimes v_i$ where as above each φ_i is a product of variables; since the query is non-repeating, each product has exactly n variables, one from each relation. The aggregated data values v_i are all from the same relation and can hence be associated with the variables from that relation. The expression α can be compiled into the same d-tree as Φ, except that instead of the leaves for the variables x from the aggregated relation, it has leaves of the form $x \otimes v$. □

The following example illustrates the proof.

Example 50. Consider the database in Figure 5.1 and the query

$$Q = \varpi_{\emptyset;\alpha\leftarrow\mathrm{SUM(price)}}\big(\sigma_{\mathrm{shop}='M\&S'}(S) \bowtie PS\big)$$

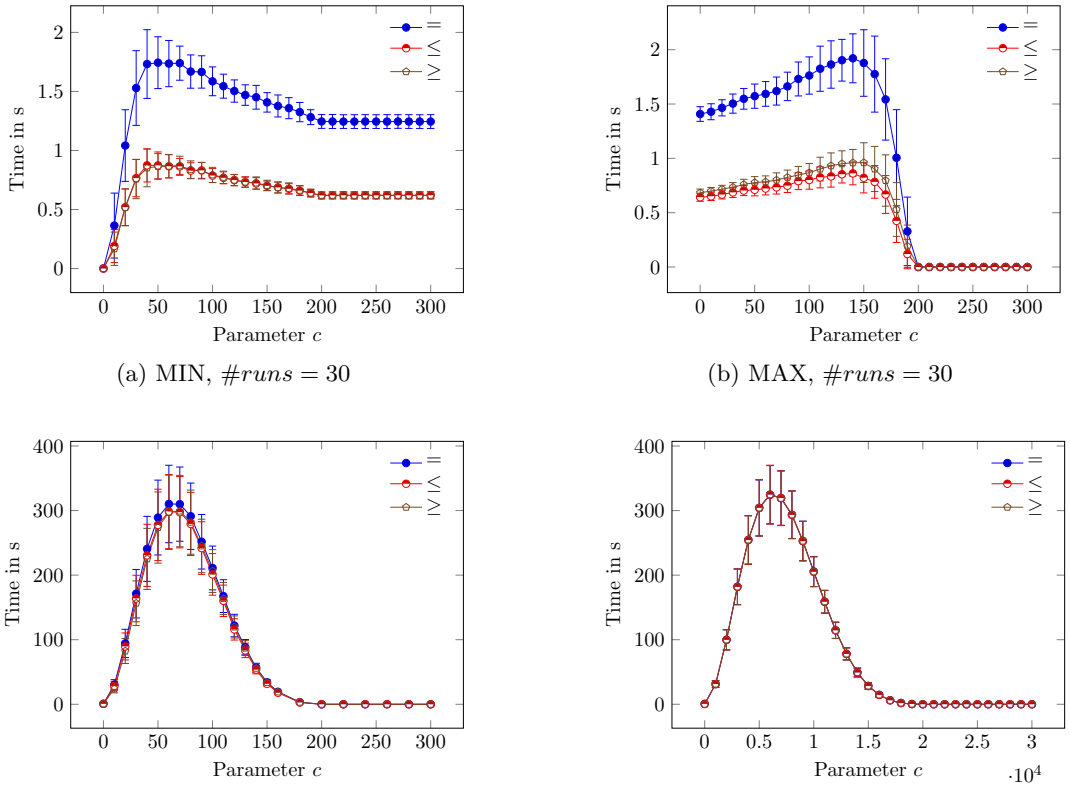
which is of type 1 in $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$. In this example, the query Q' considered in the explanation above is

$$Q' = \pi_{\emptyset}\sigma_{\mathrm{shop}='M\&S'}(S) \bowtie PS.$$

The annotation of its result tuple is $x_1y_{11} + x_1y_{12} + x_2y_{21} + x_2y_{22} + x_3y_{33} + x_3y_{34}$ which is equivalent to the read-once expression $x_1(y_{11}+y_{12})+x_2(y_{21}+y_{22})+x_3(y_{33}+y_{34})$. According to the rewriting rules $[\![\cdot]\!]$ of Algorithm 11, the annotation of the result of Q is $(x_1y_{11})\otimes 10 + (x_1y_{12})\otimes 50 + (x_2y_{21})\otimes 11 + (x_2y_{22})\otimes 60 + (x_3y_{33})\otimes 15 + (x_3y_{34})\otimes 40$. Associating the price values with the annotation variables y_{ij} of their relation, one obtains a read-once expression equivalent to the one above, except that the leaves with y_{ij} variables are now semimodule expressions $y_{ij}\otimes v_i$: $x_1(y_{11}\otimes 10 + y_{12}\otimes 50) + x_2(y_{21}\otimes 11 + y_{22}\otimes 60) + x_3(y_{33}\otimes 15 + y_{34}\otimes 40)$. □

5.5 Experimental evaluation

We have assessed our query evaluation technique in two scenarios: randomly generated expressions and aggregate queries on TPC-H data. The results give positive evidence for the feasibility of query evaluation using our technique.



(a) MIN, $\#runs = 30$

(b) MAX, $\#runs = 30$

(c) COUNT, $\#runs = 10$

(d) SUM, $\#runs = 10$

Figure 5.7: Experiment A: Varying the constant c for different aggregation monoids and comparison operators θ. $\#v$=25, L=200, R=0, $\#cl$=3, $\#l$=3, $maxv$=200.

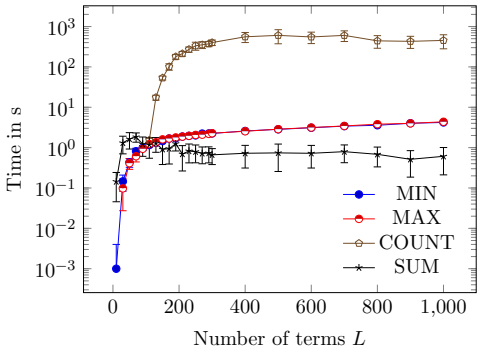
The experiments were performed on a virtual machine with Intel Xeon X5650 Quad 2.67GHz/4GB/64bit running Linux 2.6.35-25/gcc4.4.5. Our algorithms for event construction and probability computation are implemented in C. They are integrated in the SPROUT query engine, which is an extension of PostgreSQL8.3.3. We run each experiment multiple times and report average wall-clock execution times and estimated standard deviation (vertical axis in all figures) while neglecting the slowest and fastest runs.

Experiments on synthetic data

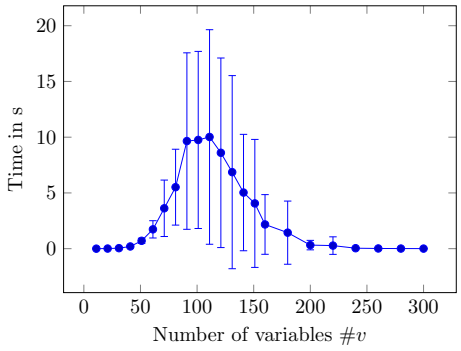
We conducted an analysis of the qualitative behaviour of our technique for randomly generated semiring expressions over Boolean random variables of the two forms

$$\left[\sum_{\text{AGGL}}^{L} \Phi_i \otimes v_i \;\theta\; \sum_{\text{AGGR}}^{R} \Psi_j \otimes w_j\right] \;,\; \left[\sum_{\text{AGGL}}^{L} \Phi_i \otimes v_i \;\theta\; c\right] \tag{5.11}$$

with the following parameters: L (R) is the number of semimodule terms on the left (right) of the comparison operator θ, AGGL and AGGR are the aggregation monoids on each side. The second form corresponds to R=0. Values v_i, w_j are from $[0, maxv]$ and the expression



(a) B: Varying the number of terms. $\#v$=25, R=0, $\#cl$=3, $\#l$=3, $maxv$=200, c=100, $\#runs$=10, θ is =.



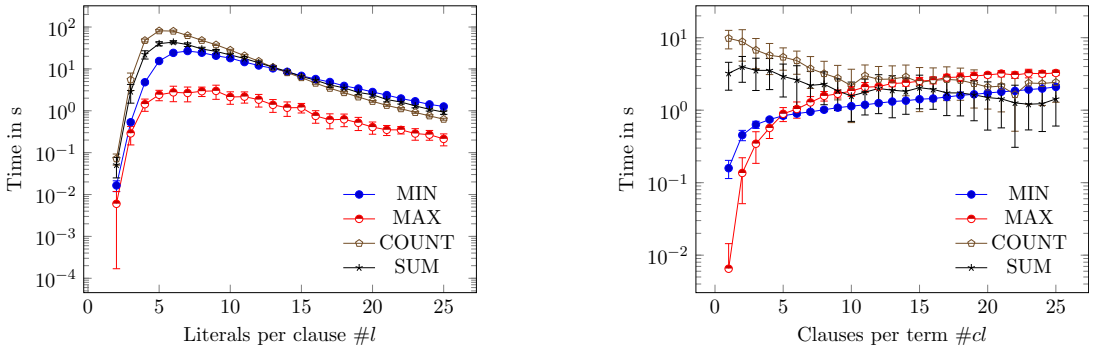
(b) C: Varying the number of variables. L=90, R=0, $\#cl$=2, $\#l$=2, $maxv$=5, c=3, θ is =, $\#runs$=40, AGGL=MIN.

