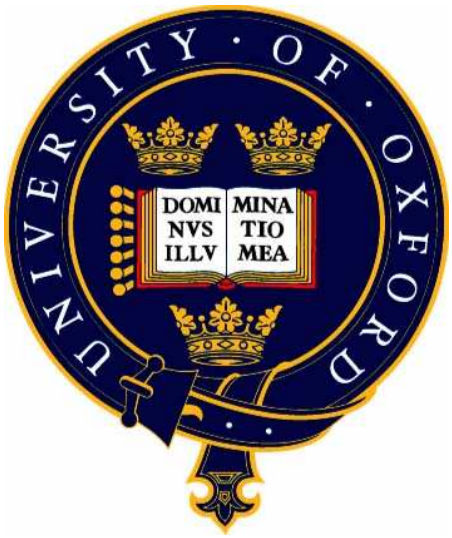
Design and Implementation of the SPROUT Query Engine for Probabilistic Databases

Jiewen Huang
Oriel College
Supervisor: Dr. Dan Olteanu



Submitted in partial fulfilment of the degree of
Master of Science by Research in Computer Science
Computing Laboratory
University of Oxford

August 2009

Acknowledgments

I would like to thank a number of people for their assistance and support.

First of all, I cannot overstate my indebtedness to my supervisor, Dan Olteanu. Since the conception of the projects, Dan has been a source of constant encouragement, sound advice, great company, and lots of good ideas. More generally, he has taught me how to become a database researcher. Without his guidance, SPROUT and MayBMS would not have been possible.

Aside from my supervisor, I must also thank Christoph Koch. He overcame several obstacles to provide me with generous financial support. The work on MayBMS was supported by a one-year scholarship from Cornell University. What is more, he initially proposed the implementation of MayBMS and guided me during its execution.

I also would like to express my gratitude to those who helped in numerous other ways, from proof-reading to providing much-needed feedback. In this regard, I would like to thank Sebastian Ordyniak, Margarita Satraki, Hao Wu and Haoxian Zhao.

Finally, no acknowledgment section would be complete without mentioning my family, but particularly my parents, for whose constant support in this and all my endeavors I am singularly fortunate.

Abstract

SPROUT is an extension of the PostgreSQL backend with state-of-the-art scalable techniques for confidence computation. SPROUT is publicly available at sourceforge.net since March 2009 and was demonstrated at ACM SIGMOD 2009.

This thesis introduces two novel techniques for scalable confidence computation. The first technique generalizes the results of my previous papers (published in IEEE ICDE 2009 and SUM 2008) to a novel class of tractable conjunctive queries without self-joins and with inequalities ($<$) on tuple-independent probabilistic databases. The problem of syntactically characterizing tractable conjunctive queries with inequalities is fundamental to probabilistic databases and was recently stated open. A confidence computation technique is given based on efficient compilation of the lineage of the query answer into Ordered Binary Decision Diagrams (OBDDs). For practical purposes, a secondary-storage variant is developed that does not need to materialize the OBDD, but computes, in one scan over the lineage, the probabilities of OBDD fragments and combines them on the fly. The results on this first query processing technique appeared in an ACM SIGMOD 2009 research paper and passed the SIGMOD repeatability and workability evaluation (RWE).

The second technique deals with arbitrary positive relational algebra queries on representation systems for probabilistic data beyond tuple-independence. It is based on a novel deterministic approximation algorithm, which uses a data structure called decomposition tree, and supports both absolute and relative approximation for confidence computation. An important property of the algorithm is that it naturally captures all known tractable conjunctive queries without self-joins on tuple-independent probabilistic databases: In this case, the algorithm requires time polynomial in the input size for exact computation. These contributions are part of an IEEE ICDE 2010 submission.

Both techniques were implemented in SPROUT and, as shown by an extensive experimental effort on various kinds of probabilistic databases, they consistently outperform prior state-of-art techniques by several orders of magnitude.

As part of my work, I have also implemented MayBMS, a state-of-the-art probabilistic database management system which leverages the strengths of previous database research. MayBMS stores probabilistic data in U-relational databases, a succinct and complete representation system for large sets of possible worlds. Queries are expressed in an extension of SQL with

specialized constructs for probability computation and what-if analysis. As the query engine, MayBMS uses SPROUT. MayBMS has been released and is available for download at http://maybms.sourceforge.net. The work on MayBMS was supported by a one-year scholarship from Cornell University.

Contents

1 Introduction

Queries on probabilistic databases have numerous applications in information retrieval [16], data cleaning [5], sensor data management [8, 15, 28], business decision support [21], crime fighting [6], and computational science [10, 1].

The computation of the probability of an event occurring in a query result – usually the occurrence of a particular tuple – is a core operation in discrete probabilistic databases: It is the key operation that probabilistic databases *add* over more traditional databases. In short, the confidence in a tuple t being in the result of a query on a probabilistic database is the combined probability weight of all possible worlds in which t is in the result of the query.

Recent probabilistic database management systems, including Trio [6], MystiQ [10], and MayBMS [3, 25], have settled on essentially a single internal mechanism for representing uncertainty, which has also been shown to be complete and succinct [6, 3]: conditional tables (cf. [20, 18]) over discrete, independent random variables. In such a conditional table, each tuple is associated with a condition that is a conjunction (or *clause*) of atomic conditions of the form $x = a$. Here x is a random variable and a is a domain value of x.

It is well known that for positive relational algebra, the conditions associated with tuples in conditional tables can be expressed as DNF formulas. This can be done efficiently, essentially using traditional relational query processing techniques with very simple extensions to combine conditions. State-of-the-art confidence computation algorithms require conditions in DNF form. Full relational algebra can be evaluated efficiently on c-tables as well, but in general does not yield DNF conditions, and turning the resulting conditions into DNFs takes provably exponential time and space in general.[1]

Example 1.1. We view a social network as an undirected graph in which nodes represent individuals and edges represent, say, friendship. Assume that the presence of edges is uncertain and edges are associated with a degree of belief in their presence (e.g., from mail server logs). No correlations between the probabilities of edges are known, so the edge probabilities are assumed independent.

[1] An important case in which difference operations can be dealt with efficiently in the DNF framework is presented in [24].

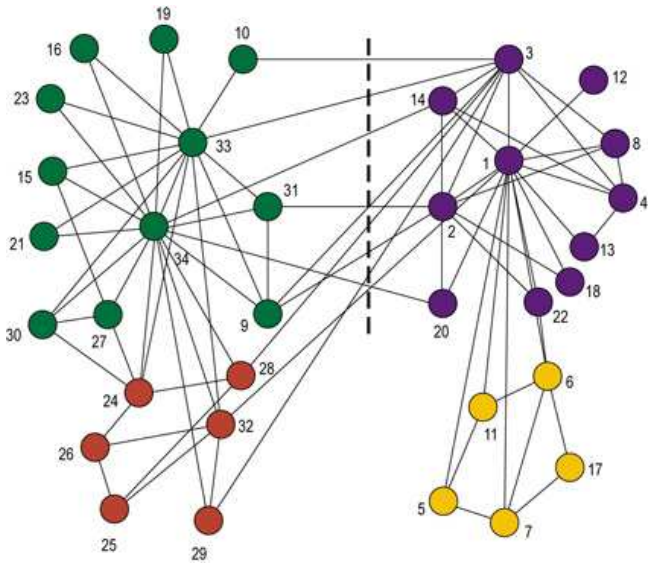


Figure 1: Social network of Zachary's Karate club [41]. In this club, two factions have formed, the one of the trainer's (node 1) supporters and the faction of the administrator (node 34). The club eventually split along the dashed line. (Image source: https://www-eng.llnl.gov/eng_sys_ki/eng_sys_ki_decomp.html)

E	U	V	P	ϕ
	5	7	.9	e_1
	5	11	.8	e_2
	6	7	.1	e_3
	6	11	.9	e_4
	6	17	.5	e_5
	7	17	.2	e_6

(a)

E'	U	V	$\in$	P	ϕ
	5	7	1	.9	e_1
	5	7	0	.1	$\neg e_1$
	⋮	⋮	⋮	⋮	
	7	17	1	.2	e_6
	7	17	0	.8	$\neg e_6$

(b)

R	ϕ
	$e_3 \wedge e_5 \wedge e_6$

(c)

R	V	ϕ
	11	$(e_1 \wedge e_2) \vee (e_3 \wedge e_4)$

(d)

Figure 2: Tuple-independent (a) and block-independent-disjoint representation (b) of the yellow subgraph of the social network of Figure 1, plus c-tables (c,d) for the results of the two queries of Example 1.1.

Consider the social network of Figure 1. For simplicity, we restrict ourselves to the yellow subgraph. We can represent this edge relation (with made-up edge probabilities) by a so-called *tuple-independent* table, shown in Figure 2 (a). The Boolean random variables $e_1, \ldots, e_6$ represent the six edges – that is, the i-th edge is present in those worlds in which e_i is true. This table represents 2^6 possible worlds, each holding a relation of schema $E(U, V)$. For instance, the world with edges e_1, e_2, and e_3, but not the others, has probability $.9 * .8 * .1 * (1 - .9) * (1 - .5) * (1 - .2)$.

Now let us ask for the probability that there is a triangle (a 3-clique of friends) in this graph.[2] Using MayBMS syntax, we can express this by the query

```
select conf() as triangle_prob
from   E e1, E e2, E e3
where  e1.v = e2.u and e2.v = e3.v
and    e1.u = e3.u
and    e1.u < e2.u and e2.u < e3.v;
```

The (Boolean) relational algebra part of this query computes the c-table of Figure 2 (c). That is, there is a triangle in those worlds that contain the third, fifth, and sixth edge, and the probability of this event is $\Pr[e_3 \wedge e_5 \wedge e_6] = .1 * .5 * .2 = .01$.

Note that we have modeled the edge relation E as a subset of the total order induced by the numbering of the nodes. Thus, the edge relation E is not yet symmetric, even though we mean the network to be an undirected graph. The above query, however, deals with this. It is easy to construct a c-table representing the symmetric closure of this relation by positive relational algebra. However, this table will not be tuple-independent. Starting from such a table, writing the triangle query is straightforward.

Now consider the alternative representation of the edge relation, E' in Figure 2 (b), still as a subset of the total order on the nodes. This is a *block-independent-disjoint* table (cf. [10], also called *x-relation* in [6]). The difference to E is that now the alternatives – each edge is either present or not – are both represented. Alternatives in a group are mutually exclusive and different groups are independent from each other, that is, E' is equivalent to E and, taking the condition columns ϕ into account, both are c-tables.

Starting from E', we can ask queries involving the absence of an edge from a world, such as the query for nodes within two, but not one, degrees of

[2]Such small patterns are also called *motifs* in this context.

separation from node number 7. The query shall be skipped here, although it is not hard to write in positive relational algebra assuming a relation of those edges missing with certainty from the graph is available. The result is the c-table of Figure 2 (d). Thus, node 11 is possibly within two (but not one) degrees of separation from node 7, but no other node is. To compute the probability of the DNF $(e_1 \wedge e_2) \vee (e_3 \wedge e_4)$, we can use the inclusion-exclusion principle:

$$\begin{aligned} \Pr[(e_1 \wedge e_2) \vee (e_3 \wedge e_4)] &= \Pr[e_1 \wedge e_2] + \Pr[e_3 \wedge e_4] \\ &- \Pr[e_1 \wedge e_2 \wedge e_3 \wedge e_4] \\ &= .7452 \end{aligned}$$

1.1 State of the Art in Confidence Computation

The inclusion-exclusion principle just mentioned yields a straightforward solution method, which however is badly exponential in the number of clauses of the DNF. It is known since [37] that counting the number of solutions to a DNF is #P-hard, and computing the probability of a DNF with independent random variables is a generalization of this problem [17, 10]. It was first shown in work by Karp, Luby, and Madras [22, 23] that there is a fully polynomial-time randomized 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 [17, 10, 35, 24].

The techniques based on [22, 23] 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 (= 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 $\Pr\left[|p - \hat{p}| \geq \epsilon \cdot p\right] \leq \delta$.

The work by Karp, Luby, and Madras has started a line of research in the theory community to derandomize these approximation techniques, eventually leading to a polynomial time deterministic $(\epsilon, 0)$-approximation algorithm [36] (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 (above 2^{50} for 3-DNF) and the algorithm is not practical. This is in contrast to observations that the Karp-Luby Monte Carlo algorithm is practical (e.g. [2, 35], and the

experiments of the present paper). In fact, the Karp-Luby algorithm is the state-of-the-art (and only) approximation algorithm used in current discrete probabilistic database management systems such as MystiQ and MayBMS.

More related work on counting in combinatorial structures has been done in the AI community, e.g. [39], which however focuses on lower-bounding in extremely large combinatorial problems, with bounds off the true count by many orders of magnitude. Thus, while such techniques are of interest in their application domains, they cannot be used for computing probabilities. Further, extensions of the Davis-Putnam procedure have been used for counting the solutions to formulae [7]. Our own recent work [25] uses similar ideas in the context of probabilistic databases, yielding exact confidence computation algorithms that are more efficient than algorithms based on the inclusion-exclusion principle. We also observed an *easy-hard-easy pattern* in input instances. Outside a certain range of variable-to-clause ratios, confidence computation tends to be easy, and our exact algorithm of [25], even though exponential-time in the worst-case, is even competitive with approximation techniques such as Karp-Luby which are guaranteed to be polynomial-time.

1.2 Scalable Confidence Computation with SPROUT

Although confidence computation in probabilistic databases is #P-hard in general, some classes of queries can be processed efficiently with scalable techniques. For instance, MystiQ [10] and SPROUT [33] propose secondary-storage algorithms for tractable conjunctive queries without self-joins on so-called tuple-independent probabilistic databases. In addition, there is strong theoretical and experimental evidence that MystiQ and SPROUT perform orders of magnitude faster than existing main-memory techniques for exact and approximate confidence computation techniques based on general-purpose compilation techniques [25] or Monte Carlo simulations using the Karp-Luby estimator [22]. This key observation supports the idea that specialized secondary-storage algorithms, which take the query and the probabilistic database model into account, have better chances at improving the state of the art in query evaluation on probabilistic databases. Surprisingly, though, there is very little available beyond the aforementioned works. While it is true that the tuple-independent model is rather limited, it still represents a valid starting point for developing scalable query processing techniques. In addition, independence occurs naturally in many large data sets, such as census data [5] and social networks [1].

This thesis is the first to investigate the problem of efficiently computing

Subscribers

Id	DomId	RDate	V_s	P_s
1	1	1995-01-10	x_1	0.1
2	1	1996-01-09	x_2	0.2
3	1	1997-11-11	x_3	0.3
4	2	1994-12-24	x_4	0.4
5	2	1995-01-10	x_5	0.5

Events

Description	PDate	V_e	P_e
XMas party	1994-12-24	y_1	0.1
Fireworks	1996-01-09	y_2	0.2
Theatre	1997-11-11	y_3	0.3

Query Answer before conf()

DomId	V_s	P_s	V_e	P_e
1	x_1	0.1	y_2	0.2
1	x_1	0.1	y_3	0.3
1	x_2	0.2	y_3	0.3
2	x_4	0.4	y_2	0.2
2	x_4	0.4	y_3	0.3
2	x_5	0.5	y_2	0.2
2	x_5	0.5	y_3	0.3

Query Answer

DomId	P
1	0.098
2	0.308

Figure 3: Tuple-independent probabilistic database and the answer to our query in Section 1.2.

the confidences of distinct tuples in the answers to conjunctive queries with inequalities ($<$) on tuple-independent probabilistic databases. It provides a characterization of a large class of queries that can be computed in polynomial time data complexity and proposes an efficient secondary-storage evaluation technique for such tractable queries. The characterizations, as well as the technique, are based on structural properties of the inequalities present in the query and of a special form of decision diagrams, called Ordered Binary Decision Diagrams (OBDDs) [29], which are used as a compiled succinct representation of the uncertainty manifested in the query answer.

We illustrate our approach on a tuple-independent probabilistic database of subscribers and events. Assume we have archived information on subscribers, including a subscriber identifier, a domain identifier, and a date of registration for event services. The information on events includes a description and a publication date. Figure 3 gives a database instance, where each tuple is associated with an independent Boolean random variable (hence the database is tuple-independent). These variables are given in the V-columns and their probabilities (for the "true" assignment) in the P-columns. Such a tuple-independent database represents exponentially many possible instances, one instance for each total valuation of the variables in the database. For example, a valuation that maps x_1 and y_1 to true and all

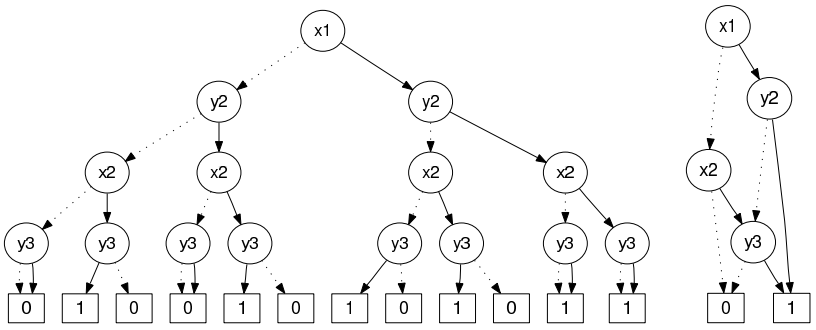


Figure 4: Decision tree (left) and OBDD (right) for the lineage of the answer tuple with DomId=1.

other variables to false defines the instance with one subscriber (with Id=1) and one event (Xmas party). The probability of this instance can be simply computed as the product of the probabilities of x_1 and y_1 being true and of all remaining variables being false.

We would like to compute, for each domain, the likelihood that its subscribers participated in the broadcasted events – subscribers can participate in an event if their registration date is before the publication date of the event:

```
select DomId, conf() as P from Subscribers, Events
where  RDate < PDate group by DomId;
```

The aggregate function `conf()` is used here to specify the confidence computation for each distinct DomId value.

The answer to a query on a probabilistic database can be represented by a relation pairing possible result tuples with a formula over random variables, called lineage [6]. For example, the lineage of the answer tuple t with DomId=1 is $x_1y_2+x_1y_3+x_2y_3$. The lineage of a tuple describes symbolically the set of worlds in which that tuple occurs in the query result: There is a one-to-one correspondence between these worlds and the total valuations that satisfy the lineage [10]. Given a decision tree over all variables of this lineage, as in Figure 4(left), the satisfying valuations are represented by the root-to-leaf paths that lead to a leaf labeled 1 (true). Each node in the decision tree corresponds to the decision for one variable. We follow the solid outgoing edge in case the variable is assigned to true and the dotted edge otherwise.

The confidence in a tuple is the probability for true of its associated lineage [10]. Decision trees that represent lineage can be used to compute

tuple confidences: Simply sum up the probabilities of each path leading to 1. This holds because the paths are pairwise mutually exclusive. This approach is, however, extremely expensive as one has to iterate over an exponential number of possible valuations.

Our approach is to directly compile the lineage into a *compressed* representation of the decision tree, called OBDD: Figure 4(right) gives an OBDD for the lineage of tuple t. To see the correspondence between the decision tree and its equivalent OBDD, consider removing redundant nodes and factoring out common subtrees and representing them only once. For example, the first node n for variable y_2 has two identical children. We only need to represent them once and have both outgoing edges of n point to the same subtree. This also means that n is redundant and can be removed, for the decision on whether y_2 is true or false at that point is not relevant for the overall satisfiability.

Computing the probability of the OBDD can be done in one bottom-up traversal. The probability Pr of a node n for a variable v and with children l for $v = false$ and r for $v = true$ can be expressed using the probabilities of the children as follows: $Pr(n) = Pr(\overline{v}) \cdot Pr(l) + Pr(v) \cdot Pr(r)$. In case of a leaf node, $Pr(1) = 1$ and $Pr(0) = 0$.

We show that the lineage of any query from a large query class on any tuple-independent database can be always efficiently compiled into an OBDD whose size is polynomial in the number of variables of that lineage. Moreover, we need not materialize the OBDD *before* computing its probability. In fact, we only need to keep around a small number of running probabilities for fragments of the overall OBDD and avoid constructing it entirely. This number depends on the query size and is independent of the database size.

Consider first a bottom-up traversal of the OBDD and two values: p_x for the probabilities of OBDD fragments rooted at nodes for V_s-variables x_1 and x_2, and p_y for V_e-variables y_2 and y_3. These values are updated using recurrence formulas that can be derived from the query structure and mirror the probability computation of OBDD nodes. Updating p_y and p_x at a node for variable v is done using the formulas

$$p_y = \Pr(\overline{v}) \cdot p_y + \Pr(v) \cdot \Pr(1)$$
$$p_x = \Pr(\overline{v}) \cdot p_x + \Pr(v) \cdot p_y$$

The difference between the recurrence formulas of p_x and p_y reflects the position in the OBDD of nodes for variables of V_s and of V_e: Whereas the nodes for V_e-variables have always a child 1 on the positive branch, those for V_s-variables have a child node for a V_e-variable on the positive branch.

Using these two recurrence formulas, we go up levelwise until we reach the root of the OBDD. The probability of the OBDD is then p_x. We now apply the recurrence formulas and obtain the update sequence (initially, $p_x = p_y = 0$)

$$\begin{aligned}
&\text{Step 1. } p_y = \Pr(\overline{y_3}) \cdot p_y + \Pr(y_3) \cdot \Pr(1) = \Pr(y_3) = 0.3\\
&\text{Step 2. } p_x = \Pr(\overline{x_2}) \cdot p_x + \Pr(x_2) \cdot p_y = 0.06\\
&\text{Step 3. } p_y = \Pr(\overline{y_2}) \cdot p_y + \Pr(y_2) \cdot \Pr(1) = 0.44\\
&\text{Step 4. } p_x = \Pr(\overline{x_1}) \cdot p_x + \Pr(x_1) \cdot p_y = 0.098
\end{aligned}$$

The confidence of our tuple is thus 0.098.

Given the recurrence formulas for updating p_x and p_y, the same result can also be obtained in one ascending scan of the relational encoding of the lineage. This approach completely avoids the OBDD construction. The updates to p_x or p_y are now triggered by changes between the current and the previous lineage clauses. We first access x_2y_3 and trigger an update to p_y. The next clause x_1y_3 differs in the V_s-variable from the previous clause and triggers an update to p_x. When we read the last clause x_1y_2, the change in the V_e-variable triggers an update to p_y. We then reach the end of table, which triggers an update to p_x, which is then also returned as the probability of the lineage.