Figure 5.8: Experiments B and C: Varying the number of terms and variables.

contains $\#v$ distinct variables. Each Φ_i has $\#cl$ clauses and each of them has $\#l$ positive literals. In each experiment, we randomly generate $\#runs$ different expressions of form Equation (5.11) according to the parameters specified in the respective figures.

Experiment A explores the effect of constant c on the evaluation of an expression with L=200 terms on the left side compared with c on the right side. Figure 5.7 depicts the run time for different aggregation and comparison operators. Let us analyse MIN. Values v_i are drawn from $[0, 200]$. For small c, our pruning techniques ensure that only terms with values smaller than c are considered by convolution operators in the d-tree. For the operator $\leq$, we thus need to compute the probability that any of these terms is present (i.e., its semiring expression evaluates to true); for the operator $\geq$, we compute the probability that none of these terms is present; for the operator $=$, we compute the probability that none of these terms is present and at least a term with monoid value c is present. The computation becomes slower when increasing c, and after $c = 200$, all terms need to be considered, hence the run time converges to a constant value. The behaviour of MAX mirrors that of MIN.

Evaluating COUNT for a constant c effectively amounts to computing the probability that (at most, at least, or exactly) $\binom{L}{c}$ terms are present. The binomial coefficient is trivial for $c = 1$ and $c = L = 200$ and bell-shaped in between. Since we consider three clauses per term ($\#cl = 3$), this experiment evaluates COUNT DISTINCT on top of a conjunctive query. The case of SUM is equivalent to COUNT with the horizontal axis scaled by factor $maxv/2 = 100$. This is to be expected in the limit $\#runs \to \infty$ since the aggregation values are drawn uniformly from $[0, maxv]$ with expected value $maxv/2$. We also considered Experiment A with $L = 100$ terms and the other parameters unaltered. The qualitative behaviour remained unchanged and the run time halved.



(a) $\#cl$=3 (b) $\#l$=3

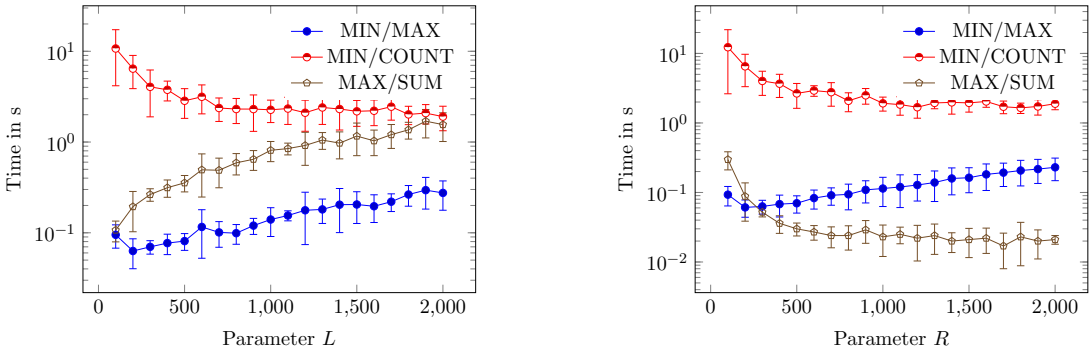
Figure 5.9: Experiment D: Varying the number of literals per clause (a) and of clauses per term (b). $\#v$=25, L=100, R=0, $maxv$=5, c=3, $\#runs$=20, θ is $\leq$.

Experiment B (Figure 5.8a) investigates the effect of varying the number of terms while keeping the number of variables constant. The initial increase in run time is due to the cost of partitioning into mutex expressions, which eventually saturates to linear growth for larger expressions once all variables have been considered for partitioning. This experiment mimics answering increasingly complex queries on a database of constant size; this produces increasingly larger expressions, yet with a constant number of variables. We repeated the above experiments (not in the figure) for the remaining comparison operators $\theta \in \{\leq, \geq\}$ and for parameters $maxv = 5, c = 3$ and obtained similar results.

Experiment C mirrors Experiment B: Fix the size of the expression and vary the number of its distinct variables. It is known from #SAT analysis that such a setup exhibits an easy/hard/easy phase transition, see Figure 5.8b. For our algorithm, the transition is understood by its limiting cases: For few variables, expressions can quickly be decomposed into mutex expressions. Conversely, expressions separate into independent sub-expressions in case of many variables since it is likely that different clauses are independent. The large standard deviation in the hard regime suggests that the run time is sensitive to the particular distribution of variables within the expression. In case θ is $\geq$ or $\leq$, the runtime improves but follows a similar pattern.

Experiment D visualises a phase transition similar to that of Experiment C (Figure 5.9). We explain for Figure 5.9a: When keeping the number of variables and terms constant while increasing the arity of clauses, the problem becomes easy for small and large clauses, and hard in between.

Experiment E investigates the behaviour of expressions with different aggregation operators on each side (Figure 5.10). We analyse the case $\sum_{\text{MAX}} \leq \sum_{\text{SUM}}$. When increasing



(a) R=150, L variable

(b) L=150, R variable

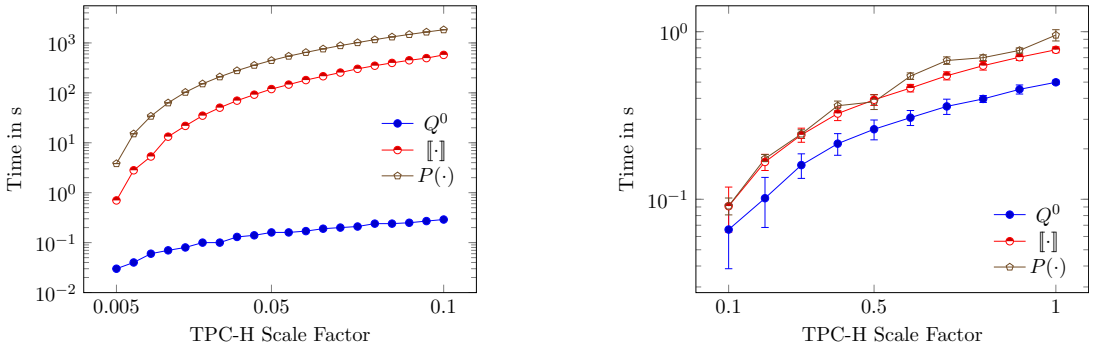
Figure 5.10: Experiment E: Expressions with different left/right aggregations. $\#v$=25, $\#cl$=2, $\#l$=2, $maxv$=200, c=100, $\#runs$=10, θ is $\leq$.

the number of terms on the left/MAX side (left figure), it becomes more likely that the maximum value on the left side is larger than the sum on the right side; hence more terms have to be compiled. When increasing the number of terms on the right/SUM side (right figure), already a few mutex decomposition steps satisfy enough clauses to make the sum larger than the maximum on the left side, i.e., the compilation is faster with increasing number of SUM terms.

Queries on TPC-H data

We consider tuple-independent TPC-H 2.14.0 databases of scales up to 1GB, and two TPC-H queries. We generated tuple-independent TPC-H databases up to scale factor 1 using TPC-H 2.7.0 and annotate tuples with distinct {0,1}-valued random variables whose probabilities are drawn uniformly at random from the interval $[0, 1]$. The query Q_1 reports the amount of business that was billed, shipped, and returned (only the COUNT aggregate is selected). The query Q_2 is a join of five relations and with a nested aggregate query, which asks for suppliers with minimum cost for an order for a given part in a given region. For each query, we compared the execution times (1) on a deterministic database (Q^0) without expression or probability computation, (2) of the computation of the expressions ($[\![\cdot]\!]$), and (3) of probability computation for the result tuples ($P(\cdot)$).

Experiment F compares the runtime for Q^0, $[\cdot]$, and $P(\cdot)$ (Figure 5.11). The overhead of expression computation and probability computation is polynomial, because the TPC-H dataset scales up the amount of tuples while keeping tuple correlations (i.e., the number of tuples directly related via joins within a group) constant. The performance difference between Q_1 and Q_2 is due to the selectivity of the queries: The size of the annotation expressions as a measure for the number of tuples contributing to Q_1's answer is 1200 times larger than the size of Q_2's annotations.



(a) TPC-H Query Q1

(b) TPC-H Query Q2

Figure 5.11: Experiment F: Queries on TPC-H Data.

5.6 Discussion

5.6.1 Related work

Aggregates are an additional source of computational complexity in the query evaluation problem: For example, already deciding whether there is a possible world in which the SUM of values of an attribute equals a given constant is NP-hard. This has motivated the investigation of restricted instances of the evaluation problem for queries with aggregates in probabilistic databases:

- A common restriction is to focus on aggregates over one probabilistic table of restricted expressiveness. There has been work on OLAP queries with aggregation operators on tables with multiple dimensions of uncertain, hierarchical attributes [15]. A framework for integrating a probabilistic semantics with ranking and top-k evaluation in deterministic databases has been presented in [77]; this work assumes the existence of an external probabilistic inference engine that evaluates probability scores for tuples. The TRIO system can compute lower and upper bounds as well as expected values for queries with aggregates over a single table [59].

- Another simplification is to consider as query answers approximate expected values rather than probability distributions over possible values. MCDB relies on Monte-Carlo sampling to compute approximate expected values for queries with aggregates over probabilistic data [46]. The input data is represented by parametrised or explicit probability distributions and queries are evaluated by first sampling from these distributions and then evaluating the query in the thus generated possible worlds. The PIP system [52] has a similar data and query model, but evaluates queries by first propagating symbolic representations of the input probability distributions, and then sampling from the thus generated symbolic expression that represent the probability distribution of query answers. The problem of efficient sampling-based evaluation in

the presence of aggregation constraints has been investigated in [92].

It is known that expected values can lead to unintuitive query answers, for instance when data values and their probabilities follow skewed and non-aligned distributions [72]. The query semantics considered in this chapter is more general than the expected value semantics in that it returns the full probability distribution over all possible answers; expected values can be straightforwardly computed from this distribution.

Tree queries with aggregates on probabilistic XML data with limited correlations have been investigated in [1]. In [54], an algebra is proposed that represents annotations and data values as rings; intuitively, this allows to represent reversible database operations and thus enables efficient incremental view maintenance in the presence of aggregations.

The algebraic annotation framework used in this chapter has been introduced in the context of databases with provenance information [5] which in turn is based on earlier results on provenance annotations for positive queries without aggregates [41, 14].

The tractability results shown in Section 5.4 generalise a known class of tractable queries of the form

$$\varpi_{\emptyset;\gamma\leftarrow\mathrm{AGG}(C)}\sigma_{\rho}(R_1 \times \cdots \times R_n)$$

in which $\pi_{\emptyset}\sigma_{\rho}(R_1 \times \cdots \times R_n)$ is a non-repeating hierarchical conjunctive query [72]; these queries are subsumed by our tractable class $\mathrm{RA}^{\varpi}_{\mathrm{hie}}$. Recall that our tractability result requires that the neutral element of the aggregation monoid does not occur in any aggregation column of the database. However, as argued in the discussion after Definition 32, for queries that only involve aggregation without grouping — such as the queries in [72] — this requirement can be dropped without jeopardising query tractability, because the annotation of the query result is always 1_K and hence the correlation of the distributions of 1_K and γ is trivial.

5.6.2 Conclusion and extensions

The pvc-tables framework proposed in this chapter is the first system that allows exact probability computation for queries with arbitrarily nested aggregation operations. The main conceptual advance is the generalisation of the annotation formalism from probability distributions defined by Boolean formulas to distributions defined by semiring and semimodule expressions. The latter allow for the concise representation of the results of queries with aggregates. In order to evaluate the semiring and semimodule expressions that annotate tuples in pvc-tables, the decomposition trees introduced in Chapter 4 for Boolean formulas — and thus Boolean probability distributions — are extended to trees representing arbitrary countable probability distributions, and hence the result of aggregation operations.

The computational complexity of evaluating queries with aggregates on probabilistic databases is notoriously high due to the entanglement of two problems that are already hard on their own: Probabilistic query evaluation is #P-hard even without aggregates, and SUM aggregation over one tuple-independent table is already NP-hard. A very promising research direction is thus the investigation of approximate probability computation techniques for queries with aggregates. Tsui discusses histogram and top-k approximations for queries with aggregates [81]; the techniques are based on pvc-tables and the decomposition techniques developed in this chapter. Approximation techniques may additionally leverage the long history of results in the field of statistics, for example regarding parametrised approximate representations of probability distributions, for example S-distributions [88].

Chapter 6

Conclusion

The query evaluation problem for probabilistic databases is to compute each possible answer to a query on a given database together with the probability that it is correct. As usual in database research, this problem can be studied from two perspectives: Firstly, the design of efficient algorithms, and secondly the understanding of its computational complexity. The two aspects are not orthogonal, but rather complementary, since a solid understanding of the complexity is essential when it comes to devising and evaluating algorithms tailored to specific instances of the evaluation problem.

Prior work on probabilistic databases has established complexity results for monotonic query languages such as conjunctive queries and unions of conjunctive queries. Algorithms and systems have been developed for the exact evaluation of restricted query languages, for example for positive queries and for queries with very limited forms of negation or aggregation. Sampling-based approximation techniques have been proposed for evaluating more expressive queries.

This thesis advances the state of the art in query processing along both aspects — complexity analysis and algorithms — by investigating the evaluation problem for query languages beyond the above-mentioned restrictions. Firstly, we propose algorithms for computing exact answers to full relational algebra queries and to positive queries with arbitrarily nested aggregation operators. Secondly, for relational algebra queries a deterministic approximation technique with error guarantees is developed, evaluated, and benchmarked against the prevailing non-deterministic, sampling-based approximation algorithms. Thirdly, we prove that the complexity dichotomy for non-repeating conjunctive queries extends to queries with negation: Each such query has either PTIME or #P-hard data complexity.

Let us loop back to the SPROUT^2 system for evaluating queries on tabular uncertain Web data (cf. Section 1). In this system, the advances in query evaluation presented in this thesis would allow users to express more powerful queries on Web data, i.e., queries beyond the simple select-project-join query from Example 2: Data can be aggregated across different data sources and negation allows for queries with universal quantification.

Despite the results of this work being self-contained, they lend themselves to possible extensions and future work, for example along the following open questions: Is it possible to lift the model-based approximations of propositional formulas to more expressive bound languages without jeopardising the low complexity of finding bounds? How can queries with aggregation operators be approximated efficiently? What is the trade-off between query and data complexity for tractable queries with negation? How does the dichotomy extend to queries with union?

A major open challenge in probabilistic database research is to bring together the research efforts and results obtained for restricted problem settings; the latter should be consolidated into a probabilistic database management system that is sufficiently generic and powerful to be considered a general-purpose platform for storing and querying uncertain, structured data. Such a system needs to understand the trade-offs between different data representations and evaluation techniques, then choose and apply those most suitable to the query and data at hand. We investigated some of these trade-offs by experimentally comparing our algorithms to those reported in prior work. We are hopeful that the results from this thesis will contribute to solving this puzzle.

Bibliography

[1] Serge Abiteboul, T.-H. Hubert Chan, Evgeny Kharlamov, Werner Nutt, and Pierre Senellart. Aggregate Queries for Discrete and Continuous Probabilistic XML. In *ICDT*, 2010.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Serge Abiteboul, Paris Kanellakis, and Gösta Grahne. On the Representation and Querying of Sets of Possible Worlds. *Theor. Comput. Sci.*, **78**(1), 1991.

[4] Serge Abiteboul, Benny Kimelfeld, Yehoshua Sagiv, and Pierre Senellart. On the expressiveness of probabilistic XML models. *VLDB J.*, 18(5):1041–1064, 2009.

[5] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.

[6] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and Simple Relational Processing of Uncertain Data. In *ICDE*, pages 983–992, 2008.

[7] Various authors. Complexity Zoo. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo:Symbols#sharpp`. Accessed: Oct 1st 2013.

[8] Pablo Barcelo, Leonid Libkin, and Miguel Romero. Efficient approximations of conjunctive queries. In *PODS*, pages 249–260, 2012.

[9] Michael Benedikt, Evgeny Kharlamov, Dan Olteanu, and Pierre Senellart. Probabilistic xml via markov chains. *PVLDB*, 3(1):770–781, 2010.

[10] Elazar Birnbaum and Eliezer Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *J. of AI Research*, **10**(6):457–477, 1999.

[11] R. K. Brayton. Factoring logic functions. *IBM J. Res. Develop.*, **31**(2):187, 1987.

[12] Randal E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, pages 688–694. IEEE Computer Society Press, 1985.

[13] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[14] Peter Buneman and Wang Chiew Tan. Provenance in databases. In *SIGMOD Conference*, 2007.

[15] Doug Burdick, Prasad M. Deshpande, T. S. Jayram, Raghu Ramakrishnan, and Shivakumar Vaithyanathan. OLAP over Uncertain and Imprecise Data. *VLDBJ*, 16(1), 2006.

[16] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[17] Graham Cormode, Minos Garofalakis, Peter Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[18] Dan Crow. Google Squared. `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/36641.pdf`.

[19] Paul Dagum, Richard M. Karp, Michael Luby, and Sheldon M. Ross. An Optimal Algorithm for Monte Carlo Estimation. *SIAM J. Comput.*, **29**(5):1484–1496, 2000.

[20] Nilesh Dalvi, Karl Schnaitter, and Dan Suciu. Computing query probability with incidence algebras. In *PODS*, pages 203–214, 2010.

[21] Nilesh Dalvi and Dan Suciu. Efficient Query Evaluation on Probabilistic Databases. In *VLDB*, pages 864–875, 2004.

[22] Nilesh Dalvi and Dan Suciu. Efficient Query Evaluation on Probabilistic Databases. *VLDB J.*, 16(4):523–544, 2007.

[23] Nilesh Dalvi and Dan Suciu. Management of Probabilistic Data: Foundations and Challenges. In *PODS*, pages 1–12, 2007.