1.3 Contributions

In summary, the thesis discusses (A) scalable exact confidence computation for conjunctive queries with inequalities and (B) general-purpose approximate confidence computation. The main technical contributions of this thesis are as follows.

(A) Scalable Exact Confidence Computation for Conjunctive Queries with Inequalities:

1 To the best of our knowledge, this work is the first to define tractable conjunctive queries with inequalities ($<$) on tuple-independent probabilistic databases. This problem is fundamental to probabilistic databases and was recently stated open [11]. The tractable queries are defined using the inequality relationships on query variables: Each input table contributes with at most one attribute to inequality conditions, yet there may be arbitrary inequalities between the contributing attributes.

2 We cast the exact confidence computation problem as an OBDD construction problem and show that the lineage of tractable queries can be efficiently compiled into polynomial-size OBDDs. We relate the OBDD size to properties of the tree of the inequality conditions present in the query, and show that the OBDDs are linear in the number of variables in the lineage.

3 The OBDD-based technique requires to first store the OBDDs in main memory. We overcome this limitation by proposing a new secondary-storage variant that avoids the materialization of the OBDD, and computes, in one scan over the lineage, the probabilities of fragments of the OBDD and then combines them on the fly.

4 We report on experiments with probabilistic TPC-H data and comparisons with an exact confidence computation algorithm and an approximate one with polynomial-time and error guarantees [25]. In cases when the competitors finish the computation within the allocated time, our algorithm outperforms them by up to two orders of magnitude.

(B) General-Purpose Approximate Confidence Computation:

5 We introduce a novel deterministic approximation algorithm with error guarantees for computing confidences of tuples in query answers. In contrast to much of the existing work in probabilistic databases, this algorithm is not only applicable to restricted classes of queries (such as hierarchical conjunctive queries without self-joins) or probabilistic databases (such as tuple-independent tables or x-relations), but is generic. The same algorithm can also be used for exact computation.

6 The approximation algorithm is based on a number of fundamental ideas from combinatorial algorithms, constraint satisfaction, and verification (such as decompositions, variable elimination, and OBDDs), and turns out to be both simple and extensible.

7 Our techniques incrementally compile the DNF condition into a novel type of decision diagram called *d-trees*. Such diagrams decompose DNFs using negative correlations, independence, and factored representations that are easy to compute. Given a d-tree and an approximate (or exact) probability for each of its leaves, we can compute an overall approximate (or exact) probability in just one pass over the d-tree.

8 We then show how a given expression can be *incrementally* compiled into fragments of a d-tree *without fully materializing it*. We devise heuristics that allow us to obtain close lower and upper probability bounds within a few compilation steps, thus avoiding exhaustive complete traversal of a d-tree for that expression. For a given absolute or relative error bound, our heuristics decide locally whether to further compile a subexpression under a certain node of the d-tree or move on to a following node. For this, we devise a safety check on which such subexpressions can be discarded while still guaranteeing the overall error bound.

9 We also show that d-trees in conjunction with our heuristics yield an alternative polynomial-time algorithm for exact confidence computation for cases from the literature for which efficient algorithms for confidence computation are known, namely the hierarchical queries without self-joins [10], with inequalities [31], and certain additional cases in which functional dependencies on the data yield tractability[33]. In fact, these are all the currently known tractable cases in the absence of self-joins. In these cases, our algorithm guarantees a running time linear in query size and quadratic in the size of the DNF.

10 We experimentally verify the robustness of our algorithm. We evaluate both tractable and hard queries on various kinds of probabilistic databases, such as tuple-independent TPC-H, random graphs, and social networks. In all these experiments, our algorithm consistently outperforms the Karp-Luby FPTRAS by orders of magnitude.

11 Our experiments also show that our algorithm performs well in practice compared to special algorithms for known tractable queries, which exploit knowledge about the query but are *only* applicable to those tractable queries, while our algorithm is generic and only relies on a smart choice of a heuristics that was not specifically designed to handle such queries well. Thus, our algorithm bears promise of dealing well, or being extensible to deal well, with "easy" cases not yet discovered.

12 We implemented both techniques and integrated them into the SPROUT query engine [33] and MayBMS probabilistic database management system. SPROUT is a scalable query engine for probabilistic databases that extends the query engine of PostgreSQL with special physical aggregation operators for confidence computation.

Several publications and submissions are based on the contributions of this thesis:

- Contributions 1 to 4 appeared in the proceedings of ACM Special Interest Group on Management of Data Conference 2009 (ACM SIGMOD 2009) under the title "Secondary-Storage Confidence Computation for Conjunctive Queries with Inequalities" [31]. In addition, this paper passed the repeatability and workability evaluation (RWE) of ACM SIGMOD 2009 (63 research papers are accepted, 19 of them participated in RWE and 10 passed).

- Contributions 5 to 11 are already submitted to 26th International Conference on Data Engineering (IEEE ICDE 2010) under the title "Approximate Confidence Computation in Probabilistic Databases" [32].

- Contribution 12, MayBMS probabilistic database management system with SPROUT query engine, has been released in March 2009 as version 2.1 beta and version 2.1 will be released in September 2009. The system is publicly available at

  http://maybms.sourceforge.net and

  http://web.comlab.ox.ac.uk/projects/SPROUT/index.html

  It was demonstrated at ACM SIGMOD 2009 under the title "MayBMS: A Probabilistic Database Management System" [19].

1.4 Outline

The structure of the thesis is as follows.

- Section 2 introduces the background acknowledge by presenting probabilistic databases (U-relational and tuple-independent databases), syntax and semantics of queries and OBDDs.

- Section 3 defines syntactically the tractable conjunctive queries with inequalities and without self-joins on tuple-independent probabilistic databases, introduces an OBDD-based approach for efficiently processing these queries and its secondary-storage variant.

- Section 4 introduces a novel type of decision diagrams, called d-trees, for lineage decomposition and shows how they naturally capture the lineage of currently known tractable queries.

- Section 5 develops a deterministic approximation algorithm for confidence computation based on d-trees.
- Section 6 gives an overview of state-of-the-art MayBMS probabilistic database management system with SPROUT as its query engine and demonstrates an application scenario of human resources management in MayBMS.
- Section 7 presents our experimental findings and compares the performance of our techniques with the state of the art.
- Appendix A gives proofs for the major statements in the thesis.
- Appendices B, C, D and E list the queries used in the experiments.

2 Preliminaries

2.1 Probabilistic Databases

The data models used in this thesis are U-relational databases and tuple-independent probabilistic databases. We first define U-relational databases and then pose some restrictions on them to get tuple-independent databases.

We consider a finite set of *independent* random variables $\mathbf{X}$ with finite domains. We denote the domain of a random variable $x \in \mathbf{X}$ by Dom_x. *Atomic events* (or *atomic formulas*) are of the form $x = a$ where $x \in \mathbf{X}$ and $a \in \mathrm{Dom}_x$. A (positive propositional) *formula* (or *event*) is constructed from atomic events using the binary operations $\vee$ (logical "or") and $\wedge$ (logical "and"). A conjunction of atomic events $(x_1 = a_1) \wedge \cdots \wedge (x_n = a_n)$ is called a clause. A *DNF formula* is a disjunction of clauses, namely, of the form $c_1 \vee ... \vee c_n$, where $c_1, ..., c_n$ are clauses.

We define finite probability distributions on these variables and their domains. Such a probability distribution is completely specified by a function P that assigns a probability $P(x = a) \in (0, 1]$ to each atomic event $x = a$ such that, for each random variable x,

$$\sum_{a \in \mathrm{Dom}_x} P(x = a) = 1.$$

The variable, their domains and the probability distributions are stored in a so-called *world-table*, which is a relation with schema W(Var,Dom,Prob) which contains all triple of (x, a, p), in which x is a variable, $a \in \mathrm{Dom}_x$ and $p = P(x = a)$ [4].

A *valuation* of $\mathbf{X}$ is an assignment of random variables in $\mathbf{X}$ to one of their domain values. We can identify *possible worlds* with valuations, or equivalently, with clauses that contain exactly one atomic event for *ecach* of the random variables. Since all variables are independent, the probability of a valuation (possible world) $(x_1 = a_1) \wedge \cdots \wedge (x_n = a_n)$ is

$$\prod_{i=1}^{n} P(x_i = a_i).$$

The *world-set* is represented by the finite set of all valuations. We are now ready to define U-relational databases.

Definition 2.1. *[4] A U-relational database for a world-set over schema* $\sum = (R_1[\overline{A}_1], ..., R_k[\overline{A}_k])$ *is a tuple* $(U_1, ..., U_k, W)$*, where* W *is a world-table and each relation* U_i *has schema* $U_i[\overline{D}_i; \overline{A}_i]$ *such that* $\overline{D}_i$ *defines clauses over* W*.* □

The clauses in the U-relational databases can be represented relationally by values in the additional variable (V) and domain value (D) columns. For instance, $\{x_1 = a_1, x_2 = a_2\}$ is represented by values x_1, a_1, x_2 and a_2 in columns V_1, D_1, V_2 and D_2.

A U-relational database represents a set of database instances, also called *possible worlds*. We define a function rep such that given a U-relational database T, $rep(T)$ is the set of possible worlds of T. Given a valuation f, the instance of each U-relation U^{rep} is the set of tuples $\vec{a}$ such that $(\vec{a}, \phi) \in U^{rep}$ and $f(\phi)$ is true, where ϕ is a clause.

Tuple-independent probabilistic databases are a special type of U-relational databases. First, without loss of generality, all variables can considered to have binary domain $\{true, false\}$. Second, every clause in tuple-independent probabilistic databases is of the form of $\{x = true\}$ and none of them share any variables. For the sake of simplicity and fast query evaluation, we represent the clauses and world-table with a variable column (V) and a probability column (P) in each tuple-independent relation.

2.2 Syntax and Semantics of Queries

2.2.1 Syntax

Two techniques introduced in this thesis target different classes of queries. For d-tree approximation algorithm, discussed in Sections 4 and 5, we focus on positive relational algebra, namely, select-project-join-union queries, in U-relational databases. For the OBDD-based technique, discussed in Section 3, we consider conjunctive queries without self-joins and with inequalities ($<$) in tuple-independent probabilistic databases. In this section, we mainly discuss the syntax of the latter.

We denote conjunctive queries without self-joins and with inequalities ($<$) by queries in the sequel. We write queries in datalog notation. For instance,

$$Q(\overline{x_0}) :\!\!- R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \phi$$

defines a query Q with head variables $\overline{x_0} \subseteq \overline{x_1} \cup \ldots \cup \overline{x_n}$ and a conjunction of distinct positive relational predicates $R_1, \ldots, R_n$, called *subgoals*, as body. The conjunction ϕ defines inequalities on query variables or variables and constants, e.g., $B < C$ or $B < 5$. Equality-based joins can be expressed by variables that occur in several subgoals. Equalities with constants can be expressed by replacing the variables with constants in subgoals. Figure 5 gives four Boolean queries with inequalities.

R	A	V_r
	2	x_1
	4	x_2
	6	x_3

S	B	C
	2	2
	2	4
	4	2
	4	6
	6	4

T	D	V_t
	2	y_1
	4	y_2
	6	y_3

R'	E	F	$V_{r'}$
	1	3	x_1
	3	5	x_2
	5	7	x_3

T'	G	H	$V_{t'}$
	1	3	y_1
	3	5	y_2
	5	7	y_3

S'	E	F	G	H
	1	3	1	3
	1	3	3	5
	3	5	1	3
	3	5	5	7
	5	7	3	5

The same lineage $x_1y_1 + x_1y_2 + x_2y_1 + x_2y_3 + x_3y_2$ is associated with the answers to:

Q_1:-$R(X), S(X, Y), T(Y)$ on database (R, S, T).

Q_2:-$R'(E, F), S(B, C), T'(G, H), E < B < F, G < C < H$ on database (R', S, T').

Q_3:-$R(A), S(E, F, G, H), T(D), E < A < F, G < D < H$ on database (R, S', T).

Q_4:-$R(X), S(X, C), T'(G, H), G < C < H$ on database (R, S, T').

Figure 5: Tuple-independent tables (P-columns not shown) and Boolean conjunctive queries with inequalities.

2.2.2 Semantics

Figure 6 gives a graphical view of query evaluation under possible world semantics in U-relational databases. Conceptually, queries are evaluated in each possible world $(\{\mathcal{A}_1, \ldots, \mathcal{A}_n\})$. Given a query q and a probabilistic database T, the probability of a distinct answer tuple t is the probability of t being in the result of q in the worlds of T, or equivalently,

$$\Pr[t \in q(T)] = \sum_{\mathcal{A}_i:\, t \in q(\mathcal{A}_i)} \Pr[\mathcal{A}_i].$$

However, performing query evaluation in all possible world is highly inefficient. Theorem 2.2 comes to rescue by showing that there is a corresponding query $\overline{q}$ for input query q which can be evaluated efficiently.

Theorem 2.2. *[4] For any positive relational algebra query q over any U-relational database T, there exists a positive relational algebra query $\overline{q}$ of*

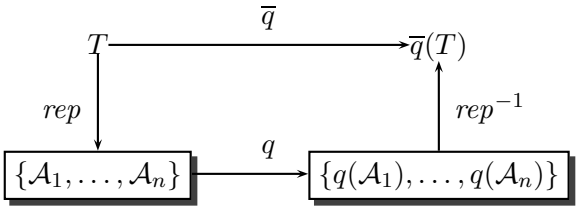


Figure 6: Semantics of query evaluation in U-relational databases[4]. T is a U-relational database, rep is a function mapping T to a set of possible worlds $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, q is a positive relational algebra query, $q(\mathcal{A}_i)$ is the output of evaluation of q in $\mathcal{A}_i$ and $\overline{q}$ is the corresponding efficient query of q.

polynomial size such that

$$\overline{q}(T) = rep^{-1}(\{q(\mathcal{A}_i) \mid \mathcal{A}_i \in rep(T)\}).$$

$$\begin{array}{rcl}
[\![R \times S]\!] & := & \pi_{U_R.\overline{VD} \cup U_S.\overline{VD} \to \overline{VD}, sch(R), sch(S)}(\\
 & & \quad U_R \bowtie_{U_R.\overline{VD} \text{ consistent with } U_S.\overline{VD}} U_S) \\
[\![\sigma_\phi R]\!] & := & \sigma_\phi(U_R) \\
[\![\pi_{\vec{B}} R]\!] & := & \pi_{\overline{VD}, \vec{B}}(R) \\
[\![R \cup S]\!] & := & U_R \cup U_S
\end{array}$$

Figure 7: Operations in query evaluation in U-relational databases[4].

Figure 7 shows how to obtain $\overline{q}$ from q by mapping the operations in q according to certain rules. The main message in Figure 7 is to carry the variable columns and domain value columns ($\overline{VD}$ in the figure) of the input U-relations to the output. Selection and union are the same as their counterparts in relational algebra. Projection is to add $\overline{VD}$ to the attribute list. Product is more complex: In addition to the product, it also checks the consistency of values in $\overline{VD}$ and performs a union on them.

2.3 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) are commonly used to represent compactly large Boolean expressions [29]. We find that OBBDs can naturally represent query lineage.

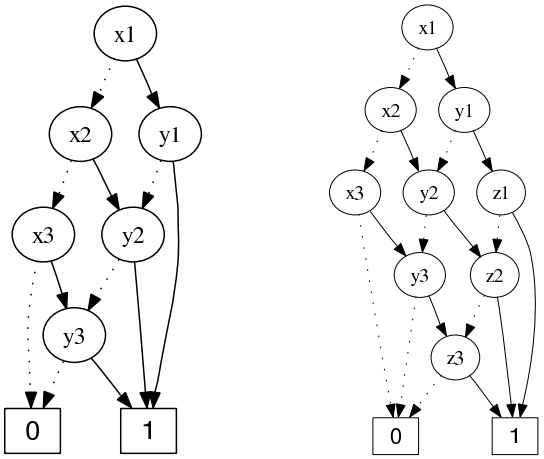


Figure 8: OBDDs used in Examples 2.3 and 3.22.

The idea behind OBDDs is to decompose Boolean formulas using variable elimination and to avoid redundancy in the representation. The decomposition step is based on exhaustive application of Shannon's expansion: Given a formula ϕ and one of its variables x, we have $\phi = x \cdot \phi\mid_x + \bar{x} \cdot \phi\mid_{\bar{x}}$, where $\phi\mid_x$ and $\phi\mid_{\bar{x}}$ are ϕ with x set to true and false, respectively. The order of variable eliminations is a total order π on the set of variables of ϕ, called *variable order*. An OBDD for ϕ is uniquely identified by the pair (ϕ, π).

OBDDs are directed acyclic graphs with two terminal nodes representing the constants 0 (false) and 1 (true), and non-terminal nodes representing variables. Each node for a variable x has two outgoing edges corresponding to the two possible variable assignments: a high (solid) edge for $x = 1$ and a low (dashed) edge for $x = 0$. To evaluate the expression for a given set of variable assignments, we take the path from the root node to one of the terminal nodes, following the high edge of a node if the corresponding input variable is true, and the low edge otherwise. The terminal node gives the value of the expression. The non-redundancy is what normally makes OBDDs more compact than the textual representation of Boolean expressions: a node n is redundant if both its outgoing edges point to the same node, or if there is a node for the same decision variable and with the same children as n.

Constructing succinct OBDDs is an NP-hard problem [29]. The choice of variable order can vary the size of the OBDD from constant to exponential in the number of variables. Moreover, some formulas do not admit polynomial-size OBDDs. In this paper, we nevertheless show that lineage associated

with the answer to any query can be compiled in OBDDs of polynomial size.

Example 2.3. Figure 8(left) depicts the OBDD for

$$x_1(y_1 + y_2 + y_3) + x_2(y_2 + y_3) + x_3y_3$$

using the variable order $x_1x_2x_3y_1y_2y_3$. We show how to construct this OBDD. Let $\alpha_2 = x_2(y_2 + y_3)$ and $\alpha_3 = x_3y_3$. By eliminating x_1, we obtain $y_1 + y_2 + y_3$, if x_1 is true, and $\alpha_2 + \alpha_3$ otherwise. We eliminate x_2 on the branch of $\overline{x_1}$ and obtain $y_2 + y_3$ in case x_2 is true, and α_3 otherwise. We continue on the path of $\overline{x_2}$ and eliminate x_3: We obtain y_3 in case x_3 is true, and false otherwise. All incomplete branches correspond to sums of variables: $y_1 + y_2 + y_3$ includes $y_2 + y_3$, which includes y_3. We compile all these sums by first eliminating y_1. In case y_1 is true, then we obtain true, otherwise we continue with the second sum, and so on. Although some variables occur several times in the input, the OBDD thus has one node per input variable. □

The probability of an OBDD can be computed in time linear in its size using the fact that the branches of any node represent mutually exclussive expressions. The probability of any node n is the sum of the probabilities of their children weighted by the probabilities of the corresponding assignments of the decision variable at that node. The probability of the terminal nodes is given by their label (1 or 0).

3 Tractable Conjunctive Queries with Inequality Joins

Section 3.1 defines classes of tractable conjunctive queries with inequalities and without self-joins. The confidence computation algorithms developed in Section 3.2 focus on a particular class of Boolean product queries with a special form of conjunction of inequalities. Section 3.3 introduces a secondary-storage variant of this algorithm. All the contributions in this section appeared in [31].

3.1 Tractable Conjunctive Queries

In this section, we first define a class of tractable conjunctive queries with inequalities and without self-joins. Sections 3.1.1 through 3.1.4 extend the tractability to considerably larger query classes by taking into account head variables, materialization, equality joins and database constraints.

Definition 3.1. *Let the disjoint sets of query variables $\overline{x_1}, \ldots, \overline{x_n}$. A conjunction of inequalities over these sets has the* max-one *property if at most one query variable from each set occurs in inequalities with variables of other sets.*

Definition 3.2. *An* IQ query *has the form*

$$Q\text{:-}R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \phi$$

where $R_1, \ldots, R_n$ are distinct tuple-independent relations, the sets of query variables $\overline{x_1}, \ldots, \overline{x_n}$ are pairwise disjoint, and ϕ has the max-one property over these sets.

Example 3.3. The following are IQ queries

$$q_1\text{:-}R'(E,F), T(D), T'(G,H), E < D < H$$
$$q_2\text{:-}R'(E,F), T(D), S(B,C), E < D, E < C$$
$$q_3\text{:-}R(A), T(D)$$
$$q_4\text{:-}R(A), T(D), R'(E,F), T'(G,H), A < E, D < E, D < G \quad \square$$

We will show in Section 3.2 that

Theorem 3.4. *IQ queries can be computed on tuple-independent databases in polynomial time.*

We represent the conjunction of inequalities ϕ by an *inequality graph*, where there is one node for each query variable, and one oriented edge from A to B if the inequality $A < B$ holds in ϕ. We keep the graph minimal by removing edges corresponding to redundant inequalities, which are inferred using the transitivity of inequality. Inequalities on variables of the same subgoal are not represented, for they can be computed trivially on the input relations. Each graph node thus corresponds to precisely one query subgoal. We categorize the IQ queries based on the structural complexity of their inequality graphs in paths, trees and graphs.

Example 3.5. Consider again the IQ queries of Example 3.3: q_1 is a path, q_2 is a tree, and q_3 and q_4 are graphs. □

IQ queries are limited in three major ways: they are Boolean, have no equality joins, and have restrictions concerning inequalities on several query variables of the same subgoal. We address these limitations next.

3.1.1 Non-Boolean Queries

Our first extension considers non-Boolean conjunctive queries, whose so-called Boolean reducts are IQ queries.

Definition 3.6. *The* Boolean reduct *of a query*

$$Q(\overline{x_0}) \text{:-} R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \phi$$

$$\textit{is } Q' \text{:-} R_1(\overline{x_1} - \overline{x_0}), \ldots, R_n(\overline{x_n} - \overline{x_0}), \phi'$$

where ϕ' is ϕ without inequalities on variables in $\overline{x_0}$.