[24] Nilesh Dalvi and Dan Suciu. The Dichotomy of Conjunctive Queries on Probabilistic Structures. In *PODS*, pages 293–302, 2007.

[25] Nilesh N. Dalvi and Dan Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *J. ACM*, 59(6):30, 2012.

[26] Adnan Darwiche and Pierre Marquis. A knowlege compilation map. *J. of AI Research*, 17:229–264, 2002.

[27] Martin Davis and Hillary Putnam. A Computing Procedure for Quantification Theory. *J. of the ACM*, **7**(3):201–215, 1960.

[28] Maximilian Dylla, Iris Miliaraki, and Martin Theobald. Top-k query processing in probabilistic databases with non-materialized views. In *ICDE*, pages 122–133, 2013.

[29] Khaled Elbassioni, Kazuhisa Makino, and Imran Rauf. On the Readability of Monotone Boolean Formulae. In *COCOON*, pages 496–505, 2009.

[30] Robert Fink, Larisa Han, and Dan Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5):490–501, 2012.

[31] Robert Fink, Andrew Hogue, Dan Olteanu, and Swaroop Rath. SPROUT^2: A squared query engine for uncertain web data. In *SIGMOD*, pages 1299–1302, 2011.

[32] Robert Fink, Jiewen Huang, and Dan Olteanu. Anytime approximation in probabilistic databases. *VLDB J.*, pages 1–26, 2013.

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

[34] Robert Fink, Dan Olteanu, and Swaroop Rath. Providing Support for Full Relational Algebra Queries in Probabilistic Databases. In *ICDE*, pages 315–326, 2011.

[35] M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[36] W. Gatterbauer, A. K. Jha, and D. Suciu. Dissociation and propagation for efficient query evaluation over probabilistic databases. TR UW-CSE-10-04-01, U. Washington, 2010.

[37] Martin Charles Golumbic, Aviad Mintz, and Udi Rotics. An Improvement on the Complexity of Factoring Read-once Boolean Functions. *Discrete Applied Mathematics*, 156(10):1633–1636, 2008.

[38] M.C. Golumbic, A. Mintza, and U. Rotics. Read-Once Functions Revisited and the Readability Number of a Boolean Function. In *Int. Colloq. on Graph Theory*, pages 357–361, 2005.

[39] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. *Handbook of Satisfiability*, chapter Model Counting. IOS Press, 2009.

[40] Erich Grädel, Yuri Gurevich, and Colin Hirsch. The Complexity of Query Reliability. In *PODS*, pages 227–234, 1998.

[41] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[42] Geoffrey Grimmett and Dominic Welsh. *Probability - An Introduction*. Oxford University Press, 1986.

[43] Rahul Gupta and Sunita Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.

[44] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. MayBMS: A Probabilistic Database Management System. In *SIGMOD*, pages 1071–1074, 2009.

[45] T. Imielinski and W. Lipski. Incomplete information in relational databases. *J. of the ACM*, **31**(4):761–791, 1984.

[46] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher M. Jermaine, and Peter J. Haas. MCDB: a Monte Carlo Approach to Managing Uncertain Data. In *SIGMOD*, pages 687–700, 2008.

[47] Abhay Kumar Jha and Dan Suciu. On the tractability of query compilation and bounded treewidth. In *ICDT*, pages 249–261, 2012.

[48] Abhay Kumar Jha and Dan Suciu. Knowledge compilation meets database theory: Compiling queries to decision diagrams. *Theory Comput. Syst.*, 52(3):403–440, 2013.

[49] David Johnson, Christos Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[50] Bhargav Kanagal, Jian Li, and Amol Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, pages 841–852, 2011.

[51] Richard M. Karp, Michael Luby, and Neal Madras. Monte-Carlo Approximation Algorithms for Enumeration Problems. *J. Algorithms*, **10**(3):429–448, 1989.

[52] Oliver Kennedy and Christoph Koch. PIP: A Database System for Great and Small Expectations. In *ICDE*, 2010.

[53] Christoph Koch. Approximating Predicates and Expressive Queries on Probabilistic Databases. In *PODS*, pages 99–108, 2008.

[54] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. In *PODS*, 2010.

[55] Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *PVLDB*, 1(1):313–325, 2008.

[56] J. Lechtenbörger, H. Shu, and Gottfried Vossen. Aggregate Queries over Conditional Tables. *JIIS*, 19(3), 2002.

[57] Jian Li and Amol Deshpande. Consensus Answers for Queries over Probabilistic Databases. In *PODS*, pages 259–268, 2009.

[58] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, 1998.

[59] Raghotham Murthy, R. Ikeda, and Jennifer Widom. Making Aggregation Work in Uncertain and Probabilistic Databases. *TKDE*, 2011.

[60] Dan Olteanu and Jiewen Huang. Using OBDDs for Efficient Query Evaluation on Probabilistic Databases. In *SUM*, pages 326–340, 2008.

[61] Dan Olteanu and Jiewen Huang. Secondary-Storage Confidence Computation for Conjunctive Queries with Inequalities. In *SIGMOD*, pages 389–402, 2009.

[62] Dan Olteanu, Jiewen Huang, and Christoph Koch. SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases. In *ICDE*, pages 640–651, 2009.

[63] Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate Confidence Computation for Probabilistic Databases. In *ICDE*, pages 145–156, 2010.

[64] Dan Olteanu, Christoph Koch, and Lyublena Antova. World-set Decompositions: Expressiveness and Efficient Algorithms. *Theor. Comp. Sci.*, 403(2-3):265–284, 2008.

[65] Dan Olteanu and Hongkai Wen. Ranking Query Answers in Probabilistic Databases: Complexity and Efficient Algorithms. In *ICDE*, pages 282–293, 2012.

[66] Joram Pe'er and Ron Y. Pinter. Minimal decomposition of boolean functions using non-repeating literal trees. In *IFIP Workshop on Logic and Architecture Synthesis*, 1995.

[67] J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983.

[68] Ralf Rantzau. Frequent itemset discovery with SQL using universal quantification. In *Database Support for Data Mining Applications*, pages 194–213, 2004.

[69] Sudhir Rao, Antonio Badia, and Dirk Van Gucht. Providing better support for a class of decision support queries. In *SIGMOD*, pages 217–227, 1996.

[70] Christopher Ré, Nilesh Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.

[71] Christopher Ré and Dan Suciu. Approximate lineage for probabilistic databases. *PVLDB*, 1(1):797–808, 2008.

[72] Christopher Ré and Dan Suciu. The Trichotomy of HAVING Queries on a Probabilistic Database. *VLDBJ*, 18(5):1091–1116, 2009.

[73] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17, 2008.

[74] Y Sagiv and M Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. of the ACM*, 27(4):633–655, 1980.

[75] Bart Selman. Knowledge compilation and theory approximation. *J. of the ACM*, 43(2):193–224, 1996.

[76] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 3(1):1068–1079, 2010.

[77] Mohamed Soliman, Ihab Ilyas, and Kevin Chang. Probabilistic Top-k and Ranking-Aggregate Queries. In *ACM TODS*, volume 33:3, 2008.

[78] Asma Souihli and Pierre Senellart. Optimizing approximations of DNF query lineage in probabilistic XML. In *ICDE*, pages 721–732, 2013.

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

[80] Luca Trevisan. A Note on Deterministic Approximate Counting for k-DNF. In *APPROX-RANDOM*, pages 417–426, 2004.

[81] Lo Po Tsui. Approximating Aggregations in Probabilistic Databases. Technical report, MSc Thesis, University of Oxford, 2013.

[82] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A New Algorithm for Generating All the Maximal Independent Sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[83] SP Vadhan. The Complexity of Counting in Sparse, Regular, and Planar Graphs. *SIAM J. Comput.*, 32(2):398–427, 2001.

[84] Leslie Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.*, **8**, 1979.

[85] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.

[86] Moshe Y. Vardi. The Complexity of Relational Query Languages. In *STOC*, pages 137–146, 1982.

[87] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[88] Eberhard O Voit and Shuiyang Yu. The s-distribution: Approximation of discrete distributions. *Biometrical journal*, 36(2):205–219, 1994.

[89] Ting-You Wang, Christopher Ré, and Dan Suciu. Implementing not exists predicates over a probabilistic database. In *QDB/MUD*, pages 73–86, 2008.

[90] Ingo Wegener. BDDs–design, analysis, complexity, and applications. *Discrete Applied Mathematics*, 138(1-2):229–251, 2004.

[91] Wei Wei and Bart Selman. A New Approach to Model Counting. In *SAT*, pages 324–339, 2005.

[92] Mohan Yang, Haixun Wang, Haiquan Chen, and J. Ku. Querying Uncertain Data with Aggregate Constraints. In *SIGMOD*, 2011.

[93] W. W. Zachary. An information flow model for conflict and fission in small groups. *J. of Anthrop. Research*, **33**:452–473, 1977.

Appendix A

Query listings and statistics

This appendix first lists the queries used in the experimental evaluation in Section 4.3, and then gives statistics regarding the size and structure of the annotation formulas of query answers.

Query Listings. The listings in Figures A.1–A.5 depict the queries used in the experimental evaluation. The :CONF macro is replaced with the appropriate confidence computation operator for the Monte-Carlo (`aconf(`ϵ, δ`)`), decomposition tree (`conf('R',` ϵ) for relative, `conf('A',` ϵ`)` for absolute approximation, and `conf('R', 0)` for exact computation), and TRACT (`conf()`) methods in order to compute probabilities for the annotations of all answer tuples.

Statistics of Annotation Formulas. The tables in Figures A.6–A.10 show the number of clauses, their arity, and number of variables of the annotation formulas generated by the TPC-H and graph queries. Figure A.11 compares the size of the query answers for the probabilistic and the deterministic case for queries with negation.

```
-- Query 2B
select :CONF
from part, supplier, partsupp, nation, region
where
  p_partkey = ps_partkey and s_suppkey = ps_suppkey
  and p_size = 15 and p_type like '%BRASS'
  and s_nationkey = n_nationkey and n_regionkey = r_regionkey
  and r_name = 'EUROPE';


-- Query 9B
select :CONF
from part, supplier, lineitem, partsupp, orders, nation
where
  s_suppkey = l_suppkey and ps_suppkey = l_suppkey
  and ps_partkey = l_partkey and p_partkey = l_partkey
  and o_orderkey = l_orderkey and s_nationkey = n_nationkey
  and p_name like '%green%';


-- Query 20B
select :CONF
from supplier, nation, partsupp, part
where
  s_suppkey = ps_suppkey and p_partkey = ps_partkey and p_name like 'forest%'
  and s_nationkey = n_nationkey and n_name = 'CANADA';


-- Query 21B
select :CONF
from supplier, lineitem, orders, nation
where
  s_suppkey = l_suppkey and o_orderkey = l_orderkey
  and o_orderstatus = 'F' and l_receiptdate > l_commitdate
  and s_nationkey = n_nationkey and n_name = 'SAUDI ARABIA';
```

Figure A.1: Intractable TPC-H queries used for the experimental evaluation.

```
-- Query 1
select l_returnflag, l_linestatus, :CONF
from lineitem
where
  l_shipdate <= date '1998-09-01'
group by
  l_returnflag, l_linestatus;


-- Query 15
select s_suppkey, s_name, s_address, s_phone, :CONF
from supplier, lineitem
where
  s_suppkey = l_suppkey and l_shipdate >= date '1991-10-10'
  and l_shipdate < date '1992-01-10'
group by
  s_suppkey, s_name, s_address, s_phone;


-- Query 1B
select :CONF
from lineitem
where
  l_shipdate <= date '1998-09-01';


-- Query 6B
select :CONF
from lineitem
where
  l_shipdate >= '1994-01-01' and l_shipdate < '1995-01-01' and l_discount >= 0.05
  and l_discount <= 0.07 and l_quantity < 24;


-- Query 16B
select :CONF
from partsupp, part
where
  part.p_partkey = partsupp.ps_partkey and p_brand <> 'Brand#45'
  and p_type like 'MEDIUM POLISHED%';


-- Query 17B
select :CONF
from lineitem, part
where
  part.p_partkey = lineitem.l_partkey and p_brand = 'Brand#23'
  and p_container = 'MED BOX';
```

Figure A.2: Tractable TPC-H queries used for the experimental evaluation.

```
-- Query IQ 1B
select :CONF
from orders, lineitem
where
  o_orderkey = l_orderkey and o_orderdate > l_shipdate - 3;


-- Query IQ 4B
select :CONF
from part, lineitem
where
  p_partkey = l_partkey and  l_extendedprice / l_quantity <= p_retailprice;


-- Query IQ 6
select s_nationkey
from supplier, customer, :CONF
where
  s_acctbal < c_acctbal and s_nationkey = c_nationkey and s_acctbal > 9000;
  group by s_nationkey;
```

Figure A.3: Tractable TPC-H queries with inequalities used for the experimental evaluation.

```
-- Triangle query G_Delta
select :CONF
from edge e1, edge e2, edge e3
where
  (e1.v = e2.u and e2.v = e3.u and e3.v = e1.u)
  or (e1.v = e2.v and e2.u = e3.u and e3.v = e1.u)


-- Path query G_P2
select :CONF
from edge e1, edge e2, no_edge e3
where
  e1.u = e3.u and e1.v = e2.u and e2.v = e3.v


-- Path query G_P3
select :CONF
from edge e1,edge e2,edge e3, no_edge ne4,no_edge ne5,no_edge ne6
where
  e1.v = e2.u and e2.v = e3.u and ne4.u = e1.u and ne4.v = e2.v
  and ne5.u = e1.u and ne5.v = e3.v and ne6.u = e2.u and ne6.v = e3.v


-- Separation query G_S
select :CONF
from edge e1, edge e2
where
  e1.v = e2.u and e1.u <> e2.v
```

Figure A.4: Graph queries G_Δ, G_{P2}, G_{P3}, G_S on random graphs.

```
-- Query N1
select p.p_partkey, p.p_name, :CONF
from part p left outer join
  (select l.l_partkey from lineitem l where l.l_quantity < 5) Q1
  on (p.p_partkey = Q1.l_partkey)
where
  Q1.l_partkey is null;

-- Query N2
select p.p_partkey, p.p_name, :CONF
from part p left outer join
  (select l.l_partkey from lineitem l where l.l_quantity < 5) Q1
  on (p.p_partkey = Q1.l_partkey)
  left outer join
  (select ps.ps_partkey from partsupp ps where ps.ps_availqty > 8500) Q2
  on (p.p_partkey = Q2.ps_partkey)
where
  Q1.l_partkey is null and Q2.ps_partkey is null ;

-- Query N3
select Q1.ps_suppkey, :CONF
from (select ps.ps_suppkey
  from partsupp ps
  group by ps.ps_suppkey ) Q1 left outer join
( select distinct Q3.ps_suppkey as ps_suppkey
  from ( select Q4.ps_suppkey as ps_suppkey, p2.p_partkey
    from part p2, (
      select distinct ps1.ps_suppkey as ps_suppkey from partsupp ps1
    ) Q4
    where
      p2.p_brand='Brand#11' and p2.p_size<10
  ) Q3
  left outer join partsupp ps
  on (ps.ps_partkey = Q3.p_partkey and ps.ps_suppkey=Q3.ps_suppkey)
  where
    ps.ps_suppkey is null
) Q2
on (Q1.ps_suppkey = Q2.ps_suppkey)
where
  Q2.ps_suppkey is null;

-- Query N4
select Q2.A1, Q2.A2, :CONF
from ( select s_suppkey as A
    from supplier T, nation n
    where T.s_nationkey=n.n_nationkey
    and (n.n_name='FRANCE')
) Q1 left outer join ( select Q3.A1, Q3.A2
  from ( select T2.ps_suppkey as A1, Q4.A as A2, T2.ps_partkey as B
    from partsupp T2, (
      select T1.s_suppkey as A
      from supplier T1, nation n1
      where
        T1.s_nationkey=n1.n_nationkey and (n1.n_name='GERMANY')
    ) Q4) Q3 left outer join partsupp T3
  on (T3.ps_partkey = Q3.B and T3.ps_suppkey=Q3.A2)
  where
    T3.ps_partkey is null
  group by Q3.A1, Q3.A2
) Q2
on (Q1.A = Q2.A1)
where
  Q2.A1 is null
group by
  Q2.A1, Q2.A2;
```

Figure A.5: TPC-H queries with negation used for the experimental evaluation.

Query	Scale Factor	Number of clauses	Arity of clauses	Number of variables
2B	0.01	5	5	18
	0.05	22	5	65
	0.1	63	5	171
	0.5	291	5	756
	1	642	5	1656
9B	0.01	3508	6	7372
	0.05	18026	6	37545
	0.1	34694	6	72419
	0.5	175219	6	365780
	1	348760	6	728042
20B	0.01	1	4	4
	0.05	12	4	33
	0.1	35	4	93
	0.5	173	4	455
	1	412	4	1050
21B	0.01	182	4	364
	0.05	3520	4	6900
	0.1	8590	4	16743
	0.5	35469	4	69449
	1	75871	4	148393

Figure A.6: Statistics of annotation formulas of intractable TPC-H queries on a TPC-H database with varying scale factors.

Query	Number of clauses	Arity of clauses	Number of variables
1	1478869*	1	1478869*
15	1*	2	2*
1B	5914746	1	5914746
6B	114160	1	114160
16B	25824	2	32280
17B	6088	2	6292

*: Typical number per result tuple.

Figure A.7: Statistics of annotation formulas of hierarchical TPC-H queries on a TPC-H database with scale factor 1. For the non-Boolean queries 1 and 15, the table lists typical numbers of clauses and variables for a single result tuple; query 1 has 4 result tuples and a total of 5914747 clauses, query 15 has 676 result tuples and a total of 704 clauses.

Query	Number of clauses	Arity of clauses	Number of variables
IQ 1B	99368	2	195595
IQ 4B	6001212	2	6201212
IQ 6	10315*	2	558*

*: Typical number per result tuple.