Example 3.7. The query from the introduction

$$q(DomId) \text{:-} Subscr(Id, DomId, RDate), Events(Descr, PDate), \\ RDate < PDate$$

has as Boolean reduct the IQ query

$$q' \text{:-} Subscr(Id, RDate), Events(Descr, PDate), RDate < PDate \square$$

In case the Boolean reduct of a query Q is in IQ, we can efficiently compute the probability of each of the distinct answer tuples of Q by employing probability computation algorithms for IQ queries. This is because for a given value of the head variables, Q becomes a Boolean IQ query.

Proposition 3.8. *The queries, whose Boolean reducts are IQ queries, can be computed on tuple-independent databases in polynomial time.*

3.1.2 Efficient-Independent Queries

Our next extension considers queries that can be efficiently materialized as tuple-independent relations.

Definition 3.9. *A query is* efficient-independent *if for any tuple-independent database, the distinct answer tuples are pairwise independent and their probability can be computed in polynomial time.*

Such queries can be used as subqueries at the place of tuple-independent relations in an IQ query.

Proposition 3.10. *Let the relations $R_1, \ldots, R_n$ be the materializations of queries $q_1(\overline{x_1}), \ldots, q_n(\overline{x_n})$. The query*

$$Q(\overline{x_0})\text{:-}R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \phi.$$

can be computed in polynomial time if $q_1, \ldots, q_n$ are efficient-independent and Q's Boolean reduct is an IQ query.

3.1.3 Equality Joins

One important class of efficient-independent queries is represented by the hierarchical queries (without self-joins) whose head variables are maximal [12].

Definition 3.11. *A conjunctive query is* hierarchical *if for any two non-head query variables, either their sets of subgoals are disjoint, or one set is contained in the other. The query variables that occur in all subgoals are maximal.*

Example 3.12. Consider a probabilistic TPC-H database with relations Cust, Ord, Item. The following query asks for the probability that Joe placed orders:

$$\begin{aligned} Q\text{:-}&\text{Cust}(ckey, 'Joe', regdate), \text{Ord}(okey, ckey, odate), \\ &\text{Item}(okey, ckey, shipdate). \end{aligned}$$

This query is hierarchical: subgoals(okey)$\subset$ subgoals(ckey), subgoals(odate)$\subset$ subgoals(okey), subgoals(discount)$\subset$ subgoals(okey), and subgoals(odate) $\cap$ subgoals(discount)$= \emptyset$. The query variable ckey is maximal. $\square$

The tractable conjunctive queries without self-joins on tuple-independent databases are precisely the hierarchical ones [10]. The following result is fundamental to the evaluation of hierarchical queries and used for the generation of safe plans [10] and for query plan optimization in general [33].

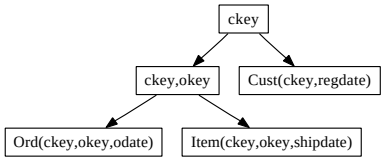


Figure 9: Tree representation of the hierarchical query of Example 3.12.

Proposition 3.13. *Hierarchical queries, whose head variables are maximal, are efficient-independent.*

We can allow restricted inequalities in hierarchical queries while still preserving the efficient-independent property.

Proposition 3.14. *Let* $Q(\overline{x_0})$*:-*$q_1(\overline{x_1}), \ldots, q_n(\overline{x_n}), \phi$ *be a query, where* $\forall 1 \leq i < j \leq n : \overline{x_0} = \overline{x_i} \cap \overline{x_j}$*, and* ϕ *has the max-one property over the disjoint variable sets* $\overline{x_1} - \overline{x_0}, \ldots, \overline{x_n} - \overline{x_0}$*. Then,* Q *is efficient-independent if* $q_1, \ldots, q_n$ *are efficient-independent.*

This class can be intuitively described using a tree representation of hierarchical queries, where the inner nodes are the common (join) variables of the children and the leaves are query subgoals [30]. The root is the set of maximal variables. Each inner node corresponds to a hierarchical subquery where the head variables are the node's variables. Such subqueries are efficient-independent because their head variables are maximal by construction. Figure 9 shows the tree representation of a hierarchical query. Proposition 3.14 thus states that a hierarchical query with maximal head variables remains efficient-independent even if max-one inequalities are allowed on the children of its tree representation.

Example 3.15. The addition of the inequality $shipdate > odate$ to the query of Example 3.12 preserves the efficient-independent property. We first join Ord and Item:

$$q'(ckey, okey)\text{:-Ord}(ckey, okey, odate), \text{Item}(ckey, okey, shipdate), \\ shipdate > odate$$

According to Proposition 3.14, q' is efficient-independent. We then join q' and Cust and obtain our query, which according to Proposition 3.14, is also efficient-independent. □

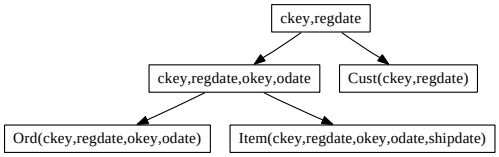


Figure 10: Tree representation of the FD-reduct of Example 3.18.

3.1.4 Database Constraints

Our last extension considers queries with inequalities that, under functional dependencies (fds) that hold on probabilistic relations, can be rewritten into IQ queries whose subgoals are efficient-independent subqueries. We use an adaptation of the rewriting framework of our previous work [33].

Definition 3.16. *Given a set of fds Σ and a query*

$$Q(\overline{x_0})\text{:-}R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \phi$$

where ϕ is a conjunction of inequalities. Then, the FD-reduct *of Q under Σ is the query*

$$Q_{fd}\text{:-}R_1(CLOSURE_\Sigma(\overline{x_1}) - CLOSURE_\Sigma(\overline{x_0})), \ldots, \\ R_n(CLOSURE_\Sigma(\overline{x_n}) - CLOSURE_\Sigma(\overline{x_0})), \phi'$$

where ϕ' is obtained from ϕ by dropping all inequalities on variables in $CLOSURE_\Sigma(\overline{x_0})$.

Similar to the case of hierarchical FD-reducts [33], it follows from the chase procedure that

Proposition 3.17. *If there is a sequence of chase steps under fds Σ that turns a query into an IQ query with efficient-independent subgoals, then the fixpoint of the chase, i.e., the FD-reduct, is such an IQ query.*

Example 3.18. Consider a modified version of the query of Example 3.12, which asks for the likelihood of items shipped with delay to old customers (see query 2 of the experiments):

$$Q'\text{:-}\mathrm{Cust}(ckey, 'Joe', regdate), \mathrm{Ord}(okey, ckey, odate), \\ \mathrm{Item}(okey, shipdate), regdate < odate < shipdate$$

Q' is not in our tractable query class because it has equality joins and yet is not hierarchical. In case ckey and okey are keys in Cust and Ord

respectively, the FD-reduct becomes

$$Q'\text{:-}\mathrm{Cust}(ckey,'Joe',regdate),\mathrm{Ord}(okey,ckey,odate,regdate),\\ \mathrm{Item}(okey,shipdate,ckey,regdate,odate),\\ regdate < odate < shipdate.$$

Query Q' is hierarchical and can be represented as in Figure 10. The subquery corresponding to any of the tree nodes is efficient-independent. In particular, each inequality is expressed on variables of the same node. □

Proposition 3.19. *The queries, whose FD-reducts are IQ queries with efficient-independent subgoals, can be computed on tuple-independent databases in polynomial time.*

3.2 OBDD-Based Query Evaluation

This section shows that the lineage of IQ queries on any tuple-independent database can be compiled into OBDDs of size polynomial in the number of variables in the lineage.

We first study the class of IQ queries with inequality paths. Here, the OBDDs have size *linear* in the number of variables in the lineage. We then investigate the more general subclass of IQ queries with inequality trees, in which case the OBDDs still have sizes *linear* in the number of variables in the lineage, but with a constant factor that depends exponentially on the size of the inequality tree. Cases of IQ queries with inequality graphs are not discussed in this thesis. A technique processing IQ queries with inequality graphs is introduced in [31] and its extension for less restricted classes of queries is in [40].

Remark 3.20. Our confidence computation algorithms are designed for IQ queries, but are applicable to a larger class of queries with efficient-independent subqueries (see Section 3.1).

The straightforward approach to deal with such subqueries is to materialize them. A different strategy is to consistently use OBDDs for the evaluation of the most general tractable query class of Section 3.1, which is that of IQ queries with efficient-independent subqueries. This approach is subject to future work. We note that previous work of the authors [30] showed that the hierarchical queries without self-joins, which represent together with the IQ characterization ingredients of efficient-independent subqueries, also admit linear-size OBDDs. □

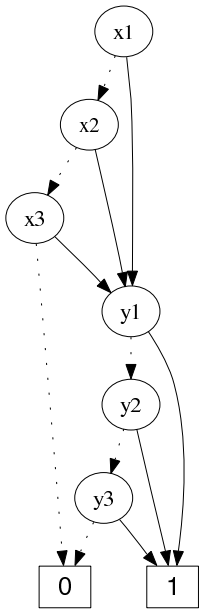


Figure 11: OBDD used in Example 3.21.

3.2.1 Independent Subqueries

Before we start with inequalities, a note on IQ queries that are products of independent subqueries is in place here. The lineage of such queries can be expressed as the product of the independent lineage of each of the subqueries. For OBDDs, product of independent lineage is expressed as concatenation of their OBDDs. We next exemplify with a simple query, but the same OBDD construction applies to any query with independent subqueries.

Example 3.21. Let the query $Q\text{:-}R(A), T(D)$ on the database (R, T) of Figure 5. The lineage consists of one clause for each pair of variables from R and of variables from T:

$$(x_1 + x_2 + x_3)(y_1 + y_2 + y_3).$$

An interesting variable order is $(x_1x_2x_3)(y_1y_2y_3)$, i.e., a concatenation of variable orders for the sums of variables in R and in T, respectively. The OBDD, shown in Figure 11, is then the concatenation of the OBDDs for the two sums. □

We next consider only IQ queries whose subgoals are connected by inequality joins.

3.2.2 Queries with Inequality Paths

We study the structure of the OBDDs for IQ queries with inequality paths. Examples of queries in this subclass are

$$Q_5\text{:-}R(A), T'(G, H), A < H$$
$$Q_6\text{:-}R'(E, F), T(D), T'(G, H), E < D < H$$

In general, queries in this subclass have the form

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

The lineage of such queries follows the inequalities on query variables. If table R_1 is sorted (ascendingly) on the attribute mapped to X_1, then the tuples of R_2, which are joined with the $i{+}1st$ tuple of R_1, are also necessarily joined with the ith tuple of R_1 because of the transitivity of inequality. This means that if the sorted table R_1 has variables $x_1, \ldots, x_k$, we can express the lineage as $\Sigma_i x_i f_{x_i}$, where the cofactor f_{x_i} of x_i includes the cofactor $f_{x_{i+1}}$ of x_{i+1}. This property holds for the relationship between the variables of any pair of tables that are involved in inequalities in Q.

The OBDDs are very effective at exploiting the overlapping between the cofactors. We can easily find a variable order for the cofactor f_{x_i} such that its OBDD already includes the OBDDs of the cofactors f_{x_j} of all variables x_j where $j > i$. This is because the clauses of $f_{x_{i+1}}$ are also clauses of f_{x_i} - we write this syntactically as $f_{x_i} \supseteq f_{x_{i+1}}$. We can obtain $f_{x_{i+1}}$ from f_{x_i} by setting to false variables that occur in f_{x_i} and not in $f_{x_{i+1}}$. The variable order for f_{x_1} must then agree with constraints on variable elimination orders imposed by migrating from f_{x_i} to $f_{x_{i+1}}$, for all $i \geq 1$. Such a variable order eliminates the variables of each table R_i in the order they occur after sorting that table, and before the variables of table R_{i+1}.

Computing such a variable order can then be done very efficiently. Under this variable order, the OBDD representations for the cofactors f_{x_j}, where $j > 1$, are obtained for free, once we computed the OBDD for f_{x_1}.

Example 3.22. Example 2.3 discusses how the lineage of the query Q_5 above on the database (R, T') of Figure 5 can be compiled into an OBDD of linear size.

We next discuss the case of the query Q_6:

$$Q_6\text{:-}R'(E, F), T(D), T'(G, H), E < D < H.$$

Consider the probabilistic database (R', T, T') of Figure 5, where the variables of T' are z_1, z_2, z_3 instead of y_1, y_2, y_3. The lineage of the answer to

query Q_6 on this database is

$$\begin{array}{lr} x_1[y_1(z_1+z_2+z_3)+ & y_2(z_2+z_3)+y_3z_3]+ \\ x_2[& y_2(z_2+z_3)+y_3z_3]+ \\ x_3[& y_3z_3]. \end{array}$$

We can check that the inclusion relation holds between the cofactors of variables x_i: $f_{x_1} \supset f_{x_2} \supset f_{x_3}$. The same applies to the cofactors of variables y_i. Although it is not here the case, in general the inclusion may not be strict. That is, two variables may have the same cofactor. For instance, if two tuples of R' have the same E-value, then their variables have the same cofactors.

The above lineage can be easily compiled into an OBDD of size linear in the number of variables in the lineage, see Figure 8(right). We first eliminate the variables x_1, x_2, x_3, and then reduce the cofactor f_{x_1} to f_{x_2} by eliminating variable y_1, and then to f_{x_3} by eliminating y_2. The variable order of our OBDD has then y_1 before y_2 before y_3. Note that the variables y_1 and z_1 are those that occur in f_{x_1} and not in f_{x_2}, although to get from f_{x_1} to f_{x_2} we only need to set y_1 to false. The same applies to variables y_2 and z_2.

After removing y_1, the branch $y_1 = 0$ points to f_{x_2}, and the other branch $y_1 = 1$ points to $z_1 + z_2 + z_3$. After removing y_2, we point to f_{x_3} and to $z_2 + z_3$. In case of y_3, we point to 0 and to z_3. The sums $z_1 + z_2 + z_3$, $z_2 + z_3$, and z_3 can be represented linearly under the variable order $z_1 z_2 z_3$, because $(z_1 + z_2 + z_3) \supset (z_2 + z_3) \supset z_3$. □

We can now summarize our results on inequality paths.

Theorem 3.23. *Let ϕ be the lineage of any IQ query with inequality paths on any tuple-independent database. Then, we can compute a variable order π for ϕ in time $O(|\phi| \cdot \log |\phi|)$ under which the OBDD (ϕ, π) has size bounded in $|Vars(\phi)|$ and can be computed in time $O(|Vars(\phi)|)$.*

We thus obtain linear-size OBDDs for lineage whose size can be exponential in the query size. This result supports our choice of OBDDs as a data structure that can naturally capture the regularity in the lineage of tractable queries.

Assumptions:
Input *tree* is the query's inequality tree and has n nodes.
Input t is the query answer before confidence computation.
For each node in *tree*, tuples in t have its column X involved in inequalities and the variable column V of its table.

processLineage(IneqTree *tree*, Tuples t) {
  assign indices {1,2,...,n} to each node in *tree* according to its position in a depth-first preorder traversal;
  sort t on (X_1 desc, V_1, ..., X_n desc, V_n), where X_i and V_i are from the table of node with index i in *tree*;
  let t' be $\pi_{V_1,V_2,...,V_n}$(sorted t);

  crtTuple = first tuple in t';
  varOrder = NULL;

  foreach node no of *tree* **do** {
   no.**firstVar** = crtTuple[$V_{no.index}$];
   no.**latestVarInVO** = NULL;
   no.**varToInsert** = crtTuple[$V_{no.index}$]; }

  { nextTuple = next tuple in t';

    find minimal i such that crtTuple[V_i] $\neq$ nextTuple[V_i];

    foreach node no of *tree* with index from n to i **do** {
     if (no.**varToInsert** $\neq$ NULL) {
      insert no.**varToInsert** at the beginning of varOrder;
      no.**latestVarInVO** = no.**varToInsert**;
      no.**varToInsert** = NULL; }

     if (crtTuple[$V_{no.index}$] $\neq$ nextTuple[$V_{no.index}$] **AND**
         crtTuple[$V_{no.index}$] = no.**latestVarInVO AND**
         nextTuple[$V_{no.index}$] $\neq$ no.**firstVar**)
      no.**varToInsert** = nextTuple[$V_{no.index}$];
    }

    crtTuple = nextTuple;
  } **do while** (crtTuple $\neq$ NULL);
}

Figure 12: Incremental computation of variable orders for IQ queries with inequality trees.

3.2.3 Queries with Inequality Trees

We generalize the results of Section 3.2.2 to the case of inequality trees. Examples of such IQ queries are:

$$Q_7 :\text{-} R'(E,F), T(D), S(B,C), E < D, E < C \text{ and}$$
$$Q_8 :\text{-} R'(E,F), T(D), S(B,C), T'(G,H), E < D, E < C < H.$$

The lineage of queries with inequality paths and of queries with inequality trees have different structures. We explain using the lineage of query Q_7, where we assume that table R' has variables $x_1, \ldots, x_n$, table T has variables $y_1, \ldots, y_m$, and table S has variables $z_1, \ldots, z_k$, and that the tables are already sorted on their attributes involved in inequalities. We will later exemplify with a concrete database. As for inequality paths, the lineage can be expressed as $\Sigma_i x_i f_{x_i}$, but now each cofactor f_{x_i} of x_i is a product of a sum of variables y_i and of a sum of variables z_j. In contrast, for an inequality path $E < D < C$, a cofactor f_{x_i} would be a sum of variables y_j, each with a cofactor f_{y_j} that is a sum of z-variables.

The inclusion relation still holds on the cofactors of variables x_i: $f_{x_1} \supseteq \ldots \supseteq f_{x_n}$, and we can thus obtain any $f_{x_{i+1}}$ from f_{x_i} by setting to false variables that occur in f_{x_i} and not in $f_{x_{i+1}}$. These variables can be both y-variables and z-variables; to compare, in the case of inequality paths, the elimination variables need only be y-variables. The inclusion relation holds because of the transitivity of inequality: If we consider any two E-values e_1 and e_2 such that $e_1 < e_2$, the tuples of T and S joined with e_2 are necessarily also joined with e_1. The inclusion relation holds even if the variables y_i or z_j have themselves further cofactors due to further inequalities, provided the cofactors of variables y_i are independent from the cofactors of variables z_j.

Example 3.24. Consider the database consisting of tables R', T, and S of Figure 5, where we add variables z_1 to z_5 to the tuples of table S. The lineage of query Q_7 is

$$x_1(y_1 + y_2 + y_3)(z_1 + z_3 + z_2 + z_5 + z_4) +$$
$$x_2(y_2 + y_3)(z_2 + z_5 + z_4) +$$
$$x_3(y_3)(z_4).$$

It indeed holds that $f_{x_1} \supset f_{x_2} \supset f_{x_3}$. We can transform f_{x_1} into f_{x_2} by eliminating in any order y_1 and (z_1, z_3). We then transform f_{x_2} into f_{x_3} by eliminating y_2 and (z_2, z_5). According to the elimination order constraints imposed by transformations on cofactors, an interesting variable order is $x_1 x_2 x_3 y_1 z_1 z_3 y_2 z_2 z_5 y_3 z_4$. Figure 15 gives a fragment of the OBDD for this lineage in case x_1 is set to false. As we can see, each variable x_i has one

OBDD node, and each variable y_i or z_i has up to two OBDD nodes. This is because the lineage states no correlation between the truth assignments of any pair of variables y_i and z_j. Hence, in case we eliminate, say, y_i, nodes for variable z_j can occur under both branches of the y_i node.

There are, of course, other variable orders that do not violate the constraints. For instance, we could eliminate $y_1 z_1 z_3$ after x_1 and before x_2, and similarly for $y_2 z_2 z_5$: We then obtain $x_1 y_1 z_1 z_3 x_2 y_2 z_2 z_5 x_3 y_3 z_4$. The reverse of any such order also induces succinct OBDDs. □

As in the case of inequality paths, we can always find a variable order for the cofactor f_{x_1} such that its OBDD already includes the OBDDs of the cofactors f_{x_i} of all variables x_i where $i > 1$. This order must agree with constraints on variable elimination orders imposed by transforming f_{x_i} into $f_{x_{i+1}}$, for all $i \geq 1$. Example 3.24 (above) shows how such orders can be computed for a reasonably small lineage. For the case of general IQ queries with inequality trees, one can use the algorithm given in Figure 12.

This algorithm works on a relational encoding of the lineage (as produced by queries) and, after sorting the lineage, it only needs one scan. It uses the inequality tree to re-discover the structure of the lineage. Because of the max-one property of the conjunction of inequalities, there is one table for each node in the inequality tree. Each node in the inequality tree contains four fields: **index**, **firstVar**, **latestVarInVO** and **varToInsert**. The field **index** serves as an identifier of the node, whereas the other three fields store information related to the variables from the corresponding table. The field **firstVar** stores the first variable from the table that has been inserted into the variable order. It is set as the variable in the first tuple after sorting and does not change afterwards. The field **latestVarInVO** stores the latest variable from the table inserted into the variable order, and the field **varToInsert** stores the new variable encountered in the input tuples but not yet inserted into the variable order.

The variable order construction is triggered by the changes in the variable columns between two consecutive tuples. The sorting is crucial to the algorithm, as it orders the lineage such that the cofactor $f_{x_{i+1}}$ is encountered before f_{x_i} in one scan of the lineage. We thus compute the variable order for $f_{x_{i+1}}$ before computing it for f_{x_i}. Because the OBDD for $f_{x_{i+1}}$ represents a subgraph of the OBDD for f_{x_i}, the variable order for $f_{x_{i+1}}$ is a suffix of the variable order for f_{x_i}. A key challenge here is to identify a variable that has not been inserted into the variable order. An inefficient approach is to look it up in the variable order constructed so far. This can be solved more efficiently, however, by only using **firstVar** and **latestVar-**

InVO. Due to sorting and the inclusion property between the cofactors of variables from the same table, all variables encountered after **firstVar** and before **latestVarInVO** while scanning the cofactor of a variable have already been inserted into the variable order. After scanning the cofactor of **latestVarInVO**, if the next variable in the same column of the next tuple is not **firstVar**, this indicates that this variable has not been encountered and we store it in **varToInsert**.

Example 3.25. Consider the lineage of Example 3.24. We scan it in the order $x_3y_3z_4$, $x_2y_3z_4$, $x_2y_3z_5$, $x_2y_3z_2$ and so on. Initially, the first tuple is read and **firstVar** and **varToInsert** are set to the variables in this tuple. On processing the second tuple, a change is found on the first variable column, all **varToInsert** values stored in the nodes are inserted into the variable order and obtain $x_3y_3z_4$. The fields **latestVarInVO** of nodes for query variables E, D, and C are also updated accordingly to x_3, y_3, and z_4 respectively. On reading the third tuple, a change in the third variable column is detected and **varToInsert** is updated to z_5. On reading the fourth tuple, a change is again detected in the third variable column and z_5 is inserted into the variable order. We thus obtain the order $z_5x_3y_3z_4$, In addition, **latestVarInVO** is updated to z_5 and **varToInsert** is set to z_2. The final variable order is $x_1y_1z_1z_3x_2y_2z_2z_5x_3y_3z_4$. □

Under such variable orders, the OBDDs can have several nodes for the same variable. As pointed out in Example 3.24 for the lineage of query Q_7, this is because there is no constraint between variables y_i and z_j: Setting a variable y_i to true or false does not influence the truth assignment of a variable z_j.

We next analyze the maximum number of OBDD nodes for a variable in case of an inequality tree consisting of a parent with n children:

$$Q_8 \text{:-} R(X), S_1(Y_1), \ldots, S_n(Y_n), X < Y_1, \ldots, X < Y_n$$

where R and S_i have variables $x_1, \ldots, x_m$, and $y_1^i, \ldots, y_{m_i}^i$, respectively. The lineage is $\Sigma_i x_i(\prod_{j=1}^{n} f_{x_i}(y^j))$, where each $f_{x_i}(\cdot)$ is a sum of variables from the same table. We know that the OBDD obtained by compiling the cofactor of x_1 contains the OBDDs for the cofactors of all other x_i $(i > 1)$, under the constraint that the variable order transforms one cofactor into the next. This means that, in order to compile the cofactor of x_1, we need to use an intertwined elimination of variables y^1 to y^n.

Consider we want to count the number of OBDD nodes for variable y_j^i. The OBDD nodes that can point to y_j^i-nodes represent expressions that can

have any of the form $s(y^i) \prod_{k=1,\neq i}^{l\leq n} s(y^k))$ or simply $s(y^i)$, where the functions $s(y^k)$ stand for sums over variables $y^k_1, \ldots, y^k_{m_k}$. All such expressions necessarily contain a sum over variables y^i, which includes y^i_j; otherwise, their OBDD nodes cannot point to y^i_j-nodes. The number of distinct forms is exponential in n (more precisely, half the size of the powerset of $\{1, \ldots, n\}$; those without $s(y^i)$ are dropped). Interestingly, there can be precisely one OBDD node for each of these forms that point to an y^i_j-node. This means that the number of y^i_j-nodes is in the order of $O(2^n)$. We explain this for three of the possible forms. A node representing an expression $s(y^i)$ can point to an y^i_j-node if, according to our elimination order, the variables y^i preceding y^i_j are all dropped, and precisely one variable from the remaining variable groups is set to true. Then, there is a single path from the OBDD root to the node for the expression $s(y^i)$, following the true and false edges of the eliminated variables, and hence only one y^i_j-node to point to. A node representing an expression $s(y^i) \prod_{k=1,\neq i}^{n} s(y^k))$, which is a product of sums of variables from each variable group, can point to y^i_j-nodes only if the sum $s(y^i)$ contains the variable y^i_j and all its preceding variables y^i are set to false. Then, again, there is one path from the root to that node that follows the false edges of the eliminated variables y^i, and hence one y^i_j-node to point to. In case of expression forms, where some of the sums are missing, we use the same argument: precisely one variable from each of these sums is set to true, and there is a single path from the root to the node representing that expression, and hence one y^i_j-node to point to. This result can be generalized to arbitrary inequality trees.

Theorem 3.26. *Let ϕ be the lineage of any IQ query with inequality tree t on any tuple-independent database. Then, we can compute a variable order π for ϕ in time $O(|\phi| \cdot \log |\phi|)$, under which the OBDD (ϕ, π) has size and can be computed in time $O(2^{|t|} \cdot |Vars(\phi)|)$.*

The OBDD for ϕ does not need all $O(2^{|t|})$ nodes for each variable. Figure 15 shows two OBDDs for a fragment of the lineage of query Q_7 of Example 3.24, one as constructed by our algorithm (right), and a reduced version of it (left).

3.3 Confidence Computation in Secondary Storage

This section introduces a secondary-storage algorithm for confidence computation for *IQ* queries with inequality trees.

OBDD_Node: probability **p**, bool vector **bv**, children **hi** from the solid edge and **lo** from the dotted edge
Level: a vector of OBDD_Nodes **nodes**, index of the inequality tree node whose table contains the variables at this level **index**
n: the number of nodes in the inequality tree
obdd_levels: **Level**[**n** + 1]

Figure 13: Data structures used by the algorithm of Figure 14.

This algorithm is in essence our algorithm for incremental computation of variable orders for queries with inequality trees given in Figure 12 of Section 3.2. An important property of this algorithm is that it does not require the OBDD to be materialized *before* it starts the computation. The key ideas are (1) to construct the OBDD levelwise, where a level consists of the OBDD nodes for one variable in the input lineage, and (2) to keep in memory only the necessary OBDD levels. Similar to the algorithm that computes variable orders, this confidence computation algorithm needs only one scan over the sorted lineage to compute its probability.

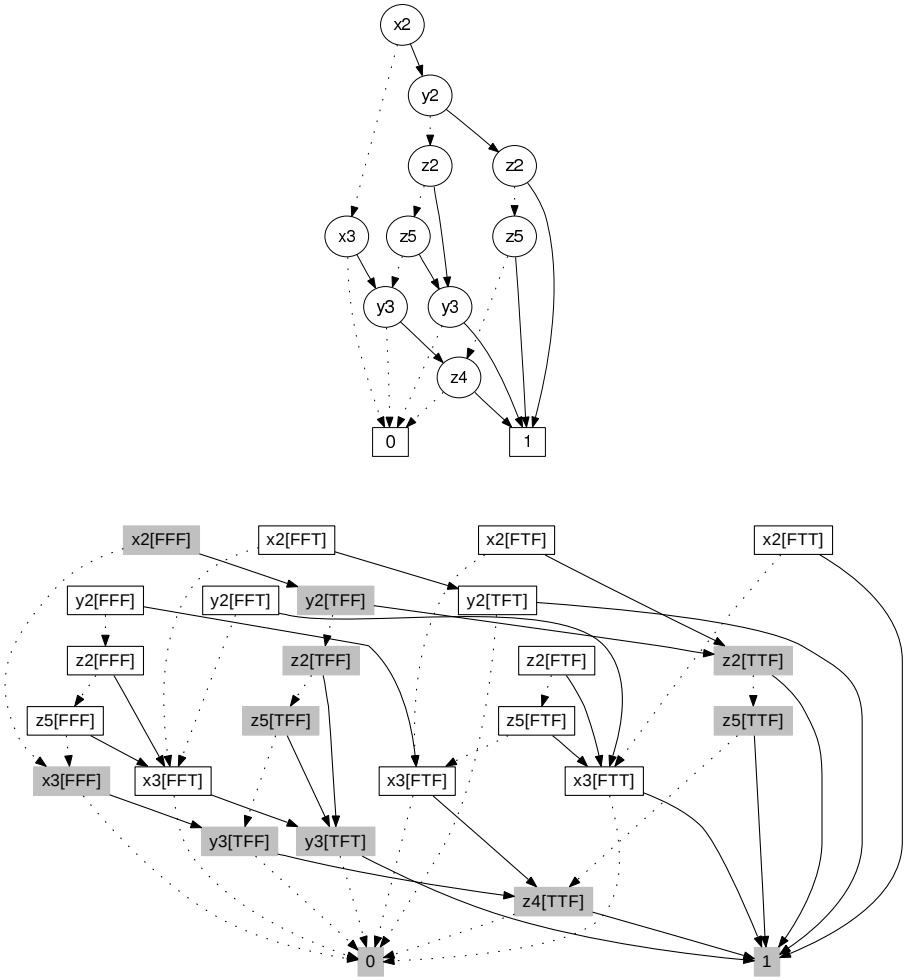
The algorithm is given in Figure 14 and uses data structures described in Figure 13: The code in the topmost box should replace the inner box of the algorithm for variable order computation given in Figure 12.

Let a query Q with inequality tree t of size n and let ϕ be the lineage of Q on some database. As discussed in Section 3.2.3, each variable in ϕ can have up to $2^{|t|}$ OBDD nodes, which form a complete OBDD level. When a new variable is encountered, instead of adding it to the variable order, we construct a new level of nodes in the OBDD for this variable. The major challenge lies in how to connect the low and high edges of a node to the correct (lower) nodes in the levels kept in memory. Recall that every OBDD node represents a partial lineage obtained by eliminating variables at the upper OBDD levels. In an OBDD, none of the nodes represent the same formula. The formulas determine the connection between nodes from different levels. For instance, in Figure 15, the left and right nodes in the level of y_3 represent formulas y_3z_4 and y_3 respectively. The formula at the leftmost node x_3 is $x_3y_3z_4$, and hence the high edge of this node must point to y_3z_4 and not to y_3.

The formula can be, however, large and, instead of materializing it, we use a compact representation of it. This is possible due to the lineage structure imposed by the query and the chosen variable elimination order. Our compact representation is that of a Boolean vector of size n. A "true" value

```
Code to replace the inner box in Figure 12:
    add_level(i, no.varToInsert.prob);

Initialization:
create Level L; L.nodes = OBDD_Node[2^n]; L.index = 0;
foreach OBDD_Node no in L.nodes do {
    no.bv = distinct bool vector of size n;
    if (any value in no.bv is false) no.p = 0;
    else no.p = 1; }
insert L at beginning of obdd_levels;

add_level(int i, Prob p)
create Level L;
L.nodes = OBDD_Node[2^(n-1)]; L.index = i;
foreach OBBD_node no in L.nodes do {
    no.bv = distinct bool vector of size n where bv[i] = false;
    no.lo = get_node(no.bv, i, false);
    no.hi = get_node(no.bv, i, true);
    no.p = p × no.hi.p + (1 − p) × no.lo.p; }
remove L′ from obdd_levels such that L′.index = i;
insert L at beginning of obdd_levels;

get_node(bool vector bv, int i, bool is_true)
bv[i] = is_true; crtLevel = the first Level in obdd_levels;
while(true)
    if (crtLevel.index = 0 OR (!bv[crtLevel.index]
      AND all_ancestors_set_true(bv, crtLevel.index)))
      foreach OBDD_node no in crtLevel.nodes do
        if (no.bv = bv) {
          bv[i] = false;
          return no; }
    else
      crtLevel = the next Level in obdd_levels;

all_ancestors_set_true(bool vector bv, int i)
foreach ancestor A of get_ineqtree_node_with_index(i) do
    if (!bv[A.index]) return false;
return true;
```

Figure 14: Secondary-storage algorithm for confidence computation.



Up: Reduced OBDD used in Section 3.2. Down: Partially reduced OBDD as constructed by the algorithm of Section 3.3.
Constant nodes are merged, and nodes z_4[FFF], z_4[FTF], z_4[TFF], y_3[FFT] and y_3[FFF] are removed for compactness (although the algorithm constructs them). Grey nodes form the equivalent reduced OBDD on the left.

Figure 15: OBDDs for the lineage discussed in Example 3.24.

in this Boolean vector at position i indicates that a variable from the table with index i has been set to true. This means that the formula at that node does not contain further variables from the table with index i (property of OBDDs representing lineage of queries with inequality trees). The Boolean vectors act as the identifiers of the OBDD nodes so that the nodes from the level above can identify a potential child node among the ones at this level.

The algorithm works as follows: We initially build a level of one constant node 1 and $2^n - 1$ constant nodes 0. Note that this construction does not lead to reduced OBDDs, but it is sufficient for our processing task. For every new variable encountered during the scan, we build a level of OBDD nodes, and for each such node find its high and low children nodes in the existing levels. The probability of a node can be computed only based on the probabilities of the nodes it points to. Therefore, the probability computation and the OBDD construction go in parallel. Instead of keeping all the levels of the OBDD in memory, our algorithm keeps only one level for every table in the condition tree t. As soon as a new level is built for a variable from a table, the old level for the variable from the same table is dropped if there is any. This is possible because of the following property of the OBDDs for queries with inequality trees: Let x_1 and x_2 be variables from the same relation and the level of x_1 higher than the level of x_2. Then, no edge from levels above the level of x_1 points to nodes in the level of x_2. This is due to the inclusion property between the cofactors in the lineage.

The algoritm also exploits two properties of such OBDDs:

- No OBDD node for a variable from a table is accessible via the high edge of an upper OBDD node for a variable from the same table.
- Let a table R and its ancestors $S_1, \ldots, S_n$ in the inequality tree. Then, a path from the root to an OBDD node for a variable from R must follow the high edges of at least one node for a variable from each $S_1, \ldots, S_n$.

Both these properties are used in the outermost if-condition of the procedure get_node in Figure 14.

Example 3.27. We show how to compute the probability of the lineage of query Q_7 of Example 3.24. Figure 15 shows a fragment of its OBDD. The inequality tree is of size 3. The number of constant nodes is thus $2^3 = 8$ and of non-constant nodes per level is $2^2 = 4$. The size of a Boolean vector is 3 and the corresponding inequality tree nodes of tables R', T and S are assigned indices 1, 2 and 3 respectively.

We build a level of eight constant nodes: Seven nodes with at least one false value (F) in the Boolean vector have probability 0 (they are merged into one in Figure 15) and the remaining one with [T,T,T] has probability 1. We then construct four nodes in the level for z_4. The corresponding Boolean vectors of the nodes are [T,T,F], [T,F,F], [F,T,F], and [F,F,F]. The value in all the vectors for the variables of table S is F because formulas of all nodes at this level contain variable z_4; otherwise, elimination of z_4 will be redundant. The first vector encodes a formula with variables only from S. The second vector encodes a formula with variables only from T and S, and so on.

The outgoing edges of nodes z_4 can only point to the only level below, which is made by constant nodes. For instance, the high and low edges of node with vector [T,T,F] point to nodes with [T,T,T] and [T,T,F] respectively, that is, to nodes 1 and 0 respectively. Hence, its probability is $\Pr(z_4) \times 1 + \Pr(\overline{z_4}) \times 0 = \Pr(z_4)$.

Consider that the level for variable y_3 is already constructed similarly to the previous level (z_4), and let us construct the level for x_3. We create four nodes with Boolean vectors [F,T,T], [F,T,F], [F,F,T], [F,F,F]. For the node with vector [F,F,F], since the corresponding value for relation R' in the vector is F and the inequality tree node of R' is the ancestor of those of S and T, its low edge cannot point to nodes in y_3 and z_4 levels, but instead points to constant node with vector[F,F,F], namely node 0. Its high edge points to the node with vector [T,F,F] in the level for y_3. Therefore, its probability is $\Pr(x_3) \times$ (node with vector [T,F,F] in y_3 level).p + $\Pr(\overline{x}_3) \times$ 0.

Let us consider the final step. We construct the level for x_1. As the other non-constant node levels, it has four nodes. Since x_1 is the first to be eliminated in the OBDD, the corresponding values for R', S, and T in the Boolean vector of the root should be F. Therefore, the probability of the lineage is given by the value p of the node with vector [F,F,F] in the level x_1 and the other three nodes are redundant. Its probability is $\Pr(x_1)\times$ (node with [T,F,F] in y_1 level).p + $\Pr(\overline{x_1})\times$ (node with [F,F,F] in x_2 level).p. □

4 Lineage Decomposition Using D-Trees

In Section 3, our discussion focuses on conjunctive queries with inequalities and without self-joins in tuple-independent probabilistic databases, while in Sections 4 and 5, we target positive relational algebra in complete representation system, less restricted queries in a more general data model. Section 4.1 introduces a novel type of decision diagrams called *d-trees* for lineage decomposition. Section 4.2 shows how d-trees compactly represent the lineage of all currently known tractable queries on tuple-independent probabilistic databases with a variable elimination heuristic.

4.1 Compiling DNFs into D-Trees

Computing the probability of a formula is #P-hard. In general, there is no efficient way of computing the probability $P(\phi \wedge \psi)$ or $P(\phi \vee \psi)$ from $P(\phi)$ and $P(\psi)$. However, there are important special cases in which this is feasible, in particular,

- if ϕ and ψ are independent, then

$$\begin{aligned} P(\phi \wedge \psi) &= P(\phi) \cdot P(\psi) \\ P(\phi \vee \psi) &= 1 - (1 - P(\phi)) \cdot (1 - P(\psi)) \end{aligned}$$

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

$$P(\phi \vee \psi) = P(\phi) + P(\psi).$$

We will use explicit notation to mark such $\wedge$ and $\vee$-operations: We will use $\otimes$ for independent-or, $\odot$ for independent-and and $\oplus$ for exclusive-or.

Example 4.1. *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 formula* $(x \otimes y) \odot ((z \odot u) \oplus (\neg z \odot v))$. *The probability of this formula thus is* $\big(1 - (1 - P(x)) \cdot (1 - P(y))\big) \cdot \big(P(z) \cdot P(u) + P(\neg z) \cdot P(v)\big)$. □

For convenience, we also use the Boolean combinators on sets of formulae; i.e., we write $\bigwedge \Phi$ for $\phi_1 \wedge \cdots \wedge \phi_n$ if $\Phi = \{\phi_1, \ldots, \phi_n\}$ and analogously $\bigvee \Phi$, $\bigotimes \Phi$, $\bigoplus \Phi$, and $\bigodot \Phi$. (All the operations are associative, and computing the probabilities of formulae using these set operations is straightforward.)

Definition 4.2. *A (partial)* d-tree *(for decomposition tree) is a formula constructed from $\otimes$, $\oplus$, $\odot$ and nonempty DNFs (as "leaves"). A d-tree in which each DNF is a singleton – i.e., contains a single clause – is called a* complete d-tree. *Given a partial d-tree, the d-tree obtained by replacing a leaf DNF by an equivalent partial d-tree is called a* refinement. □

Thus, in a d-tree (viewed as a parse tree of the d-tree formula), an $\wedge$ or $\vee$ node never occurs above a $\oplus$, $\otimes$, or $\odot$ node.

It follows from the definitions of $\oplus$, $\otimes$, and $\odot$ that

Proposition 4.3. *Given the probabilities of all the DNF leaves of a partial d-tree, its probability can be computed in linear time.* □

Since computing the probability of a clause is straightforward, the probability of a complete d-tree can be computed in time linear in its size.

Next we present an algorithm for computing a complete (or, if we stop the compilation early, a partial) d-tree from a DNF. For this purpose, we assume a DNF is represented by a set of sets of atomic formulae. In essence, the algorithm repeatedly applys three decomposition methods that correspond to the three types of inner nodes in a d-tree $\oplus$, $\otimes$, and $\odot$:

- Independent-or $\otimes$: Partition Φ into independent DNFs $\Phi_1, \Phi_2 \subset \Phi$ such that Φ is equivalent to $\Phi_1 \vee \Phi_2$.
- Independent-and $\odot$: Partition Φ into independent DNFs $\Phi_1, \Phi_2 \subset \Phi$ such that Φ is equivalent to $\Phi_1 \wedge \Phi_2$.
- Exclusive-or $\oplus$: Choose a variable x in Φ. Replace Φ by

$$\bigoplus_{a \in \mathrm{Dom}_x, \Phi|_{x=a} \neq \emptyset} \left(\{\{x = a\}\} \odot \Phi \mid_{x=a} \right)$$

  where $\Phi \mid_{x=a}$ denotes the DNF obtained from Φ by removing all clauses $\phi \in \Phi$ for which $\phi \wedge (x = a)$ is inconsistent and (syntactically) removing the atomic formula $x = a$ from the remaining clauses in which it occurs. Thus, obviously, $(x = a) \wedge \Phi$ is equivalent to $(x = a) \wedge \Phi \mid_{x=a}$.

Figure 16 sketches our general compilation approach, which will be refined in the next sections. Here, we consider that the compilation is exhaustive, i.e., the leaves of the d-tree only hold DNFs that are singleton clauses. If approximate probabilities are sought for, however, the compilation need not be exhaustive and the leaves can hold larger DNFs.

Compile (DNF Φ with $\Phi \neq \emptyset$) returns d-tree

if $(\emptyset \in \Phi)$ **then return** $\{\emptyset\}$

1. remove all subsumed clauses Φ :

   foreach $s, t \in \Phi$ **such that** $s \neq t$ **do**

   if $(s \subset t)$ **then** $\Phi := \Phi - \{t\}$

2. apply independent-or:

   if there are non-empty and pairwise indep. DNFs $\Phi_1, \ldots, \Phi_{|I|}$ **such that** $\Phi = \Phi_1 \cup \ldots \cup \Phi_{|I|}$

   then return $\bigotimes_{i \in I} \big(\text{Compile}(\Phi_i)\big)$

3. apply independent-and:

   if there are non-empty and pairwise indep. DNFs $\Phi_1, \ldots, \Phi_{|I|}$

   such that Φ is equivalent to $\Phi_1 \wedge \ldots \wedge \Phi_{|I|}$

   then return $\bigodot_{i \in I} \big(\text{Compile}(\Phi_i)\big)$

4. apply Shannon expansion:

   choose a variable x in Φ;

   $T := \{\phi \mid \phi \in \Phi, \nexists a \in \text{Dom}_x : (x = a) \subseteq \phi\}$;

   $\forall a \in \text{Dom}_x : \Phi\,|_{x=a} := \big\{\{y_1 = b_1, \ldots, y_m = b_m\} \mid \{x = a, y_1 = b_1, \ldots, y_m = b_m\} \in \Phi\big\} \cup T$;

   return $\bigoplus_{a \in \text{Dom}_x, \Phi|_{x=a} \neq \emptyset} \big(\{\{x = a\}\} \odot \text{Compile}(\Phi\,|_{x=a})\big)$

Figure 16: Compiling DNFs into d-trees.