Figure A.8: Statistics of annotation formulas of TPC-H queries with inequalities on a TPC-H database with scale factor 1. For the non-Boolean query IQ 6, the table lists typical numbers of clauses and variables for a single result tuple; query IQ 6 has 25 result tuples and a total of 256188 clauses.

Number of nodes	Number of clauses		Arity of clauses		Number of variables	
	G_Δ	G_{P2}	G_Δ	G_{P2}	G_Δ	G_{P2}
6	20	120	3	3	15	15
7	35	210	3	3	21	21
8	56	336	3	3	28	28
9	84	504	3	3	36	36
10	120	720	3	3	45	45
11	165	990	3	3	55	55
12	220	1320	3	3	66	66
13	286	1716	3	3	78	78
14	364	2184	3	3	91	91
15	455	2730	3	3	105	105
16	560	3360	3	3	120	120
17	680	4080	3	3	136	136
18	816	4896	3	3	153	153
19	969	5814	3	3	171	171
20	1140	6840	3	3	190	190
21	1330	7980	3	3	210	210
22	1540	9240	3	3	231	231
23	1771	10626	3	3	253	253
24	2024	12144	3	3	276	276
25	2300	13800	3	3	300	300
26	2600	15600	3	3	325	325
27	2925	17550	3	3	351	351
28	3276	19656	3	3	378	378
29	3654	21924	3	3	406	406
30	4060	24360	3	3	435	435
31	4495	26970	3	3	465	465
32	4960	29760	3	3	496	496
33	5456	32736	3	3	528	528
34	5984	35904	3	3	561	561
35	6545	39270	3	3	595	595
36	7140	42840	3	3	630	630
37	7770	46620	3	3	666	666
38	8436	50616	3	3	703	703
39	9139	54834	3	3	741	741
40	9880	59280	3	3	780	780

Figure A.9: Statistics of annotation formulas of the triangle query G_Δ and the graph query G_{P2} on random graphs.

Dataset	Query	Number of clauses	Arity of clauses	Number of variables
Dolphins	G_Δ	1140	3	242
	G_{P2}	536	3	242
	G_{P3}	564	6	130
	G_S	1846	2	318
Karate	G_Δ	540	3	134
	G_{P2}	240	3	134
	G_{P3}	240	6	50
	G_S	1056	2	156

Figure A.10: Statistics of annotation formulas of the triangle query G_Δ, path queries G_{P2}, G_{P3}, and separation query G_S on "Karate" and "Dolphin" social network graphs.

TPC-H Scale	Answer size and proportion to the size of the non-probabilistic query answer							
	N1	N1-Ratio	N2	N2-Ratio	N3	N3-Ratio	N4	N4-Ratio
0.005	1000	11	1000	21	50	∞	0	∞
0.01	2000	12	2000	29	100	∞	10	∞
0.05	10000	11	10000	21	500	∞	513	∞
0.1	20000	11	20000	21	1000	∞	1750	∞
0.5	100000	11	100000	21	5000	∞	42188	∞
1	200000	11	200000	21	10000	∞	159192	∞

(a)

TPC-H Scale	Average annotation size in bytes			
	N1	N2	N3	N4
0.005	24	30	431	0
0.01	24	31	446	834
0.05	27	34	876	972
0.1	28	35	1373	971
0.5	30	38	4890	1127
1	31	39	10002	1128

(b)

Figure A.11: Statistics of annotation formulas of TPC-H queries with negation for varying scale factors.

Appendix B

Social network databases

Karate club data

The following is a list of directed edges ⟨from, to, p⟩ between the nodes *from* and *to* and probability p in the *Karate club* database used in the experiments in Section 4.3.

⟨1, 32, 0.941176⟩, ⟨1, 22, 0.941176⟩, ⟨1, 20, 0.941176⟩, ⟨1, 18, 0.941176⟩, ⟨1, 14, 0.941176⟩, ⟨1, 13, 0.941176⟩, ⟨1, 12, 0.941176⟩, ⟨1, 11, 0.941176⟩, ⟨1, 9, 0.941176⟩, ⟨1, 8, 0.941176⟩, ⟨1, 7, 0.941176⟩, ⟨1, 6, 0.941176⟩, ⟨1, 5, 0.941176⟩, ⟨1, 4, 0.941176⟩, ⟨1, 3, 0.941176⟩, ⟨1, 2, 0.941176⟩, ⟨10, 34, 0.117647⟩, ⟨10, 3, 0.117647⟩, ⟨11, 6, 0.176471⟩, ⟨11, 5, 0.176471⟩, ⟨11, 1, 0.176471⟩, ⟨12, 1, 0.0588235⟩, ⟨13, 4, 0.117647⟩, ⟨13, 1, 0.117647⟩, ⟨14, 34, 0.294118⟩, ⟨14, 4, 0.294118⟩, ⟨14, 3, 0.294118⟩, ⟨14, 2, 0.294118⟩, ⟨14, 1, 0.294118⟩, ⟨15, 34, 0.117647⟩, ⟨15, 33, 0.117647⟩, ⟨16, 34, 0.117647⟩, ⟨16, 33, 0.117647⟩, ⟨17, 7, 0.117647⟩, ⟨17, 6, 0.117647⟩, ⟨18, 2, 0.117647⟩, ⟨18, 1, 0.117647⟩, ⟨19, 34, 0.117647⟩, ⟨19, 33, 0.117647⟩, ⟨2, 31, 0.529412⟩, ⟨2, 22, 0.529412⟩, ⟨2, 20, 0.529412⟩, ⟨2, 18, 0.529412⟩, ⟨2, 14, 0.529412⟩, ⟨2, 8, 0.529412⟩, ⟨2, 4, 0.529412⟩, ⟨2, 3, 0.529412⟩, ⟨2, 1, 0.529412⟩, ⟨20, 34, 0.176471⟩, ⟨20, 2, 0.176471⟩, ⟨20, 1, 0.176471⟩, ⟨21, 34, 0.117647⟩, ⟨21, 33, 0.117647⟩, ⟨22, 2, 0.117647⟩, ⟨22, 1, 0.117647⟩, ⟨23, 34, 0.117647⟩, ⟨23, 33, 0.117647⟩, ⟨24, 34, 0.294118⟩, ⟨24, 33, 0.294118⟩, ⟨24, 30, 0.294118⟩, ⟨24, 28, 0.294118⟩, ⟨24, 26, 0.294118⟩, ⟨25, 32, 0.176471⟩, ⟨25, 28, 0.176471⟩, ⟨25, 26, 0.176471⟩, ⟨26, 32, 0.176471⟩, ⟨26, 25, 0.176471⟩, ⟨26, 24, 0.176471⟩, ⟨27, 34, 0.117647⟩, ⟨27, 30, 0.117647⟩, ⟨28, 34, 0.235294⟩, ⟨28, 25, 0.235294⟩, ⟨28, 24, 0.235294⟩, ⟨28, 3, 0.235294⟩, ⟨29, 34, 0.176471⟩, ⟨29, 32, 0.176471⟩, ⟨29, 3, 0.176471⟩, ⟨3, 33, 0.588235⟩, ⟨3, 29, 0.588235⟩, ⟨3, 28, 0.588235⟩, ⟨3, 14, 0.588235⟩, ⟨3, 10, 0.588235⟩, ⟨3, 9, 0.588235⟩, ⟨3, 8, 0.588235⟩, ⟨3, 4, 0.588235⟩, ⟨3, 2, 0.588235⟩, ⟨3, 1, 0.588235⟩, ⟨30, 34, 0.235294⟩, ⟨30, 33, 0.235294⟩, ⟨30, 27, 0.235294⟩, ⟨30, 24, 0.235294⟩, ⟨31, 34, 0.235294⟩, ⟨31, 33, 0.235294⟩, ⟨31, 9, 0.235294⟩, ⟨31, 2, 0.235294⟩, ⟨32, 34, 0.352941⟩, ⟨32, 33, 0.352941⟩, ⟨32, 29, 0.352941⟩, ⟨32, 26, 0.352941⟩, ⟨32, 25, 0.352941⟩, ⟨32, 1, 0.352941⟩, ⟨33, 34, 0.705882⟩, ⟨33, 32, 0.705882⟩, ⟨33, 31, 0.705882⟩, ⟨33, 30, 0.705882⟩, ⟨33, 24, 0.705882⟩, ⟨33, 23, 0.705882⟩, ⟨33, 21, 0.705882⟩, ⟨33, 19, 0.705882⟩, ⟨33, 16, 0.705882⟩, ⟨33, 15, 0.705882⟩, ⟨33, 9, 0.705882⟩, ⟨33, 3, 0.705882⟩, ⟨34, 33, 1⟩, ⟨34, 32, 1⟩, ⟨34, 31, 1⟩, ⟨34, 30, 1⟩, ⟨34, 29, 1⟩, ⟨34, 28, 1⟩, ⟨34, 27, 1⟩, ⟨34, 24, 1⟩, ⟨34, 23, 1⟩, ⟨34, 21, 1⟩, ⟨34, 20, 1⟩, ⟨34, 19, 1⟩, ⟨34, 16, 1⟩, ⟨34, 15, 1⟩, ⟨34, 14, 1⟩, ⟨34, 10, 1⟩, ⟨34, 9, 1⟩, ⟨4, 14, 0.352941⟩, ⟨4, 13, 0.352941⟩, ⟨4, 8, 0.352941⟩, ⟨4, 3, 0.352941⟩, ⟨4, 2, 0.352941⟩, ⟨4, 1, 0.352941⟩, ⟨5, 11, 0.176471⟩, ⟨5, 7, 0.176471⟩, ⟨5, 1, 0.176471⟩, ⟨6, 17, 0.235294⟩, ⟨6, 11, 0.235294⟩, ⟨6, 7, 0.235294⟩, ⟨6, 1, 0.235294⟩, ⟨7, 17, 0.235294⟩, ⟨7, 6, 0.235294⟩, ⟨7, 5, 0.235294⟩, ⟨7, 1, 0.235294⟩, ⟨8, 4, 0.235294⟩, ⟨8, 3, 0.235294⟩, ⟨8, 2, 0.235294⟩, ⟨8, 1, 0.235294⟩, ⟨9, 34, 0.294118⟩, ⟨9, 33, 0.294118⟩, ⟨9, 31, 0.294118⟩, ⟨9, 3, 0.294118⟩, ⟨9, 1, 0.294118⟩

Dolphins data

The following is a list of directed edges ⟨from, to, p⟩ between the nodes *from* and *to* and probability p in the *Karate club* database used in the experiments in Section 4.3.