Example 4.4. *Figure 17 shows a DNF and the complete d-tree obtained by executing the algorithm of Figure 16 to completion.* □

This algorithm is correct:

Proposition 4.5. *If Φ is a DNF, then Φ is equivalent to Compile(Φ).*

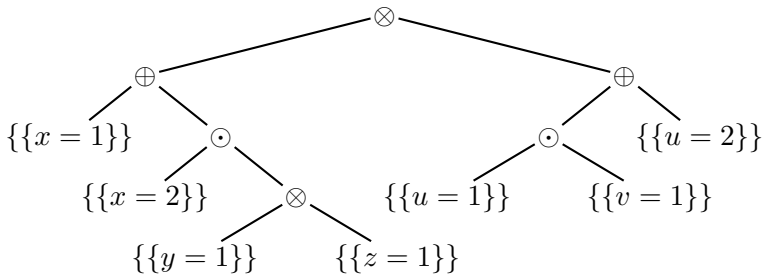


Figure 17: D-tree of DNF $\Phi = \{\{x = 1\}, \{x = 2, y = 1\}, \{x = 2, z = 1\}, \{u = 1, v = 1\}, \{u = 2\}\}$.

Both independent-or and independent-and partitioning tasks can be done efficiently. The first one is finding connected components in the dependency graph of the input DNF Φ, which consists of a node for each variable of Φ and, for each clause $\bigwedge_{i=1}^{n} x_i = a_i$ of Φ, of the edges (x_i, x_{i+1}) for $1 \leq i < n$. This can be done in time linear in the size of the Φ (using a well-known depth-first algorithm for computing strongly connected components, Tarjan's algorithm).

The independent-and partitioning is a special algebraic factorization of DNFs, and requires time $O(m \cdot n \cdot \log n)$, where n and m are the sizes of the DNF and of the constituent clauses, respectively [34].

The order of the variable choices needed for Shannon expansion (also known as variable elimination in e.g. the Davis-Putnam algorithm) greatly influences the size of the d-tree. In general, the compilation of a DNF creates a d-tree of exponential size, and an important challenge is to find compilation strategies that lead to decision diagrams of small sizes [13, 25]. We introduce a heuristics for choosing variables in Section 4.2, and we will resort to approximation of probabilities by compiling DNFs only into partial d-trees, upper- and lower-bounding the residual DNFs.

Remark 4.6. *D-trees are a generalization of the ws-trees of [25] in three ways: We have decoupled the exclusive-or operation from our algorithm's method of obtaining exclusive-or decompositions (variable elimination). Moreover, we have added independent-and decompositions, which are crucial for application of d-trees to hierarchical queries in the next section. Finally, we have generalized the formalism to partial decompositions, which are the foundation of the approximation techniques of Section 5. The probabilistic and/xor trees of [27] are modeled on the ws-trees but are a weaker represen-*

tation system in that they have tuples, rather than clauses, at their leaves. In addition, decomposition using variable elimination is reminiscent of the Davis-Putnam SAT solving algorithm [14] and OBDDs (cf. e.g. [30]). □

4.2 From Tractable Queries to Linear-Size D-Trees

D-trees can represent compactly DNFs associated with answers to known tractable queries on tuple-independent probabilistic databases. In such a database, each tuple is associated with a distinct random Boolean variable.

The classes of queries considered here are (1) the tractable queries without self-joins [10], (2) queries that are "hard" in general, but become tractable on restricted databases [33], and (3) tractable queries with inequalities [31].

We first introduce a correspondence between a factored form of formulae and d-trees. A formula Φ is in *one-occurrence normal form (1OF)* if any variable of the formula occurs exactly once in Φ [30]. For instance, Φ of Figure 2 is in 1OF. Such formulae can be arbitrarily nested using $\wedge$ and $\vee$, e.g., $((x_1 \vee x_2) \wedge (\neg y_1 \vee y_2)) \vee (x_3 \wedge \neg y_3)$. Complete d-trees can naturally represent 1OF formulae by turning $\vee$ into $\otimes$ and $\wedge$ into $\odot$. It follows from our compilation scheme in Fig. 16 that

Proposition 4.7. *Any formula factorizable in 1OF can be compiled in polynomial time into a complete d-tree with one leaf per distinct variable and inner nodes $\otimes$ and $\odot$.* □

The DNFs associated with answers to any tractable conjunctive query without self-joins are factorizable in 1OF [30]. Such queries are called *hierarchical*, and can be easily defined using a Datalog notation, where joins are expressed by occurrences of query variables in several subgoals.

Definition 4.8 ([10])**.** *A conjunctive query is hierarchical if for any two non-head query variables, either their sets of subgoals are disjoint or one set is contained in the other.* □

Example 4.9. *The following queries are hierarchical:*

$$q_1 :\text{-} R_1(A,B), R_2(A,C)$$
$$q_2 :\text{-} R_1(A,B,C), R_2(A,B), R_3(A,D)$$ □

The query $q :\text{-} R(X), S(X,Y), T(Y)$ is the prototypical #P-hard query[10]. It is non-hierarchical since the sets of subgoals of X and Y overlap, yet one does not include the other.

The DNFs for hard queries are factorizable in 1OF for restricted tuple-independent databases. Due to lack of space, we only present here without

MAX_OCCUR: (DNF Φ with $\Phi \neq \emptyset$) returns variable
  $\mathcal{V}(\Phi) :=$ set of all variables in Φ;
  foreach $x \in \mathcal{V}(\Phi)$ **do**
    $\#(x) :=$ number of occurrences of x in Φ;
  choose variable $x \in \mathcal{V}(\Phi)$ **such that**
    $\forall x' \in \mathcal{V}(\Phi) : \#(x) \geq \#(x')$;
  return x;

Figure 18: Order of variable elimination used in Shannon expansion.

proof new tractable cases that basically exploit regularities in the structure of table S.

Theorem 4.10. *The DNFs of hard query patterns $R(X)$, $S(X,Y), T(Y)$ are factorizable in 1OF if each connected component of the bipartite graph of S is functional or complete.* □

This result generalizes an early tractability result obtained for hard patterns where functional dependencies hold on the *entire* table S [33].

Very recent work defines tractable queries with inequalities [31]. We only consider here the core tractable language of so-called IQ queries defined in that work.

Definition 4.11. *Let the disjoint sets of query variables $\overline{x_1}, \ldots, \overline{x_n}$. A conjunction of inequalities over these sets has the* max-one *property if at most one query variable from each set occurs in inequalities with variables of other sets.* □

Definition 4.12. *An* IQ query *has the form*

$$Q :\text{-} R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \Phi$$

where $R_1, \ldots, R_n$ are distinct tuple-independent tables, the sets of query variables $\overline{x_1}, \ldots, \overline{x_n}$ are pairwise disjoint, and Φ has the max-one property over these sets. □

Example 4.13. *The following are IQ queries*

$$q_1 \text{:-} R(E,F), T(D), T'(G,H), E < D < H$$
$$q_2 \text{:-} R'(E,F), T(D), S(B,C), E < D, E < C$$
$$q_3 \text{:-} R(A), T(D)$$
$$q_4 \text{:-} R(A), T(D), R'(E,F), T'(G,H), A < E, D < E, D < G \ \square$$

We compile DNFs of IQ queries using the variable elimination order MAX_OCCUR given in Figure 18, where a variable that occurs most frequently in the input DNF is chosen first. The following result summarizes the core observation of our previous work [31], now expressed using MAX_OCCUR.

Lemma 4.14. *Given a DNF Φ of an IQ query and variable $v = MAX\_OCCUR(\Phi)$. Then, $\Phi|_v$ subsumes Φ.* $\square$

This property is what makes IQ queries tractable. We exemplify with a query $q \text{:-} R(X), S(Y), X < Y$ on a database with random Boolean variables $x_1, \ldots, x_n$ in R and $y_1, \ldots, y_m$ in S ($n \geq m$). Assume wlog that the indices of these variables correspond to the sorting order of relations R and S. Then, $x_1 = \text{MAX\_OCCUR}(\Phi)$ in the DNF Φ of q and

$$\Phi|_{x_1} = \bigvee_j (y_j) \vee \bigvee_{1<i\leq n} (x_i \wedge \Phi|_{x_i})$$
$$\Phi|_{\neg x_1} = \bigvee_{1<i\leq n} (x_i \wedge \Phi|_{x_i}).$$

The formula $\bigvee_j (y_j)$ is a disjunction of all the variables in S that annotate Y-values that are greater than the X-value annotated by x_1. Following the semantics of the inequality join and the max-occur property of x_1, any other variable x_i can only be paired with a disjunction of a (non-necessarily strict) subset of variables in S, hence $\bigvee_j (y_j) \vee \bigvee_{1<i\leq n} (x_i \wedge \Phi|_{x_i}) = \bigvee_j (y_j)$. Both formulas $\Phi|_{x_1}$ and $\Phi|_{\neg x_1}$ can be decomposed using the same heuristic: Any variable y_j is chosen in $\Phi|_{\neg x_1}$, and variable x_2 (or some y_j) is chosen in $\Phi|_{x_1}$. This observation leads to the following result.

Theorem 4.15. *DNFs of IQ queries can be compiled in polynomial time into complete d-trees with one $\oplus$ node for each input atomic formula.* $\square$

5 Approximate Confidence Computation Using D-Trees

As discussed in Section 4.1, the exact confidence of a DNF can be easily computed following the DNF compilation into a complete d-tree. Such an exhaustive compilation is not practical in general. If an approximate confidence suffices, then we may only explore a few levels in a d-tree and approximate the probability at its leaves using efficient heuristics.

The key challenges addressed in this section are (i) the design of efficient and good heuristics for approximating the probability of DNFs at the leaves of d-trees, and (ii) the design of an efficient algorithm that can compute an approximate probability for a given DNF by incrementally refining its d-tree compiled form.

5.1 Lower and Upper Probability Bounds for DNFs

We next discuss how to compute lower and upper bounds of the probabilities of DNFs at the leaves of a d-tree without refining them. Figure 19 gives two heuristics that partition the input DNF Φ into a set of buckets such that the exact probability of each bucket can be computed efficiently. The lower and upper bounds of the exact probability of Φ are then computed as the maximum over the probabilities of the buckets, and the sum of probabilities of the buckets, respectively. Both bounds are correct: Assume that a bucket with the maximal probability is B_i. Since Φ is a set of clauses, $\Phi = B_i \vee \Phi'$. Since each clause in Φ has a non-null probability by definition, $P(B_i) \leq P(B_i \vee \Phi') = P(\Phi)$, and thus $P(B_i)$ is indeed a lower bound for $P(\Phi)$. To see why the sum of probabilities of the buckets is indeed an upper bound, consider the following cases. If the buckets are negatively correlated, then the probability of their disjunction is the sum of their probabilities. In case they are independent or positively correlated, then it follows by definition that the probability of their disjunction is at most the sum of their probabilities.

Let us now look closer at how the buckets are created. In case of the first heuristic (Independent), each bucket only contains pairwise independent clauses, and each such bucket is maximal, i.e., for a given bucket B there is no clause in Φ and not in B that is pairwise independent with each clause in B. The probability of each bucket can be computed efficiently, as shown in Figure 19. As there may be several possible minimal partitionings of Φ, we empirically noticed that the lower bound computed by this heuristic can be further improved by first sorting Φ descending on the marginal probability of

Independent (DNF Φ with $\Phi \neq \emptyset$) returns [Lower, Upper]

minimally partition Φ **into** $B_1 \vee \ldots \vee B_n$ **such that**

$\forall 1 \leq i \leq n, \forall d, d' \in B_i : d, d'$ are independent;

foreach bucket B_i **do**

$P(B_i) := 0$;

foreach clause $d \in B_i$ **do**

$P(B_i) := 1 - (1 - P(B_i)) \cdot (1 - P(d))$;

return $[\max_{i=1}^{n} P(B_i), \min(1, \sum_{i=1}^{n} P(B_i))]$;

Exact-k (DNF Φ with $\Phi \neq \emptyset$) returns [Lower, Upper]

$n := \lceil \frac{|S|}{k} \rceil$;

partition Φ **into** $B_1 \vee \ldots \vee B_n$ **such that**

$\forall 1 \leq i \leq n-1 : |B_i| = k$ **and**

$\forall d' \in (B_2 \vee \ldots \vee B_n), \forall d \in B_1 : P(d') \leq P(d)$;

foreach bucket B_i **do**

$P(B_i) := \sum_{i=1}^{|B_i|} (-1)^{i-1} \sum_{I \subseteq B_i, |I|=i} (\bigwedge_{d \in I} d)$;

return $[\max_{i=1}^{n} P(B_i), \min(1, \sum_{i=1}^{n} P(B_i))]$;

Figure 19: Computing lower and upper bounds for the probability of DNFs.

its clauses, and then constructing a bucket that contains the most probable clause and subsequent independent clauses. It turns out that this heuristic behaves very well for all of our experimental scenarios (see Section 7). This heuristic requires time quadratic in the size of the input DNF, the most expensive part being the minimal partitioning.

Example 5.1. *Let the DNF* $\Phi = c_1 \vee c_2 \vee c_3$*, where*

$$c_1 = (x \wedge y), c_2 = (x \wedge z), c_3 = v$$

and $P(x) = 0.3$, $P(y) = 0.2$, $P(z) = 0.7$, $P(v) = 0.8$. *One minimal partitioning of* Φ *is* $B_1 = c_1 \vee c_3$ *and* $B_2 = c_2$. *Then,*

$$P(B_1) = 1 - (1 - 0.06) \cdot (1 - 0.8) = 0.812, P(B_2) = 0.21.$$

The bounds are $L(\Phi) = P(B_1) = 0.812$ *and* $U(\Phi) = \min(1, 0.821{+}0.21) = 1$. *Another minimal partitioning can be obtained by first sorting the clauses descending on their marginal probabilities. Then,* $B_1 = c_2 \vee c_3$, $B_2 = c_1$, *and*

$$P(B_1) = 1 - (1 - 0.21) \cdot (1 - 0.8) = 0.842, P(B_2) = 0.06.$$

The new bounds are $L(\Phi) = 0.842$ *and* $U(\Phi) = 0.848$, *which approximates better the exact probability of 0.8456.* □

In case of the second heuristic (exact-k), each bucket contains k clauses (except possibly the last one), where k is chosen small (3-4). We then compute the probability of each bucket using the inclusion-exclusion formula, which can take time exponential in k. The first bucket consists of k clauses with top marginal probability. The reason for this special bucket is to ensure that the lower bound is computed based on the top-k most probable clauses.

The heuristic Exact-k requires time $O(k \cdot |\Phi| + \lceil \frac{|\Phi|}{k} \rceil \cdot 2^k)$, which is linear in the size of the DNF Φ for a fixed k.

Proposition 5.2. *Let* $[L_1, U_1]$ = *Independent(*Φ*) and* $[L_2, U_2]$ = *Exact-k(*Φ*) for a DNF* Φ. *It then holds that* $L_1 \leq P(\Phi) \leq U_1$ *and* $L_2 \leq P(\Phi) \leq U_2$. □

5.2 Lower and Upper Probability Bounds for D-Trees

The lower and upper bounds can be propagated from leaves to the root of the d-tree. For this, we make use of the observation that the formulas for probability computation of each decomposition type, are monotonically increasing. (A function is monotonically increasing if for all x and y such that $x \leq y$, one has $f(x) \leq f(y)$.) If some of the children of an inner node ($\otimes$, $\odot$, or $\oplus$) have smaller (larger) probabilities, then it immediately follows that the probability at that node becomes smaller (larger).

Given bounds at the children, the lower and upper bounds at the parent node are obtained by replacing in the formulas for computing the probability of nodes $\oplus$, $\otimes$, and $\odot$, the exact probability of the children with their lower and upper bounds, respectively. We are now ready to generalize the result of Proposition 5.2 from DNFs to d-trees.

Proposition 5.3. *If a d-tree d for a DNF Φ has bounds $[L, U]$, then it holds that $L \leq P(\Phi) \leq U$.* □

A key property of d-trees is the incremental refinement of their bounds through compilation. This observation is used in the incremental algorithm presented in Section 5.4.

Proposition 5.4. *Given a d-tree d with bounds $[L_d, U_d]$. Then, for any refinement e of d with bounds $[L_e, U_e]$, it holds that $[L_e, U_e] \subseteq [L_d, U_d]$.* □

In case the refinement e is a complete d-tree, then its bounds are a point interval.

5.3 Approximation Errors and Probability Bounds

We consider here two types of approximations, given a fixed error factor ϵ $(0 \leq \epsilon < 1)$.

Definition 5.5. *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 $(1 - \epsilon) \cdot p \leq \hat{p} \leq (1 + \epsilon) \cdot p$. □

Given a d-tree for a DNF Φ, its bounds $[L, U]$ may contain several ϵ-approximations of $P(\Phi)$, although not every value between these bounds is an ϵ-approximation. The connection between the bounds of a d-tree for Φ and ϵ-approximations of $P(\Phi)$ is given by the following theorem.

Proposition 5.6. *Given a DNF Φ, a fixed error ϵ, and a d-tree for Φ with bounds $[L, U]$.*

- *If $U - \epsilon \leq L + \epsilon$, then any value in $[U - \epsilon, L + \epsilon]$ is an absolute ϵ-approximation of $P(\Phi)$.*
- *If $(1 - \epsilon) \cdot U \leq (1 + \epsilon) \cdot L$, then any value in $[(1 - \epsilon) \cdot U, (1 + \epsilon) \cdot L]$ is a relative ϵ-approximation of $P(\Phi)$.* □

Proof. See Appendix A. □

In the sequel, we call a d-tree for a DNF Φ an (absolute or relative) ϵ-approximation of Φ if its bounds satisfy the above sufficient condition. Written differently, the condition becomes

$$U - L \leq 2 \cdot \epsilon \text{ in the absolute case, and}$$
$$(1 - \epsilon) \cdot U - (1 + \epsilon) \cdot L \leq 0 \text{ in the relative case.}$$

This condition can be checked in linear time in the size of the d-tree: We need one pass over the d-tree to compute its lower and upper bounds, and then check the above condition, which only involves the bounds and the fixed error.

Example 5.7. *Recall the DNF* Φ *from Example 5.1. Its exact probability is* $p = 0.8456$*. With bounds [0.842,0.848], we obtain precisely one absolute 0.03-approximation* $\hat{p} = 0.845$*, because* $0.848 - 0.03 = 0.842 + 0.03$*. We can also obtain an interval of absolute 0.04-approximations [0.844,0.846].* □

5.4 Approximation Algorithms

5.4.1 A Naive Main-Memory Algorithm

Proposition 5.6 can be effectively used for approximate confidence computation as follows. While compiling a DNF into a d-tree, we can ask before the construction of each node of the d-tree whether the sufficient condition on the approximation is reached. If this is the case, then we can stop the compilation and output the interval of ϵ-approximations. If this is not the case, then we continue with the compilation and choose the leaf with the largest bounds interval and further refine it. This already gives us a simple incremental algorithm for computing an ϵ-approximation.

5.4.2 An Incremental and Memory-Efficient Algorithm

The algorithm sketched in Section 5.4.1 needs to keep every node it creates in main memory. This is unfeasible. We therefore consider next the practical question of whether the sufficient condition for ϵ-approximation can still be fulfilled after subsequent refinement even if some leaves are not refined anymore. In the sequel, we call such leaves *closed*; an *open* leaf may be further refined to completion.

The technical challenge here is to obtain an ϵ-approximation condition in the presence of closed leaves. Based on this, we can develop an algorithm for computing ϵ-approximations for a DNF that incrementally compiles it into a d-tree in depth-first left-to-right traversal, and decides *locally* whether the current leaf under exploration can be closed or must be refined further. When a leaf is closed, its bounds are used to update a pair of aggregated bounds of all the leaves already closed, and the leaf is released.

This gives us a very efficient algorithm that need only keep in memory the current root-to-leaf path under construction and some local information at each node along this path.

In the sequel, we consider d-trees, where at most one child of each $\odot$ node may be closed without being complete. This does not restrict our encoding of variable elimination as given in Figure 16, since the $\odot$ nodes needed there are binary and one of their children is always a clause, i.e., it is complete, and for which the exact probability is known.

To understand the worst-case scenario in case we want to close a leaf in a d-tree d, we need to compute the largest bounds interval of d for any possible probability each open leaf may take. If these bounds fail to satisfy the condition for an ϵ-approximation, then we may not reach such an approximation by any refinement that completes the open leaves. In this case, we must not close that leaf.

Definition 5.8. *The* bound space *of a d-tree d is the set of possible bounds $[L, U]$ of d obtained by choosing for each open leaf any point interval between the bounds of that leaf.* □

Let us denote by $L(d)$ the element of the bound space obtained by choosing for each open leaf the point interval $[L_i, L_i]$, where L_i is a lower bound for that leaf.

Lemma 5.9. *For a d-tree d, $L(d)$ is the pair of bounds $[L, U]$ that maximizes each of $U - L$ and $(1-\epsilon)\cdot U - (1+\epsilon)\cdot L$ over the entire bound space of d.*□

Proof. See Appendix A. □

Lemma 5.9 gives us the necessary strategy to decide whether closing leaves in a d-tree still allows to obtain an ϵ-approximation. Finding the maximal values of $U-L$ and $(1-\epsilon)\cdot U-(1+\epsilon)\cdot L$ can be done very efficiently by computing $L(d)$ in just one scan of d. Our main result concerning the closing of leaves follows then from Lemma 5.9 and Proposition 5.4.

Theorem 5.10. *Given a d-tree d for a DNF Φ, and a fixed error ϵ. If the bounds $L(d)$ satisfy the sufficient condition for an ϵ-approximation in Proposition 5.6, then there is a refinement of d that is an ϵ-approximation of Φ.* □

Example 5.11. *Consider the d-tree d of Fig. 20 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

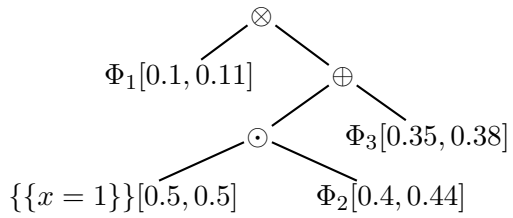


Figure 20: D-tree. Leaves: Φ_1 is closed, Φ_2 is current, Φ_3 is open.

$L = 0.1 \otimes ((0.5 \odot 0.4) \oplus 0.35) = 0.595$. *Similarly for the upper bound:* $U = 0.11 \otimes ((0.5 \odot 0.44) \oplus 0.38) = 0.644$. *The condition* $U - L = 0.049 \leq 2 \cdot 0.012 = 0.024$ *is not satisfied. Hence, we cannot stop now.*

(2) We compute $L(d)$ *as before:* $L(d) = [L, U']$, *where* $U' = 0.11 \otimes ((0.5 \odot 0.44) \oplus 0.35) = 0.6173$. *We then have that* $U' - L = 0.0223 \leq 0.024$. *We may thus close this point.* □

Our incremental algorithm is the compilation scheme of Figure 16, where the variable choice is according to the variable elimination order of Figure 18. The nodes in the d-tree are constructed in depth-first manner. Before constructing a node, we perform two checks: (1) the sufficient condition of Proposition 5.6, which tells us whether we already reached an ϵ-approximation and we can safely stop, and (2) the condition of Theorem 5.10 on whether the current node to be constructed can be safely closed, in case the condition at step (1) is not satisfied. In step (2), we compute the bounds of the DNF at the leaf using the Independent heuristic of Figure 19.

6 MayBMS with the SPROUT Query Engine

The *MayBMS* system (note: MayBMS is read as "maybe-MS", like DBMS) is a complete probabilistic database management system that leverages robust relational database technology. The MayBMS system has been under development since 2005 and has undergone several transformations. MayBMS has been released and is available for download at

http://maybms.sourceforge.net.

A fundamental design choice that sets MayBMS apart from existing research prototypes such as Trio and MystiQ is that MayBMS is an extension of the open-source PostgreSQL *server backend*, and not a front-end application of PostgreSQL. Our backend is easily accessible through multiple APIs (inherited from PostgreSQL), and has efficient internal operators for processing probabilistic data.

Several demonstration scenarios (data cleaning using constraints, human resources management, and analysis of social networks) are available at the MayBMS website. To demonstrate how easily applications can be built on top of MayBMS, we used PHP to construct a website that offers NBA[3]-related information based on what-if analysis of team dynamics (player fitness, skill management) using data available at www.nba.com. We show for instance how such an application can predict the players' fitness by simulating random walks on stochastic matrices encoding the transitions in player's fitness based on severity of recent injuries.

MayBMS was demonstrated at ACM SIGMOD 2009 under the title "MayBMS: A Probabilistic Database Management System" [19].

6.1 System Overview

MayBMS stores probabilistic data in U-relational databases, a succinct and complete representation system for large sets of possible worlds [4]. Queries are expressed in an extension of SQL with specialized constructs for probability computation and what-if analysis [26]. In addition, MayBMS uses several state-of-the-art exact and approximate confidence computation techniques [9, 25, 33].

[3]NBA stands for National Basketball Association.

6.1.1 U-Relational Databases

A U-relational database consists of a set of U-relations. U-relations are standard relations extended with condition and probability columns to encode correlations between the uncertain values and probability distribution for the set of possible worlds [4]. The condition columns store variables from a finite set of independent random variables and their assignments; the probability columns store the probabilities of the variable assignments occurring in the same tuple. A U-relation can have several such condition (and probability) columns. Attribute-level uncertainty is achieved through vertical decompositions, and an additional (system) column is used for storing tuple ids and undoing the vertical decomposition on demand. A formal definition of U-relational databases is in Section 2.

6.1.2 The MayBMS Query Language

The MayBMS query language extends SQL with uncertainty-aware constructs [26]. Its features include genericity, compositionality, and a well-understood relationship to existing query languages. An important subset of its constructs is presented next.

In the sequel, U-relations without condition and probability columns, which correspond to standard relations, are called *typed-certain (t-certain) tables*. The MayBMS query language has constructs that map (i) uncertain tables to t-certain tables, such as confidence computation constructs, (ii) uncertain to uncertain and t-certain to t-certain tables, such as full SQL, and (iii) t-certain to uncertain tables, such as constructs that extend the hypothesis space and create new possible worlds. Some restrictions are in place to assure that query evaluation is feasible. In particular, we do not support the standard SQL aggregates such as `sum` or `count` on uncertain relations (but we do support expectations of aggregates). This can be easily justified: in general, these aggregates will produce exponentially many different numerical results in the various possible worlds, and there is no way of representing exactly these results in an efficient manner. In addition to standard SQL, MayBMS supports the following uncertainty-aware constructs:

1. conf, aconf, tconf, and possible: These constructs map uncertain tables to t-certain tables consisting of tuples possible in some of the worlds represented by the input, with or without their exact or approximate confidences.

The construct `conf` returns the exact confidence of each distinct tuple

while both `conf`$(approach, \epsilon)$ and `aconf`(ϵ, δ) computes an approximation. `conf`$(approach, \epsilon)$ outputs an absolute or relative ϵ-approximation (specified by the parameter *approach*). `aconf`(ϵ, δ) computes an (ϵ, δ)-approximation of this confidence, i.e., the probability that the computed value $\hat{p}$ deviates from the correct probability p by more than $\epsilon \cdot p$ is less than δ. Syntactically, the confidence computation constructs `conf` and `aconf` are treated like SQL aggregates. By using aggregation syntax and not supporting `select distinct` on uncertain relations, we avoid the need for conditions beyond the special conjunctions that can be stored with each tuple in U-relations.

The construct `tconf` computes the marginal probability of each tuple in isolation from the other (possibly duplicate) tuples. The construct `possible` can be added to `select` statements and its effect is to filter out the tuples with probability zero and eliminate the duplicates. It can thus be reformulated using `tconf`.

2. repair-key and pick-tuples: These constructs map t-certain tables to uncertain tables. Conceptually, `repair- key` takes a set of attributes $\vec{K}$ and a relation R as arguments and nondeterministically chooses a maximal repair of key $\vec{K}$ in R, that is, it removes a minimal set of tuples from R such that the key constraint is no longer violated. The `repair-key` operation accepts an optional argument that allows us to assign nonuniform probabilities to the possible choices.

The construct `pick-tuples` creates a probabilistic relation representing all the possible subsets of the input table.

Note that repair-key and pick-tuples are queries, rather than update statements. They have the following syntax:

- repair key <attributes> in <t-certain-query>
  [weight by <expression>]

- pick tuples from <t-certain-query>
  [independently] [with probability <expression>]

The parameter [independently] ensures that the output probabilistic relation is tuple-independent.

3. argmax: The aggregate argmax(arg,value) outputs all the `arg` values in a group (specified by the group-by clause) whose tuples have a maximum `value` within that group.

4. esum and ecount: Although the standard SQL aggregates are forbidden on uncertain relations, MayBMS supports aggregate operations on uncertain relations such as `esum` and `ecount`, which compute expected sums and counts across groups of tuples. While it may seem that these

aggregates are at least as hard as confidence computation (which is #P-hard), this is in fact not so. These aggregates can be efficiently computed using linearity of expectation.

Uncertain queries can be constructed from t-certain queries (queries that produce t-certain tables), `select-from -where` queries over uncertain tables, the multiset union of uncertain queries (using SQL `union`), and `repair-key` and `pick-tuples` statements. The `select-from-where` queries may use any t-certain subqueries in the conditions, plus uncertain subqueries in IN-conditions that occur positively.

6.1.3 Query Processing

MayBMS evaluates queries on top of U-relations.

Positive relational algebra: The answers to positive relational algebra queries (without confidences) can be computed using a parsimonious translation of such queries into (again) positive relational algebra queries that are then evaluated in standard relational way on U-relations [4].

Approximate confidence computation with Monte Carlo Estimation: The approximation algorithm used by MayBMS is a combination of the Karp-Luby unbiased estimator for DNF counting in a modified version adapted to confidence computation in probabilistic databases, and the Dagum-Karp-Luby-Ross optimal algorithm for Monte Carlo estimation [9]. The latter is based on sequential analysis and 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.

Approximate confidence computation with d-trees: This is described in Sections 4 and 5.

General-purpose exact confidence computation: Our general-purpose exact algorithm for confidence computation is described in [25]. Given a DNF (of which each clause is a conjunctive local condition), the algorithm employs a combination of variable elimination and decomposition of the DNF into independent subsets of clauses (i.e., subsets that do not share variables), with cost-estimation heuristics for choosing whether to use the former (and for which variable) or the latter. Outside a narrow range of variable-to-clause count ratios, it outperforms the approximation techniques [25].

Exact confidence computation on tractable queries: For tractable conjunctive queries with only equalities on probabilistic databases, MayBMS

uses the techniques in [33] for scalable query processing by reduction of confidence computation to a sequence of SQL-like aggregations. For tractable conjunctive queries with also inequalities, OBDD-based technique in Section 3 is applied.

Updates, concurrency control, and recovery: As a consequence of our choice of a purely relational representation system, these issues cause surprisingly little difficulty. U-relations are represented relationally and updates are just modifications of these tables that can be expressed using the standard SQL update operations.

6.1.4 Implementation

MayBMS is built entirely inside PostgreSQL. The major changes lie in the system catalog, parser, and executor. U-relations are implemented by storing the variables and their possible assignments as pairs of integers, and probabilities as floating-point numbers. The system catalog can distinguish between U-relations and standard relational tables. Confidence computation and other aggregates such as `esum` and `ecount` are registered in the system catalog and implemented as operators in the PostgreSQL executor. The constructs `repair-key`, `pick-tuples`, and `possible` are implemented by rewriting to SQL.

6.2 An Application Scenario: Human Resources Management

We have developed several applications on top of MayBMS. They are available at the MayBMS website. We next discuss an implemented application for risk management in the human resources space, in the context of basketball.

Using PHP, we have built on top of MayBMS a web-based application that offers NBA-related decision support functionality based on what-if analysis of team dynamics such as player fitness and skill management. The application uses data available at www.nba.com.

Team management. In the pre-season period, the manager intends to attract new players to strengthen the team. An important question is whether the skills status of the team can be improved. For this, we compute for each skill (such as defense, three-point, and free shooting) the probability that someone with that skill will be playing in the team given the current status of the players (injured, top-form). In a scenario of financial crisis, the team

budget is reduced and the manager intends to lay off some players with high salaries but at the same time without compromising the competitiveness of the team significantly. For instance, we may want to keep the availability of skill shooting at least 90% and of passing at least 95%. The manager needs to know whether this is possible and who can be laid off. Figure 21 shows the screen shots of this scenario.

Performance prediction. Coaches would like to predict the performance of players, for example how many points a player will score in the next game. Based on the player's recent performance, one can build a simple model to calculate this: if we associate higher weights to the more recent performance of the players, their predicted performance can be expressed in terms of the weighted points. Figure 22 shows the screen shots of this scenario.

Fitness prediction. Suppose there is a must-win match three days later and some of the key players of the team have been recently plagued by injuries. The report from the team doctor shows that the players are likely to get injured and the time for comeback varies depending on the recovery progress. We can model the fitness of each player using a stochastic matrix that states the probabilities for one-day transitions between states such as fit (F), seriously injured (SE), and slightly injured (SL). Asking for the three-day fitness of a player can be performed as a random walk of length three on this matrix. Random walks can be encoded as queries using `repair-key` and confidence computation on top of relational encodings of stochastic matrices. Figure 23 gives a stochastic matrix, its relational encoding FT, and the U-relation R2 representing a 1-step random walk on FT. The 3-step random walk on FT is achieved by the following query statements, where the initial state of each player is considered given in a (certain) table States (Player, State).

```
create table FT2 as
select R1.Player, R1.Init, R2.Final, conf() as p from
(repair key Player, Init in FT weight by p) R1,
(repair key Player, Init in FT weight by p) R2, States S
where R1.Player = S.Player and R1.Init = S.State
and R1.Final = R2.Init and R1.Player = R2.Player
group by R1.Player, R1.Init, R2.Final;

select R1.Player, R2.Final as State, conf() as p from
(repair key Player, Init in FT2 weight by p) R1,
(repair key Player, Init in FT weight by p) R2
where R1.Final = R2.Init and R1.Player = R2.Player
group by R1.player, R2.Final;
```

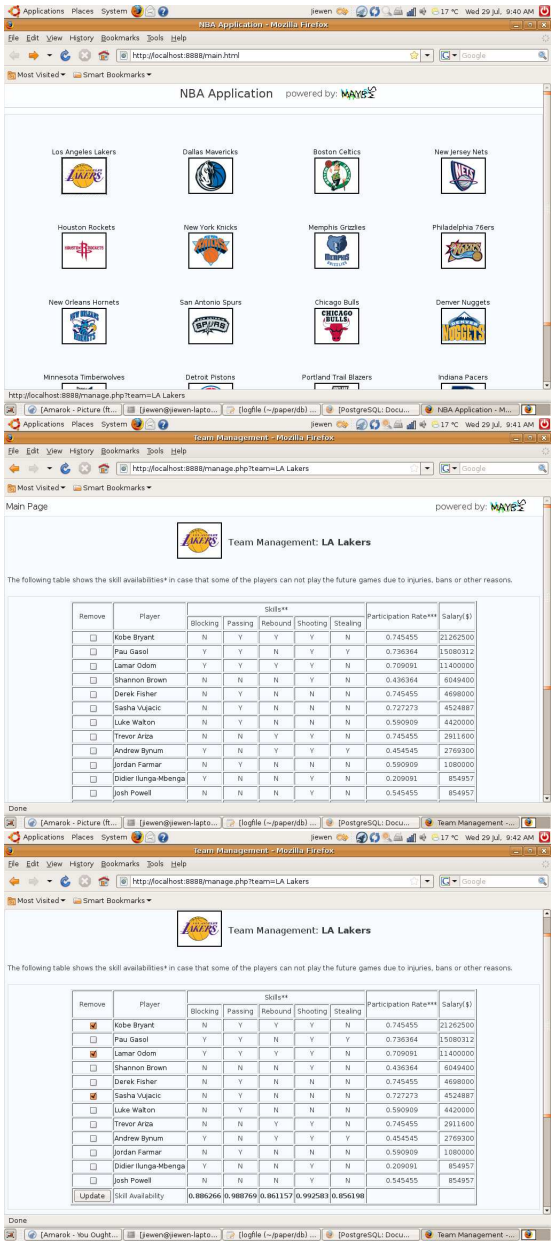


Figure 21: Screen shots of team management scenario.

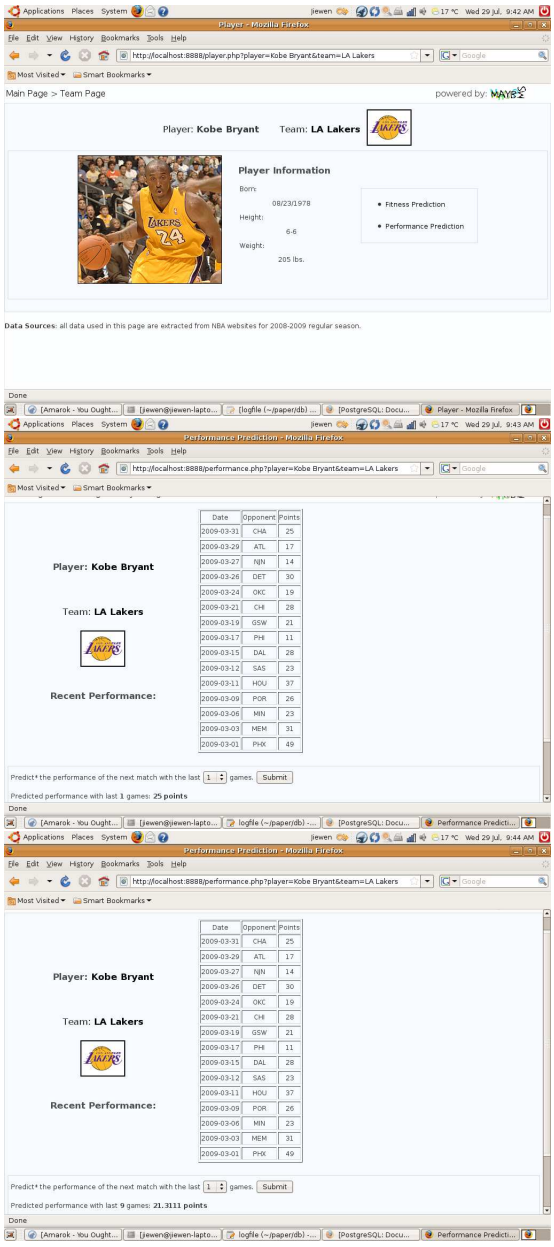


Figure 22: Screen shots of performance prediction scenario.

Fitness stochastic matrix For player Bryant			
	F	SE	SL
F	0.8	0.05	0.15
SE	0.1	0.6	0.3
SL	0.8	0.0	0.2

FT (FitnessTransition)			
Player	Init	Final	P
Bryant	F	F	0.8
Bryant	F	SE	0.05
Bryant	F	SL	0.15
Bryant	SE	F	0.1
Bryant	SE	SE	0.6
Bryant	SE	SL	0.3
Bryant	SL	F	0.8
Bryant	SL	SL	0.2
Other players			

U-relation R2 (1-step random walk on FT)				
Player	Init	Final	condition	P
Bryant	F	F	$x \mapsto 1$	0.8
Bryant	F	SE	$x \mapsto 2$	0.05
Bryant	F	SL	$x \mapsto 3$	0.15
Bryant	SE	F	$y \mapsto 1$	0.1
Bryant	SE	SE	$y \mapsto 2$	0.6
Bryant	SE	SL	$y \mapsto 3$	0.3
Bryant	SL	F	$z \mapsto 1$	0.8
Bryant	SL	SL	$z \mapsto 2$	0.2
Other players				

Figure 23: Random walk on a stochastic matrix.

A 1-step random walk on FT is performed by nondeterministically choosing from each Init state of each player a possible Final state using the repair-key construct. We encode it as a U-relation (see R2) that only adds to FT a condition column over independent random variables x, y, and z, which are used to express the correlations created by repair-key: for each Init value, the possible Final values are mutually exclusive, and the choices of Init values are pairwise independent. A 2-step random walk is expressed as a join of two 1-step walks, whereby the Final state of the first walk becomes the Init state of the second. The probability that each player has a certain state is computed using the conf() construct. The table FT2 encodes the stochastic matrix representing the product of the initial stochastic matrix with itself. For a 3-step random walk, we join the outcome of the previous 2-step walk with a 1-step walk. Figure 24 shows the screen shots of this scenario.

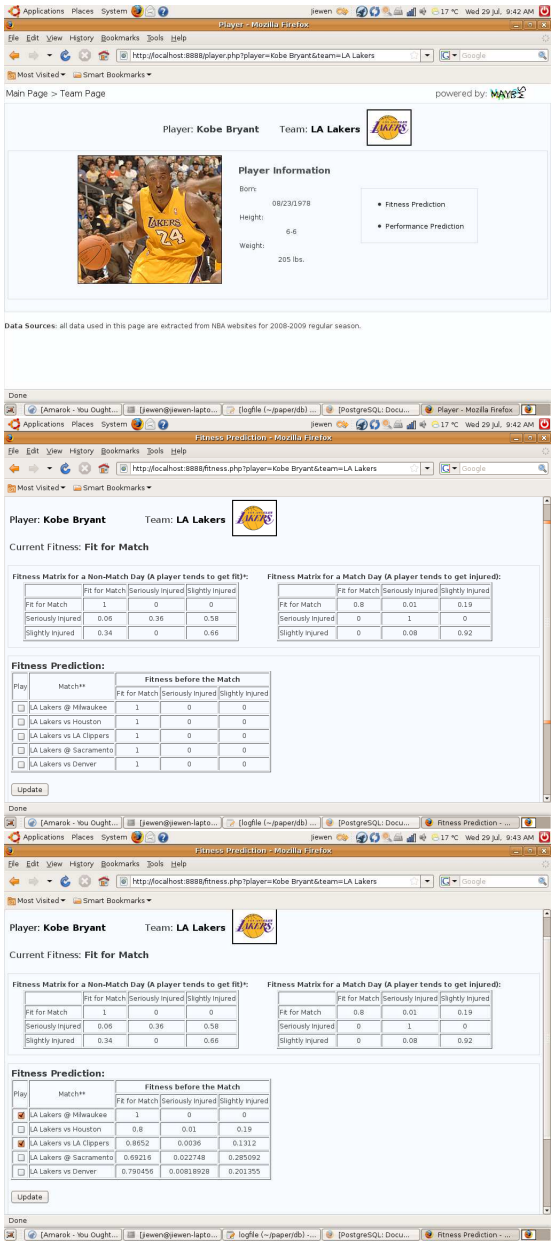


Figure 24: Screen shots of fitness prediction scenario.

7 Experiments

In this section, we report our experimental designs and findings for our previously discussed OBDD-based technique (Section 3) and d-tree approximation algorithm (Sections 4 and 5). Overall, experiments shows that our approaches outperform state-of-the-art exact and approximation algorithms for confidence computation by orders of magnitude. Sections 7.2 and 7.3 present the experimental results for the OBDD-based technique and d-tree algorithm, respectively.

7.1 Experimental Setup

All experiments were conducted on an AMD Athlon Dual Core Processor 5200B 64bit/3.9GB/Linux2.6.25/gcc 4.3.0. We report wall-clock execution times of queries run in the psql shell. All resources needed to reproduce our experiments (algorithms, queries, data sets, data set generators) are available at http://web.comlab.ox.ac.uk/projects/SPROUT/.

7.2 Experiments for Tractable Queries with Inequality Joins

Our experiments are focused on three key issues: scalability, comparison with existing state-of-the-art algorithms, and comparison with "plain" querying where we replaced confidence computation by a simple aggregation (counting). The findings suggest that our confidence computation technique scales very well: We report on wall-clock times around 200 seconds to compute the probability of query lineage of up to 20 million clauses. When compared with existing confidence computation algorithms, our technique outperforms them by up to two orders of magnitude in cases when the competitors need less than the allocated time budget of 20 minutes. We also found that lineage sorting has the lion's share of the time needed to compute the distinct answer tuples and their confidences.

7.2.1 TPC-H Data

We generated tuple-independent databases from deterministic databases produced using TPC-H 2.8.0. We added to the table Customer a `c_registrationdate` column and set all of its fields to `1993-12-01`. This value was chosen so that the inequality predicates in our queries are moderately selective: `1993-12-01`$<$`o_orderdate` holds for instance for about one fourth of the 1.5 million tuples in Orders (scale factor 1). We associated each tuple

with a distinct Boolean random variable and chose at random a probability distribution over these variables.

7.2.2 Queries

We evaluated the six queries shown in Figure 25. The aggregate construct `conf()` specifies confidence computation of distinct tuples in the query answer.