⟨Fish, Beak, 0.416667⟩, ⟨Grin, Beak, 1⟩, ⟨Haecksel, Beak, 0.583333⟩, ⟨SN9, Beak, 0.666667⟩, ⟨SN96, Beak, 0.5⟩, ⟨TR77, Beak, 0.5⟩, ⟨Jet, Beescratch, 0.75⟩, ⟨Knit, Beescratch, 0.333333⟩, ⟨Notch, Beescratch, 0.25⟩, ⟨Number1, Beescratch, 0.416667⟩, ⟨Oscar, Beescratch, 0.416667⟩, ⟨SN100, Beescratch, 0.583333⟩, ⟨SN90, Beescratch, 0.416667⟩, ⟨Upbang, Beescratch, 0.583333⟩, ⟨Fish, Bumper, 0.416667⟩, ⟨SN96, Bumper, 0.5⟩, ⟨Thumper, Bumper, 0.333333⟩, ⟨Zipfel, Bumper, 0.25⟩, ⟨Double, CCL, 0.5⟩, ⟨Grin, CCL, 1⟩, ⟨Zap, CCL, 0.416667⟩, ⟨Trigger, Cross, 0.833333⟩, ⟨Feather, DN16, 0.583333⟩, ⟨Gallatin, DN16, 0.666667⟩, ⟨Wave, DN16, 0.166667⟩, ⟨Web, DN16, 0.75⟩, ⟨Feather, DN21, 0.583333⟩, ⟨Gallatin, DN21, 0.666667⟩, ⟨Jet, DN21, 0.75⟩, ⟨Upbang, DN21, 0.583333⟩, ⟨Wave, DN21, 0.166667⟩, ⟨Web, DN21, 0.75⟩, ⟨Knit, DN63, 0.333333⟩, ⟨Number1, DN63, 0.416667⟩, ⟨PL, DN63, 0.416667⟩, ⟨SN9, DN63, 0.666667⟩, ⟨Upbang, DN63, 0.583333⟩, ⟨CCL, Double, 0.25⟩, ⟨Kringel, Double, 0.75⟩, ⟨Oscar, Double, 0.416667⟩, ⟨SN4, Double, 0.916667⟩, ⟨Topless, Double, 0.916667⟩, ⟨Zap, Double, 0.416667⟩, ⟨DN16, Feather, 0.333333⟩, ⟨DN21, Feather, 0.5⟩, ⟨Gallatin, Feather, 0.666667⟩, ⟨Jet, Feather, 0.75⟩, ⟨Ripplefluke, Feather, 0.25⟩, ⟨SN90, Feather, 0.416667⟩, ⟨Web, Feather, 0.75⟩, ⟨Beak, Fish, 0.5⟩, ⟨Bumper, Fish, 0.333333⟩, ⟨Patchback, Fish, 0.75⟩, ⟨SN96, Fish, 0.5⟩, ⟨TR77, Fish, 0.5⟩, ⟨Trigger, Five, 0.833333⟩, ⟨Scabs, Fork, 0.833333⟩, ⟨DN16, Gallatin, 0.333333⟩, ⟨DN21, Gallatin, 0.5⟩, ⟨Feather, Gallatin, 0.583333⟩, ⟨Jet, Gallatin, 0.75⟩, ⟨Ripplefluke, Gallatin, 0.25⟩, ⟨SN90, Gallatin, 0.416667⟩, ⟨Upbang, Gallatin, 0.583333⟩, ⟨Web, Gallatin, 0.75⟩, ⟨Beak, Grin, 0.5⟩, ⟨CCL, Grin, 0.25⟩, ⟨Hook, Grin, 0.5⟩, ⟨MN83, Grin, 0.5⟩, ⟨Scabs, Grin, 0.833333⟩, ⟨Shmuddel, Grin, 0.416667⟩, ⟨SN4, Grin, 0.916667⟩, ⟨SN63, Grin, 0.666667⟩, ⟨SN9, Grin, 0.666667⟩, ⟨Stripes, Grin, 0.583333⟩, ⟨TR99, Grin, 0.583333⟩, ⟨TSN103, Grin, 0.333333⟩, ⟨Beak, Haecksel, 0.5⟩, ⟨Jonah, Haecksel, 0.583333⟩, ⟨MN83, Haecksel, 0.5⟩, ⟨SN9, Haecksel, 0.666667⟩, ⟨Topless, Haecksel, 0.916667⟩, ⟨Vau, Haecksel, 0.166667⟩, ⟨Zap, Haecksel, 0.416667⟩, ⟨Grin, Hook, 1⟩, ⟨Kringel, Hook, 0.75⟩, ⟨Scabs, Hook, 0.833333⟩, ⟨SN4, Hook, 0.916667⟩, ⟨SN63, Hook, 0.666667⟩, ⟨TR99, Hook, 0.583333⟩, ⟨Beescratch, Jet, 0.666667⟩, ⟨DN21, Jet, 0.5⟩, ⟨Feather, Jet, 0.583333⟩, ⟨Gallatin, Jet, 0.666667⟩, ⟨MN23, Jet, 0.0833333⟩, ⟨Mus, Jet, 0.25⟩, ⟨Number1, Jet, 0.416667⟩, ⟨Quasi, Jet, 0.0833333⟩, ⟨Web, Jet, 0.75⟩, ⟨Haecksel, Jonah, 0.583333⟩, ⟨Kringel, Jonah, 0.75⟩, ⟨MN105, Jonah, 0.5⟩, ⟨MN83, Jonah, 0.5⟩, ⟨Patchback, Jonah, 0.75⟩, ⟨Topless, Jonah, 0.916667⟩, ⟨Trigger, Jonah, 0.833333⟩, ⟨Beescratch, Knit, 0.666667⟩, ⟨DN63, Knit, 0.416667⟩, ⟨PL, Knit, 0.416667⟩, ⟨Upbang, Knit, 0.583333⟩, ⟨Double, Kringel, 0.5⟩, ⟨Hook, Kringel, 0.5⟩, ⟨Jonah, Kringel, 0.583333⟩, ⟨Oscar, Kringel, 0.416667⟩, ⟨SN100, Kringel, 0.583333⟩, ⟨SN63, Kringel, 0.666667⟩, ⟨Thumper, Kringel, 0.333333⟩, ⟨TR77, Kringel, 0.5⟩, ⟨TR99, Kringel, 0.583333⟩, ⟨Jonah, MN105, 0.583333⟩, ⟨Patchback, MN105, 0.75⟩, ⟨Scabs, MN105, 0.833333⟩, ⟨SN4, MN105, 0.916667⟩, ⟨Topless, MN105, 0.916667⟩, ⟨Trigger, MN105, 0.833333⟩, ⟨Jet, MN23, 0.75⟩, ⟨SN100, MN60, 0.583333⟩, ⟨Topless, MN60, 0.916667⟩, ⟨Trigger, MN60, 0.833333⟩, ⟨Grin, MN83, 1⟩, ⟨Haecksel, MN83, 0.583333⟩, ⟨Jonah, MN83, 0.583333⟩, ⟨Patchback, MN83, 0.75⟩, ⟨Topless, MN83, 0.916667⟩, ⟨Trigger, MN83, 0.833333⟩, ⟨Jet, Mus, 0.75⟩, ⟨Notch, Mus, 0.25⟩, ⟨Number1, Mus, 0.416667⟩, ⟨Beescratch, Notch, 0.666667⟩, ⟨Mus, Notch, 0.25⟩, ⟨Number1, Notch, 0.416667⟩, ⟨Beescratch, Number1, 0.666667⟩, ⟨DN63, Number1, 0.416667⟩, ⟨Jet, Number1, 0.75⟩, ⟨Mus, Number1, 0.25⟩, ⟨Notch, Number1, 0.25⟩, ⟨Beescratch, Oscar, 0.666667⟩, ⟨Double, Oscar, 0.5⟩, ⟨Kringel, Oscar, 0.75⟩, ⟨PL, Oscar, 0.416667⟩, ⟨TR77, Oscar, 0.5⟩, ⟨Fish, Patchback, 0.416667⟩, ⟨Jonah, Patchback, 0.583333⟩, ⟨MN105, Patchback, 0.5⟩, ⟨MN83, Patchback, 0.5⟩, ⟨SMN5, Patchback, 0.0833333⟩, ⟨Stripes,