The first query returns the likelihood that orders are shipped within five days from the date of order. In case the database has violations regarding order dates, this query also returns the wrong database entries for orders that are placed after they are shipped. The query is a join on tables Orders and Lineitem and involves one equality and one inequality join conditions. The second query asks for likelihood of items shipped with delay to old customers. It involves an inequality path of length two. Query 3 returns the likelihood that the customers that ordered shortly after their registration registration and have not yet received the ordered items. The condition graph of this query is a tree. Query 4 returns the parts without profit. Query 5 gives the likelihood that large quantities are shipped within a few days after some particular order placements. Query 6 computes for each nation the likelihood that a customer has a higher account balance than a supplier.

7.2.3 Competitors

To the best of our knowledge, our technique (denoted by "ours" in the graphics) is the first technique for exact confidence computation of conjunctive queries with inequalities on tuple-independent probabilistic databases that has polynomial-time guarantees. We cannot therefore experimentally compare our technique with an existing one specifically designed to tractable queries with inequalities.

We compare it instead with two state-of-the-art confidence computation algorithms that are applicable to arbitrary lineage. They represent the query evaluation techniques of MayBMS [19], which is a publicly available extension of the PostgreSQL backend (`http://maybms.sourceforge.net`).

The first algorithm (denoted by "conf") is an exact confidence computation algorithm with good behaviour on randomly generated data [25]. It compiles the lineage into a weak form of d-NNF (decomposable negation normal form) on which probability computation can be done linearly [13].

1	select conf() from orders, lineitem where o_orderkey = l_orderkey and o_orderdate > l_shipdate - 3;
2	select conf() from customer, orders, lineitem where c_custkey = o_custkey and o_orderkey = l_orderkey and c_registrationdate + 30 < o_orderdate and o_orderdate + 100 < l_shipdate;
3	select conf() from customer, orders, lineitem where c_custkey = o_custkey and o_orderkey = l_orderkey and c_registrationdate + 30 < o_orderdate and c_registrationdate + 100 < l_receiptdate;
4	select conf() from part, lineitem where p_partkey = l_partkey and l_extendedprice / l_quantity ≤ p_retailprice;
5	select conf() from orders, lineitem where o_orderdate < l_shipdate and l_quantity > 49 and o_totalprice > 450000;
6	select s_nationkey, conf() from supplier, customer where s_acctbal < c_acctbal and s_nationkey = c_nationkey and s_acctbal > 9000 group by s_nationkey;

Figure 25: Queries used in the experiments.

The second algorithm (denoted by "aconf") is a Monte Carlo simulation for confidence computation [35, 10] based on the Karp-Luby (KL) fully polynomial randomized approximation scheme for DNF counting [22]. In short, given a DNF formula with m clauses, the base algorithm computes an (ϵ, δ)-approximation $\hat{c}$ of the number of solutions c of the DNF formula such that $\Pr[|c-\hat{c}| \leq \epsilon \cdot c] \geq 1-\delta$ for any given $0 < \epsilon < 1$, $0 < \delta < 1$. It does so within $\lceil 4 \cdot m \cdot \log(2/\delta)/\epsilon^2 \rceil$ iterations of an efficiently computable estimator. Following [25], we used in the experiments the optimal Monte-Carlo estimation algorithm of [9].

In contrast to our technique, both competitors can process lineage of arbitrary queries on complete probabilistic database models (such as U-relations [25]), with no special consideration for tractable queries. On the down side, they require main-memory representation of the entire lineage and also random access to its clauses and variables. In addition, as our experiments show, they are very time inefficient for the considered workload.

The experiments use TPC-H scale factors 0.005, 0.01, 0.1, 0.5, and 1 (a scale factor of x means a database of size xGBs).

Query	Lineage size	# duplicates per distinct tuple
1	99,368	99,368 (Boolean query)
2	725,625	725,625 (Boolean query)
3	4,153,850	4,153,850 (Boolean query)
4	6,001,215	6,001,215 (Boolean query)
5	20,856,686	20,856,686 (Boolean query)
6	256,187	min #duplicates: 5028 max #duplicates: 14252 avg #duplicates: 10247

Figure 26: Lineage Characteristics (scale factor 1).

7.2.4 Sizes of Query Lineage

We experimented with queries that produce large lineage. Figure 26 reports the sizes of the lineage (ie, number of clauses) for each of our queries, before confidence computation and duplicate elimination are performed, and the number of duplicates per distinct answer tuple. The performance of confidence computation algorithms depends dramatically on the number of duplicates. In case of Boolean queries, all answer tuples are duplicates; for scale factor 1, this means that our algorithm computes the probability of a DNF formula of about 20 million clauses in one of our experiments – according to Figure 27 it does so in about 200 seconds. Further inspection shows that the actual confidence computation needs under one second, the rest of the time being needed for sorting the lineage necessary for duplicate elimination and confidence computation.

7.2.5 Comparison with State-of-the-Art Algorithms

Figure 27 shows the results of our experimental comparison. For aconf, the allowed error is 10% with probability 99%. We also consider the time taken by (unmodified) PostgreSQL to evaluate the queries, where confidence computation has been replaced by counting ("plain"). Overall, our algorithm outperforms the competitors by up to two orders of magnitude even for very small scale factors such as 0.01.

The algorithm aconf runs out of the allocated time (20 minutes) in most of the scenarios, and the diagrams for queries 3 and 4 do not have data points for aconf. We also verified experimentally that the optimal Monte-Carlo estimation algorithm [9] used for aconf in MayBMS is sensitive to the probability distribution, while this is not true for our technique nor for conf. For positive DNF formulas representing lineage of queries on tuple-independent databases, aconf needs less time if the probability values for

the *true* assignments of the variables are close to 0. The algorithm conf performs better than aconf, although conf also exceeds the allocated time in about half of the tests.

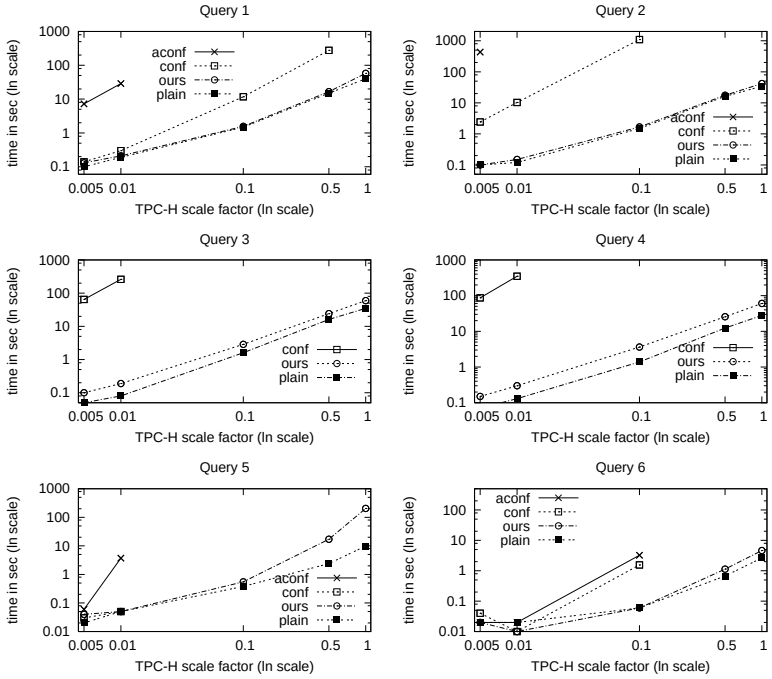


Figure 27: Effect of varying the scale factor on query evaluation using our technique, plain, conf, and aconf.

7.2.6 Cost of Lineage Sorting

We verified experimentally that in all of our scenarios, the actual confidence computation time takes up to few seconds only. For instance, for scale factor 1, it stays within 5% of the total execution time, while the remaining time is taken by sorting the answer tuples and their lineage. This sorting step is necessary for duplicate elimination and confidence computation. Whereas for non-Boolean queries both counting and confidence computation need sorting for duplicate elimination, this is not the case for Boolean queries. In the latter case, sorting is still required for confidence computation, but not for counting. For query 5, the difference in execution time between counting and confidence computation is indeed due to sorting. Besides sorting, both counting and confidence computation require one scan over the answer tuples

and need comparable time.

7.3 Experiments for D-Trees

7.3.1 Competitors

We experimentally compare the following algorithms:
1. d-tree: This is the new algorithm developed in this thesis. It takes as argument an error budget ϵ, which can be interpreted as either an absolute or a relative error bound. Since this is a deterministic algorithm, repeating an experiment for a given query and dataset yields exactly the same result, at close to the same running times.
2. aconf: The algorithm aconf() computes an (ϵ, δ)-approximation of tuple confidence and takes ϵ and δ as arguments. It is a combination of the Karp-Luby unbiased estimator for DNF counting [22, 23] in a modified version adapted for confidence computation in probabilistic databases (cf. e.g. [24]) and the Dagum-Karp-Luby-Ross optimal algorithm for Monte Carlo estimation [9]. The latter is based on sequential analysis and 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. We actually use the probabilistic variant of a version of the Karp-Luby estimator described in the book [38] which computes fractional estimates that have smaller variance than the zero-one estimates of the classical Karp-Luby estimator.
3. OBDD-based techniques: These efficient secondary-storage algorithms are the state-of-the-art exact confidence computation techniques for all currently known classes of conjunctive queries with inequalities and without self-joins on tuple-independent probabilistic databases [33, 31].

7.3.2 Experiment Design

Our experiments were designed to provide insight into the performance of our confidence approximation algorithm across a variety of datasets and queries that are representative of future applications of probabilistic databases. Since no benchmark has been established so far for query processing in probabilistic databases, and there is not even wide agreement yet on a set of most relevant use cases, we have to rely on our understanding of the possible sources of hardness in confidence computation that may arise in a variety of applications.

In addition to the obvious sources of hardness (such as large data and queries with many joins, which create complex conditions), there are a few

more subtle issues to be considered:

1. Tuple-independent databases versus databases with more complicated lineage. The queries in our experiments create complex "lineage" formulas. (This is the main reason why we have to study algorithms for confidence computation.) However, we focus on queries whose relational algebra part is positive since the relational difference operation is a substantial source of complexity (cf. e.g. [34]). Thus, if we start with tuple-independent relations in which each tuple is associated with its own Boolean random variable, conjunctive queries will only create conditions that are conjunctions of non-negated occurrences of these variables. This has an effect on confidence computation algorithms; in fact, for our algorithm, mixed positive and negated variables in the conditions may possibly make confidence computation easier, because it may allow the upper- and lower-bounding mechanisms to converge more quickly.

2. Easy-hard-easy pattern. In [25], we observed such a pattern similar to those observed in combinatorial algorithms for propositional satisfiability and constraint satisfaction: When the ratio of variables to clauses is very large, then the result probability is rather small and the input to the algorithm is small: such a case tends to be easy. Similarly, if the ratio of variables to clauses is very small, then the result probability tends to be very close to 1 and lower-bounding with sufficient accuracy is easy. However, there is a critical region of variable-to-clause ratios inbetween for which probability computation is hard. For our experiments, this means that there is a pitfall in increasing the instance sizes: 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 and on many instances we will do well without difficulty.

3. Absolute versus relative approximation. When result probabilities are reasonably close to 1, then there is no great difference between 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 the d-tree algorithm is important, since relative approximation is a staple of the Karp-Luby approximation scheme (aconf). Designing a Monte Carlo algorithm for efficient absolute approximation is trivial.

Our experiments are designed to study all of these issues. In our experiments with TPC-H data, we start with tuple-independent databases, but

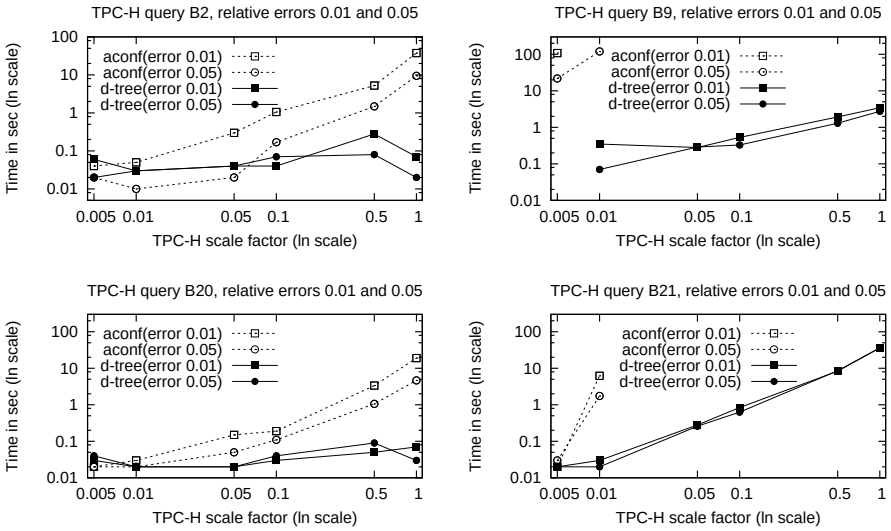


Figure 28: Experimental results for hard TPC-H queries.

construct scenarios where the result probabilities are small and relative approximation is interesting. These experiments also exhibit interesting join patterns arising from business decision support queries.

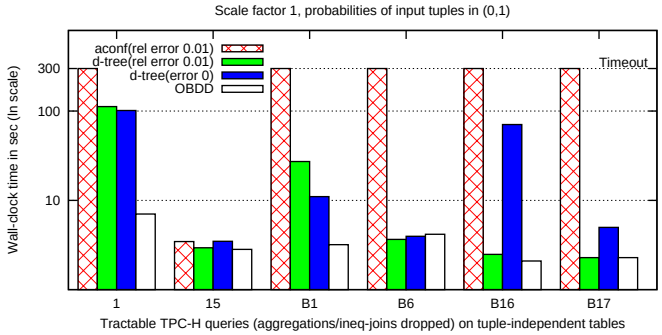
Our experiments with graph data study a wide variety of distributions and data densities, and start with block-independent-disjoint tables which lead to negated boolean random variables; this allows us to study the impact of rich forms of lineage on our algorithm.

We considered both relative and absolute approximations (particularly where the result probabilities are small and relative approximation is needed for high-quality results).

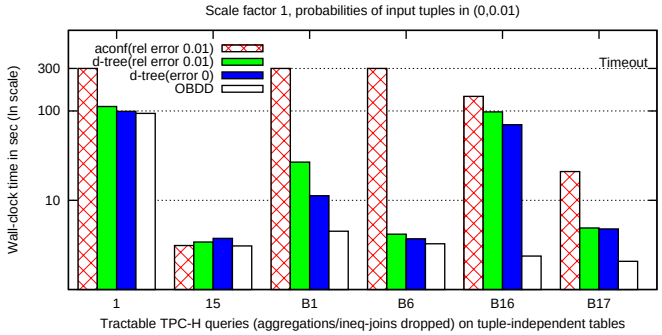
7.3.3 TPC-H Experiments

The first broad class of experiments was performed on data generated by a modified version of the TPC-H data generator which creates tuple-independent probabilistic databases [10], that is, each tuple occurs in the database independently with a given probability. We consider slightly modified versions of the TPC-H queries without aggregations but with confidence computation.

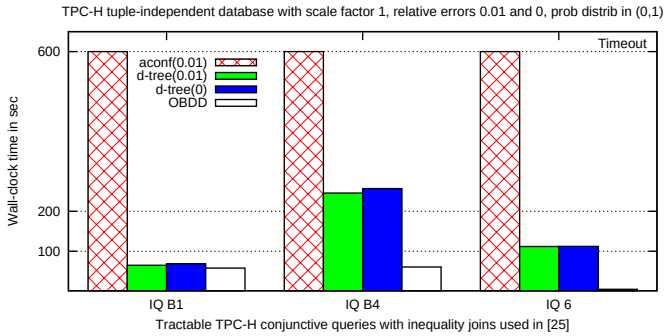
The queries of the TPC-H benchmark fall into two main classes: tractable queries with inequalities (six hierarchical queries used in [33] and three inequality queries used in [31]), and four #P-hard queries. Queries marked with "B" are Boolean. Two of the tractable queries are selections on the



(a)



(b)



(c)

Figure 29: Experimental results for tractable queries.

large lineitem table, all other tractable queries are joins of two large tables (e.g., lineitem with supplier, or orders, or part). The hard queries are more complex: 20B is a join on supplier, nation, partsupplier, and part, 21B is a join on supplier, lineitem, orders, and nation, 2B is a join on part, supplier, partsupplier, nation, and region, and 9B is a join on part, supplier, lineitem, partsupplier, orders, and nation.

Fig. 29 shows the running times for computing the answers to tractable queries and their confidences. Overall, d-tree performs worse than OBDD-based technique because it learns the structure of the DNF from the query, whereas d-tree has to rediscover it on its own. The timing of the two is however comparable in almost all cases.

For hierarchical queries (Fig. 29(a) and (b)) we considered input data with probability distributions in (0,1) and also in (0,0.01). Our algorithm d-tree finishes in all cases within 100 seconds, even for computing the exact confidence. In contrast, aconf only finishes in four out of the 12 experiments. Overall, we obtain a better timing for error 0 than for relative error 0.01, because in the former case we do not need to compute the lower and upper bounds of each leaf during compilation. This becomes more evident in the case of small probabilities. In case of queries B16 and B17 in Fig. 29(a), checking the bounds clearly pays off: For these cases, no compilation is needed, since the lower and upper bounds are already approximated very well and we stop early. Without checking the bounds, we would have to construct the entire d-tree, which is then more expensive.

For tractable queries with inequalities (Fig. 29(c)), aconf does not finish in the allocated time, and d-tree follows closely OBDD-baed technique.

For all tractable queries, about 90% of the nodes in the d-tree are $\otimes$ nodes, which suggests that our approximation of lower and upper bounds for non-independent sets of clauses works very well and avoids possible exponentiality introduced by variable elimination. In addition, in case of inequality joins, the clause subsumption procedure is very effective. As explained in Section 4.2, this is vital for the overall polynomial time computation. For instance, the *IQ* query 6 has about 25 distinct answer tuples, each with a DNF of (in average) 10,000 clauses and 550 variables. For each answer tuple, d-tree creates (in average) 20,000 nodes, and subsumes ca. one million clauses (overall, on all branches of the d-tree).

Our algorithm d-tree performs consistently better than aconf also for hard queries. The hard queries have many joins, which ultimately lead to overall low probabilities of clauses, and with final confidences that range from 10^{-3} to 0.93, while answers have up to 500 clauses and 500 variables (query 20), up to 75,000 clauses and 150,000 variables (query 21), up to 640

clauses and 1,600 variables (query 2), and up to 350,000 clauses and 725,000 variables (query 9).

Statistics collected from d-tree traces show that in most of the cases, as the size of DNF increases, the number of nodes constructed by our algorithm also goes up. However, two scenarios may change this trend. First, in the lower and upper bound computation, with more input clauses, both the lower and upper bounds increase but maximal values of upper bounds are 1. If upper bounds reach 1 and lower bounds still increase, this can lead to quick convergence. For instance, for TPC-H query B2 and relative error 0.01, the number of nodes constructed by our algorithm reaches its peak at scale factor 0.5 and drops dramatically at scale factor 1. For larger errors, the U-turn happens even earlier. Still, for TPC-H query B2 but relative error 0.05, the maximal number of nodes appear at scale factor 0.1. Second, the DNF of some TPC-H queries (that have equality selections with constants) has the property that very few variables from one input table occur in most of the clauses. For instance, for queries B20 and B21, there is only one variable coming from table nation. After we eliminate this variable (chosen by MAX_OCCUR), the remaining DNF consists of many independent clauses and our approximation approach captures this and tighten the lower and upper bounds very quickly. Therefore, the number of nodes constructed remains low and are not affected by the DNF size.

7.3.4 Random Graph and Social Networks Experiments

The second broad class of experiments deals with graph data in which edges are independently either in the graph or absent. We consider two classes of datasets, which we model as block-independent disjoint tables. The first are generated random graphs in which 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 case the membership of each edge in the graph is uniform, the probability distribution over this set of possible worlds is uniform, too, and each world has probability $2^{n\cdot(n-1)}$.

The second class of graph datasets are well-known social networks taken from the literature (one is Zachary's Karate club [41], with 34 nodes, a classic, and the other represents friendship among a group of dolphins). The social networks generalize our 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

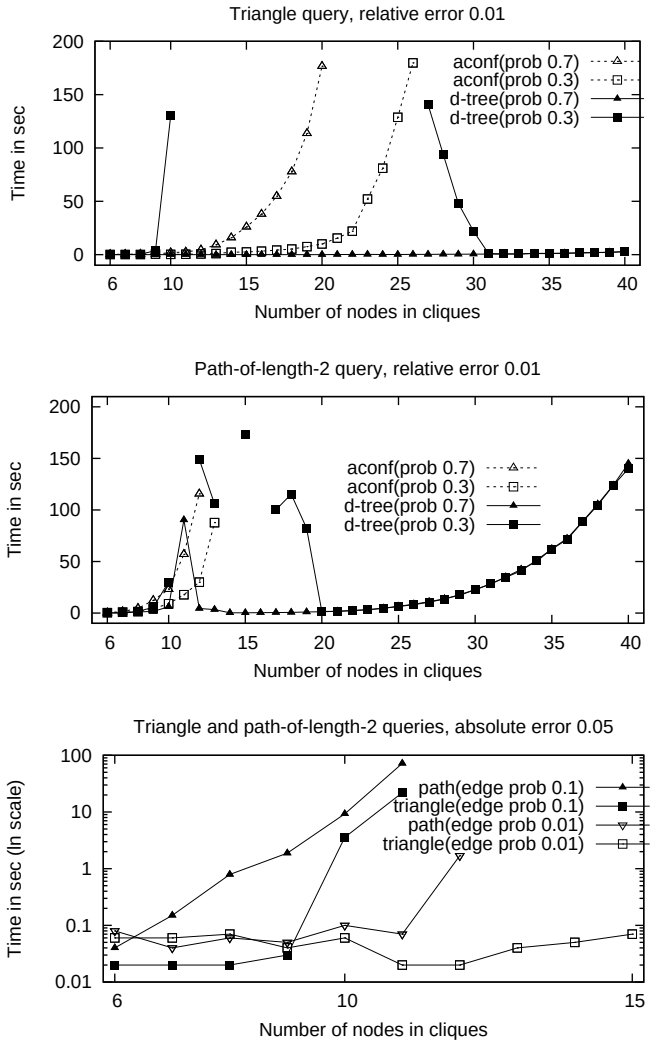


Figure 30: Experimental results for random graphs.