Patchback, 0.583333⟩, ⟨Topless, Patchback, 0.916667⟩, ⟨Trigger, Patchback, 0.833333⟩, ⟨TSN103, Patchback, 0.333333⟩, ⟨DN63, PL, 0.416667⟩, ⟨Knit, PL, 0.333333⟩, ⟨Oscar, PL, 0.416667⟩, ⟨SN96, PL, 0.5⟩, ⟨TR77, PL, 0.5⟩, ⟨Jet, Quasi, 0.75⟩, ⟨Feather, Ripplefluke, 0.583333⟩, ⟨Gallatin, Ripplefluke, 0.666667⟩, ⟨Zig, Ripplefluke, 0.0833333⟩, ⟨Fork, Scabs, 0.0833333⟩, ⟨Grin, Scabs, 1⟩, ⟨Hook, Scabs, 0.5⟩, ⟨MN105, Scabs, 0.5⟩, ⟨Shmuddel, Scabs, 0.416667⟩, ⟨SN4, Scabs, 0.916667⟩, ⟨SN63, Scabs, 0.666667⟩, ⟨SN9, Scabs, 0.666667⟩, ⟨Stripes, Scabs, 0.583333⟩, ⟨TR99, Scabs, 0.583333⟩, ⟨Grin, Shmuddel, 1⟩, ⟨Scabs, Shmuddel, 0.833333⟩, ⟨SN4, Shmuddel, 0.916667⟩, ⟨Thumper, Shmuddel, 0.333333⟩, ⟨TR88, Shmuddel, 0.166667⟩, ⟨Patchback, SMN5, 0.75⟩, ⟨Beescratch, SN100, 0.666667⟩, ⟨Kringel, SN100, 0.75⟩, ⟨MN60, SN100, 0.25⟩, ⟨SN4, SN100, 0.916667⟩, ⟨SN89, SN100, 0.166667⟩, ⟨SN9, SN100, 0.666667⟩, ⟨Zap, SN100, 0.416667⟩, ⟨Double, SN4, 0.5⟩, ⟨Grin, SN4, 1⟩, ⟨Hook, SN4, 0.5⟩, ⟨MN105, SN4, 0.5⟩, ⟨Scabs, SN4, 0.833333⟩, ⟨Shmuddel, SN4, 0.416667⟩, ⟨SN100, SN4, 0.583333⟩, ⟨SN9, SN4, 0.666667⟩, ⟨Stripes, SN4, 0.583333⟩, ⟨Topless, SN4, 0.916667⟩, ⟨Zipfel, SN4, 0.25⟩, ⟨Grin, SN63, 1⟩, ⟨Hook, SN63, 0.5⟩, ⟨Kringel, SN63, 0.75⟩, ⟨Scabs, SN63, 0.833333⟩, ⟨Stripes, SN63, 0.583333⟩, ⟨Thumper, SN63, 0.333333⟩, ⟨TSN103, SN63, 0.333333⟩, ⟨Whitetip, SN63, 0.0833333⟩, ⟨SN100, SN89, 0.583333⟩, ⟨Web, SN89, 0.75⟩, ⟨Beak, SN9, 0.5⟩, ⟨DN63, SN9, 0.416667⟩, ⟨Grin, SN9, 1⟩, ⟨Haecksel, SN9, 0.583333⟩, ⟨Scabs, SN9, 0.833333⟩, ⟨SN100, SN9, 0.583333⟩, ⟨SN4, SN9, 0.916667⟩, ⟨TSN103, SN9, 0.333333⟩, ⟨Beescratch, SN90, 0.666667⟩, ⟨Feather, SN90, 0.583333⟩, ⟨Gallatin, SN90, 0.666667⟩, ⟨Upbang, SN90, 0.583333⟩, ⟨Web, SN90, 0.75⟩, ⟨Beak, SN96, 0.5⟩, ⟨Bumper, SN96, 0.333333⟩, ⟨Fish, SN96, 0.416667⟩, ⟨PL, SN96, 0.416667⟩, ⟨TR77, SN96, 0.5⟩, ⟨TR99, SN96, 0.583333⟩, ⟨Grin, Stripes, 1⟩, ⟨Patchback, Stripes, 0.75⟩, ⟨Scabs, Stripes, 0.833333⟩, ⟨SN4, Stripes, 0.916667⟩, ⟨SN63, Stripes, 0.666667⟩, ⟨TR120, Stripes, 0.166667⟩, ⟨TSN83, Stripes, 0.166667⟩, ⟨Bumper, Thumper, 0.333333⟩, ⟨Kringel, Thumper, 0.75⟩, ⟨Shmuddel, Thumper, 0.416667⟩, ⟨SN63, Thumper, 0.666667⟩, ⟨Double, Topless, 0.5⟩, ⟨Haecksel, Topless, 0.583333⟩, ⟨Jonah, Topless, 0.583333⟩, ⟨MN105, Topless, 0.5⟩, ⟨MN60, Topless, 0.25⟩, ⟨MN83, Topless, 0.5⟩, ⟨Patchback, Topless, 0.75⟩, ⟨SN4, Topless, 0.916667⟩, ⟨TR99, Topless, 0.583333⟩, ⟨Trigger, Topless, 0.833333⟩, ⟨Zap, Topless, 0.416667⟩, ⟨Stripes, TR120, 0.583333⟩, ⟨TR88, TR120, 0.166667⟩, ⟨Beak, TR77, 0.5⟩, ⟨Fish, TR77, 0.416667⟩, ⟨Kringel, TR77, 0.75⟩, ⟨Oscar, TR77, 0.416667⟩, ⟨PL, TR77, 0.416667⟩, ⟨SN96, TR77, 0.5⟩, ⟨Web, TR82, 0.75⟩, ⟨Shmuddel, TR88, 0.416667⟩, ⟨TR120, TR88, 0.166667⟩, ⟨Grin, TR99, 1⟩, ⟨Hook, TR99, 0.5⟩, ⟨Kringel, TR99, 0.75⟩, ⟨Scabs, TR99, 0.833333⟩, ⟨SN96, TR99, 0.5⟩, ⟨Topless, TR99, 0.916667⟩, ⟨Trigger, TR99, 0.833333⟩, ⟨Cross, Trigger, 0.0833333⟩, ⟨Five, Trigger, 0.0833333⟩, ⟨Jonah, Trigger, 0.583333⟩, ⟨MN105, Trigger, 0.5⟩, ⟨MN60, Trigger, 0.25⟩, ⟨MN83, Trigger, 0.5⟩, ⟨Patchback, Trigger, 0.75⟩, ⟨Topless, Trigger, 0.916667⟩, ⟨TR99, Trigger, 0.583333⟩, ⟨Vau, Trigger, 0.166667⟩, ⟨Grin, TSN103, 1⟩, ⟨Patchback, TSN103, 0.75⟩, ⟨SN63, TSN103, 0.666667⟩, ⟨SN9, TSN103, 0.666667⟩, ⟨Stripes, TSN83, 0.583333⟩, ⟨Zipfel, TSN83, 0.25⟩, ⟨Beescratch, Upbang, 0.666667⟩, ⟨DN21, Upbang, 0.5⟩, ⟨DN63, Upbang, 0.416667⟩, ⟨Gallatin, Upbang, 0.666667⟩, ⟨Knit, Upbang, 0.333333⟩, ⟨SN90, Upbang, 0.416667⟩, ⟨Web, Upbang, 0.75⟩, ⟨Haecksel, Vau, 0.583333⟩, ⟨Trigger, Vau, 0.833333⟩, ⟨DN16, Wave, 0.333333⟩, ⟨DN21, Wave, 0.5⟩, ⟨DN16, Web, 0.333333⟩, ⟨DN21, Web, 0.5⟩, ⟨Feather, Web, 0.583333⟩, ⟨Gallatin, Web, 0.666667⟩, ⟨Jet, Web, 0.75⟩, ⟨SN89, Web, 0.166667⟩, ⟨SN90, Web, 0.416667⟩, ⟨TR82, Web, 0.0833333⟩, ⟨Upbang, Web, 0.583333⟩, ⟨SN63, Whitetip, 0.666667⟩, ⟨CCL, Zap, 0.25⟩, ⟨Double, Zap, 0.5⟩, ⟨Haecksel, Zap, 0.583333⟩, ⟨SN100, Zap, 0.583333⟩, ⟨Topless, Zap, 0.916667⟩, ⟨Ripplefluke, Zig, 0.25⟩, ⟨Bumper, Zipfel, 0.333333⟩, ⟨SN4, Zipfel, 0.916667⟩, ⟨TSN83, Zipfel, 0.166667⟩