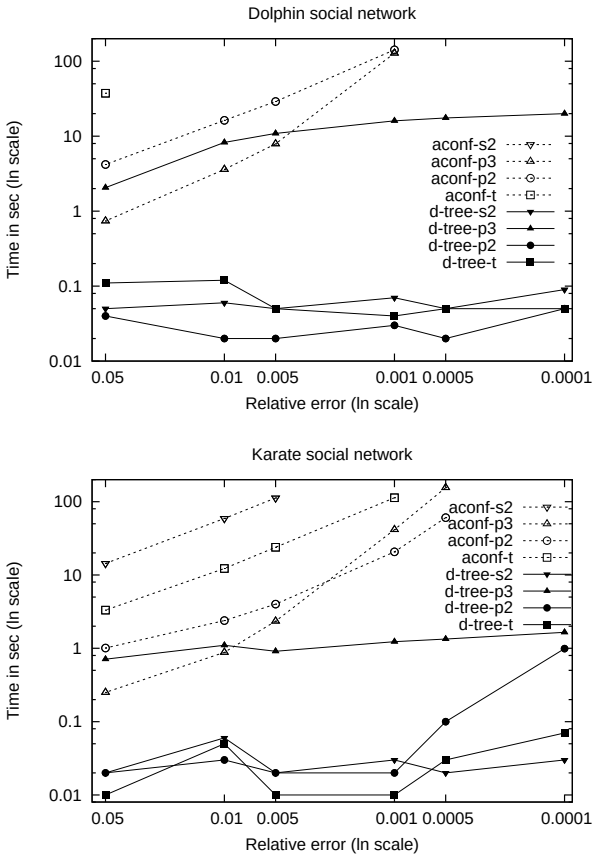


Figure 31: Experimental results for social networks.

in that a pair of nodes are friends (very credible for dolphins), or varying degrees of friendship (very credible for karatekas).

We consider four different queries. The first two, triangle (t) and "path of length 2" (p2) were discussed in the introduction. The query p3 computes the probability that the graph contains at least one path of length 3. The "separation" query (s2) computes the probability that two given nodes have at most two degrees of separation.

Our experimental results on queries on random graphs and social networks are reported in Figures 30 and 31. In case of random graphs, for large edge probabilities (above 0.5), d-tree 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 triangle and path2 queries.

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 DNF of 780 variables and 9880 clauses; the path2 and path3 queries use a three-way and eight-way self-joins, respectively.

8 Conclusions and Future Work

The two major contributions of the thesis are (1) OBDD-based confidence computation technique for tractable conjunctive queries with inequalities and without self-joins and (2) lineage decomposition using d-tree for general-purpose confidence computation.

We first give a syntactical characterization of tractable conjunctive queries with inequalities on tuple-independent probabilistic databases. For the tractable queries, we present a new secondary-storage technique for exact confidence computation based on OBDDs.

We then propose a generic deterministic approximation algorithm for confidence computation. We compile the lineage into a novel kind of decision diagrams exploiting negative correlations, independence and factored representations. We also show that in combination with our heuristic, d-trees can naturally capture the special structures of the lineage of tractable queries.

Both techniques are fully integrated into the SPROUT query engine and achieve orders of magnitude improvement over state-of-the-art exact and approximate confidence computation algorithms.

Exciting followup research can be centered around algorithms for exact and approximate confidence computation for various classes of queries and probabilistic database models. Some possible directions are listed as follows:

- Extend the tractability frontier of conjunctive queries with inequalities. The thesis gives some classes of tractable queries but up to now no boundaries between the easy and hard are found.

- Apply the tractability results and OBDD-based techniques on tuple-independent probabilistic databases to block-independent disjoint ones (BIDs). We conjecture that they are applicable to BIDs readily or only with minor adjustments.

- Find efficient approaches to accommodate self-joins. All found tractable queries exclude self-joins; however, since the use of self-joins are quite common in real applications, extending the existing framework to take self-joins into account is of significant importance.

- Develop more accurate approximation methods. We introduce in Section 5.1 two simple ways of computing lower and upper bounds for DNFs. Although d-trees outperform state-of-art techniques with these

approaches, more advanced and more accurate but still efficient techniques for bound estimation are highly desired. A potential approach is to construct maximal 1OFs with the clauses in DNFs.

- Design efficient query evaluation approaches for conditional probability computation. To compute conditional probability $P(A|B)$ of events A and B, a naive approach is to compute both $P(A)$ and $P(B)$. More sophisticated methods involving computation sharing may bring a significant performance improvement.

References

[1] E. Adar and C. Ré. "Managing Uncertainty in Social Networks". *IEEE Data Eng. Bull.*, 30(2), 2007.

[2] V. Akman. "Implementation of Karp-Luby Monte Carlo method: an exercise in approximate counting". *The Computer Journal*, **34**(3):279–282, 1991.

[3] L. Antova, T. Jansen, C. Koch, and D. Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.

[4] L. Antova, T. Jansen, C. Koch, and D. Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.

[5] L. Antova, C. Koch, and D. Olteanu. "10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information". In *Proc. ICDE*, 2007.

[6] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. "ULDBs: Databases with Uncertainty and Lineage". In *Proc. VLDB*, 2006.

[7] E. Birnbaum and E. Lozinskii. "The Good Old Davis-Putnam Procedure Helps Counting Models". *Journal of AI Research*, **10**(6):457–477, 1999.

[8] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. "Evaluating Probabilistic Queries over Imprecise Data". In *Proc. SIGMOD*, pages 551–562, 2003.

[9] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross. "An Optimal Algorithm for Monte Carlo Estimation". *SIAM J. Comput.*, **29**(5):1484–1496, 2000.

[10] N. Dalvi and D. Suciu. "Efficient Query Evaluation on Probabilistic Databases". *VLDB Journal*, **16**(4), 2007.

[11] N. Dalvi and D. Suciu. "Management of Probabilistic Data: Foundations and Challenges". In *Proc. PODS*, 2007.

[12] N. Dalvi and D. Suciu. "The Dichotomy of Conjunctive Queries on Probabilistic Structures". In *Proc. PODS*, 2007.

[13] A. Darwiche and P. Marquis. "A knowlege compilation map". *Journal of AI Research*, 17:229–264, 2002.

[14] M. Davis and H. Putnam. "A Computing Procedure for Quantification Theory". *Journal of ACM*, **7**(3):201–215, 1960.

[15] A. Deshpande and S. Madden. "MauveDB: supporting model-based user views in database systems". In *Proc. SIGMOD*, pages 73–84, 2006.

[16] N. Fuhr and T. Rölleke. "A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems". *ACM Trans. Inf. Syst.*, **15**(1):32–66, 1997.

[17] E. Grädel, Y. Gurevich, and C. Hirsch. "The Complexity of Query Reliability". In *Proc. PODS*, pages 227–234, 1998.

[18] T. J. Green and V. Tannen. "Models for Incomplete and Probabilistic Information". In *International Workshop on Incompleteness and Inconsistency in Databases (IIDB)*, 2006.

[19] J. Huang, L. Antova, C. Koch, and D. Olteanu. "MayBMS: A Probabilistic Database Management System". In *Proc. SIGMOD*, 2009.

[20] T. Imielinski and W. Lipski. "Incomplete information in relational databases". *Journal of ACM*, **31**(4), 1984.

[21] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. "MCDB: a Monte Carlo Approach to Managing Uncertain Data". In *Proc. SIGMOD*, 2008.

[22] R. M. Karp and M. Luby. "Monte-Carlo Algorithms for Enumeration and Reliability Problems". In *Proc. FOCS*, pages 56–64, 1983.

[23] R. M. Karp, M. Luby, and N. Madras. "Monte-Carlo Approximation Algorithms for Enumeration Problems". *J. Algorithms*, **10**(3):429–448, 1989.

[24] C. Koch. "Approximating Predicates and Expressive Queries on Probabilistic Databases". In *Proc. PODS*, 2008.

[25] C. Koch and D. Olteanu. "Conditioning Probabilistic Databases". *PVLDB*, 1(1), 2008.

[26] C. Koch, D. Olteanu, L. Antova, and J. Huang. *MayBMS: A Probabilistic Database System. User Manual.* http://maybms.sourceforge.net/manual/,2009.

[27] J. Li and A. Deshpande. "Consensus Answers for Queries over Probabilistic Databases". In *Proc. PODS*, 2009.

[28] X. Lian and L. Chen. "Monochromatic and Bichromatic Reverse Skyline Search over Uncertain Databases". In *Proc. SIGMOD*, pages 213–226, 2008.

[29] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, 1998.

[30] D. Olteanu and J. Huang. "Using OBDDs for Efficient Query Evaluation on Probabilistic Databases". In *Proc. SUM*, 2008.

[31] D. Olteanu and J. Huang. "Secondary-Storage Confidence Computation for Conjunctive Queries with Inequalities". In *Proc. SIGMOD*, 2009.

[32] D. Olteanu, J. Huang, and C. Koch. "Approximate Confidence Computation in Probabilistic Databases". Submitted to ICDE 2010.

[33] D. Olteanu, J. Huang, and C. Koch. "SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases". In *Proc. ICDE*, 2009.

[34] D. Olteanu, C. Koch, and L. Antova. "World-set Decompositions: Expressiveness and Efficient Algorithms". *Theoretical Computer Science*, 403(2-3), 2008.

[35] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. ICDE*, 2007.

[36] L. Trevisan. "A Note on Deterministic Approximate Counting for k-DNF". In *Proc. APPROX-RANDOM*, pages 417–426, 2004.

[37] L. Valiant. "The Complexity of Enumeration and Reliability Problems". *SIAM J. Comput.*, **8**:410–421, 1979.

[38] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[39] W. Wei and B. Selman. "A New Approach to Model Counting". In *Proc. SAT*, 2005.

[40] R. Wissman. "Tractable Queries with Inequalities on Probabilistic Databases". MSc Thesis at University of Oxford.

[41] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, **33**:452–473, 1977.

A Proofs

Theorem 3.23. Let ϕ be the lineage of any IQ query with inequality paths on any tuple-independent database. Then, we can compute a variable order π for ϕ in time $O(|\phi|\cdot\log|\phi|)$ under which the OBDD (ϕ,π) has size bounded in $|Vars(\phi)|$ and can be computed in time $O(|Vars(\phi)|)$.

Proof. (1) We first prove that the variable order algorithm in Figure 12 can be done in time $O(|\phi|\cdot\log|\phi|)$. The algorithm has two computation-intensive components, namely, lineage sorting and a loop over all tuples. The sorting takes time $O(|\phi|\cdot\log|\phi|)$. Each step in the loop takes constant time and hence the loop takes time $O(|\phi|)$ in total. Therefore, the algorithm takes time $O(|\phi|\cdot\log|\phi|+|\phi|)=O(|\phi|\cdot\log|\phi|)$.

(2) We then prove that the OBDD (ϕ,π), where π is computed with the variable order algorithm, has size bound $|Vars(\phi)|$ by showing that each variable in ϕ appears only once in the OBDD.

As discussed in Section 3.2.1, the lineage of a query with independent subqueries is the product of the independent lineage of each subqueries. The OBDDs for such queries are the concatenation of OBDDs of independent subqueries. Therefore, we next focus on IQ queries without independent subqueries. In addition, if variables $y_1,\ldots,y_n$ share exactly the same cofactor, we can factor them as $y=y_1+\ldots+y_n$ and treat y as a single variable. Therefore, we assume without loss of generality in the lineage that all variables sharing the same cofactors are factored together.

Suppose the IQ query is

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

and the lineage ϕ of Q is

$$\phi=\sum_{j_1=1}^{m}(x_{j_1}^1(\sum_{j_2=j_1}^{m}(x_{j_2}^2(\ldots(\sum_{j_n=j_{n-1}}^{m}(x_{j_n}^n))))))$$

, where $x_1^i,\ldots,x_m^i$ are variables in R_i and the cofactor of x_j^i includes that of x_{j+1}^i. Since all variables sharing the same cofactors are factored together, the number of variables involved in the lineage from each relation is the same (m in this proof). As a notation, we define a function f such that

$$f(i,j)=\sum_{j_1=j}^{m}(x_{j_1}^i(\sum_{j_2=j_1}^{m}(x_{j_2}^{i+1}(\ldots(\sum_{j_n=j_{n-1}}^{m}(x_{j_n}^n))))))$$

. As we can see, $\phi=f(1,1)$.

Since each node of variable x in OBDDs is associated with a unique formula containing x, we need to prove that given any variable x_j^i in ϕ, where $1 \leq i \leq n$ and $1 \leq j \leq m$, there is exactly one formula containing x_j^i after eliminating the variables prior to x_j^i in the variable order. To be more precise, this unique formula is

$$f(i,j) = \sum_{j_1=j}^{m}(x_{j_1}^i(\sum_{j_2=j_1}^{m}(x_{j_2}^{i+1}(...(\sum_{j_n=j_{n-1}}^{m}(x_{j_n}^n)))))).$$

We prove this by showing that under the variable order generated by the algorithm, the formulas of the nodes in the OBDD are always of the form of $f(i,j)$, where $1 \leq i \leq n$ and $1 \leq j \leq m$.

The algorithm outputs a variable order

$$x_1^1, \ldots, x_1^n, \ldots, x_m^1, \ldots, x_m^n.$$

That is, variable x_j^i is eliminated and then the variables involved in the clauses containing x_j^i and not involved in the clauses containing x_{j+1}^i follows immediately in the order of $x_j^{i+1}, \ldots, x_j^n$.

Given a formula of the form

$$\Phi = f(i,j) = \sum_{j_1=j}^{m}(x_{j_1}^i(\sum_{j_2=j_1}^{m}(x_{j_2}^{i+1}(...(\sum_{j_n=j_{n-1}}^{m}(x_{j_n}^n)))))).$$

, with the above-mentioned variable order, variable x_j^i is the next variable to be eliminated. If x_j^i is set to true, using the cofactor inclusion property, we get

$$\Phi|_{x_j^i=true} = f(i+1,j) = \sum_{j_2=j}^{m}(x_{j_2}^{i+1}(...(\sum_{j_n=j_{n-1}}^{m}(x_{j_n}^n)))).$$

If x_j^i is set to false, we get

$$\Phi|_{x_j^i=false} = f(i,j+1) = \sum_{j_1=j+1}^{m}(x_{j_1}^i(\sum_{j_2=j_1}^{m}(x_{j_2}^{i+1}(...(\sum_{j_n=j_{n-1}}^{m}(x_{j_n}^n)))))).$$

(3) With the proof in (2), we can construct the OBDD in time $O(|Vars(\phi)|)$ by pointing the low edge of the node of variable x_j^i (note that there is only one node for each variable) to the node of variable x_{j+1}^i and high edge to the node of variable x_j^{i+1}. The nodes of x_{m+1}^i are the constant node 0 and those of x_j^{n+1} are the constant node 1, where $1 \leq i \leq n$ and $1 \leq j \leq m$.

□

Theorem 3.26. Let ϕ be the lineage of any IQ query with inequality tree t on any tuple-independent database. Then, we can compute a variable order π for ϕ in time $O(|\phi| \cdot \log |\phi|)$, under which the OBDD (ϕ, π) has size and can be computed in time $O(2^{|t|} \cdot |Vars(\phi)|)$.

Proof. (1) As shown in the proof of Theorem 3.23, the variable order algorithm takes time $O(|\phi| \cdot \log |\phi| + |\phi|) = O(|\phi| \cdot \log |\phi|)$.

(2) We then prove that the OBDD (ϕ, π), where π is computed with variable order algorithm, has size bound $O(2^{|t|} \cdot |Vars(\phi)|)$ by showing that each variable in ϕ has at most $2^{|t|}$ nodes in the OBDD. As we do in the proof of Theorem 3.23, we only consider IQ queries without independent subqueries and factor together in the lineage the variables sharing the same cofactors.

Similar to the strategy in the proof of inequality paths, we prove that the formulas associated with the nodes in the OBDD are of particular forms, of which the number is at most $2^{|t|}$.

Given an inequality tree T and the lineage ϕ of the query, we construct a factored form of ϕ with T. Traverse T in a depth-first manner, factor partially ϕ for each node in the following procedure: Suppose a node N has n children and each of them has factored partial lineage $\sum_{k=1}^{m} \phi_k^1, \ldots, \sum_{k=1}^{m} \phi_k^n$, respectively, where m is the number of variables from each relation in the lineage (since variables sharing the same cofactors are factored together, the numbers of variables from each relation are identical), the factored partial lineage of N is

$$\sum_{i=1}^{m} x_i (\prod_{j=1}^{n} (\sum_{k=i}^{m} \phi_k^{i-j}))$$

. The factored partial lineage of the root is the factored form of ϕ. We now define a notation ITF: Formula Φ is of form ITF (inequality tree form) if Φ can be factored in the above-mentioned procedure or Φ is a product of ITL formulas.

Given an ITF formula ϕ of form

$$\phi = \prod_{l=1}^{n} (\sum_{i=a}^{m} x_i^l (\prod_{j=1}^{n_i^l} (\sum_{k=i}^{m} \phi_k^{l-i-j})))$$

. Wlog, we assume that variable x_a^1 is the next variable to be eliminated. If

x_a^1 is set to true, using the cofactor inclusion property, we get

$$\prod_{l=2}^{n}(\sum_{i=a}^{m} x_i^l(\prod_{j=1}^{n_i^l}(\sum_{k=i}^{m}\phi_k^{l-i-j}))) \cdot (\prod_{j=1}^{n_a^1}(\sum_{k=i}^{m}\phi_k^{1-a-j}))$$

If x_a^1 is set to false, we get

$$\prod_{l=2}^{n}(\sum_{i=a}^{m} x_i^l(\prod_{j=1}^{n_i^l}(\sum_{k=i}^{m}\phi_k^{l-i-j}))) \cdot (\sum_{i=a+1}^{m} x_i^1(\prod_{j=1}^{n_1}(\sum_{k=i}^{m}\phi_k^{1-i-j})))$$

The formulas remain ITF no matter x_a^1 is true or false.

Under the variable order π and the structure of the corresponding OBDD, the formula ϕ associated with a node N for variable x from relation R has the following property:

1 It does not contain any variables coming from the relation whose inequality column is smaller than that from R.

2 It contains variables coming from the relation whose inequality column is larger than that from R.

3 The variables from other relations, whose inequality columns are incomparable to that from R, may exist in ϕ or not.

Since the formula has to be of form ITF, the possible forms of ϕ is up to $2^{|t|}$ according to the third property. To be more precise, the OBDD reach its maximal size if in the inequality tree, all leaf nodes are children of the root.

(3) OBDDs are constructed by following confidence computation algorithm in Figure 14 and removing the redundant nodes without parents. This algorithm constructs $2^{|t|}$ nodes for each variables and takes time $O(2^{|t|} \cdot |Vars(\phi)|)$.

□

Theorem 4.10. The DNFs of hard query patterns $R(X)$, $S(X,Y), T(Y)$ are factorizable in 1OF if each connected component of the bipartite graph of S is functional or complete. □

Proof. Consider the hard query pattern: $R(X), S(X,Y), T(Y)$, where R and T are tuple-independent probabilistic relations, and S is a deterministic

relation. Assume that R has random variables $a_1, \ldots, a_{|R|}$ and T has random variables $b_1, \ldots, b_{|T|}$.

If a connected component is functional, wlog, suppose the functional dependency $X \to Y$ holds on tuples of this component (note that different functional dependencies may hold on tuples of different components). Since the component is connected, there is only one Y-value y in this component and all X-values $x', \ldots, x''$ are connected to y.

If a connected component is complete, every value $x', \ldots, x''$ in column X is connected to every value $y', \ldots, y''$ in column Y in that component.

Assume now $a_1, \ldots, a_k$ are those variables paired with tuples with X-values in $\{x', \ldots, x''\}$ in R, and $b_1, \ldots, b_l$ are those variables paired with tuples with Y-values y in case of a functional component, or $y', \ldots, y''$ in case of a complete component. Then, this component creates the following DNF in the answer to the hard pattern:

$$\bigvee_{i=1}^{k}\bigvee_{j=1}^{l}(a_i \wedge b_j) = (\bigvee_{i=1}^{k} a_i) \wedge (\bigvee_{j=1}^{l} b_j).$$

The latter form is already in 1OF.

Since unconnected components do not share any vertices and edges, DNFs of different unconnected component do not share any variables. Therefore, the DNF of the query can be factorized in $\bigvee_{i=1}^{n} \Phi_i$, where n is the number of connected components and Φ_i is the 1OF of the ith component. □

Lemma 4.14. Given a DNF Φ of an IQ query and variable $v = \text{MAX\_OCCUR}(\Phi)$. Then, $\Phi|_v$ subsumes Φ. □

Proof. Let the IQ query be $R_1(\overline{x_1}), \ldots, R_n(\overline{x_n}), \phi$.

Case 1: v is a variable from relation R_i and there exists a condition $x_{i'} < x_{j'}$ in ϕ, where $x_{i'} \in \overline{x_i}$, $x_{j'} \in \overline{x_j}$ and $1 \leq i \neq j \leq n$. Let v come from a tuple with a in column $x_{i'}$.

We prove the theorem by contradiction. Assume that there exists a clause c in $\Phi|_v$ such that the cofactor of v does not subsume c. Due to the semantics of join operations, there exists in c a variable v' coming from R_i in a tuple with a' in column $x_{i'}$. If $a \leq a'$, v must be paired with all variables in c except v' because of transitivity of $<$; however, since the subsumption does not hold, $a' < a$. Therefore, v' is paired with all variables that v is paired with and the number of occurrences of v' is larger than that of v. Contradiction.

Case 2: v is a variable from relation R_i and there exists a condition $x_{i'} > x_{j'}$ in ϕ, where $x_{i'} \in \overline{x_i}$, $x_{j'} \in \overline{x_j}$ and $1 \leq i \neq j \leq n$. This can be proved similarly as case 1. □

Proposition 5.2. Let $[L_1, U_1] = \text{Independent}(\Phi)$ and $[L_2, U_2] = \text{Exact-k}(\Phi)$ for a DNF Φ. It then holds that $L_1 \leq P(\Phi) \leq U_1$ and $L_2 \leq P(\Phi) \leq U_2$. □

Proof. Note that both heuristics compute the exact confidence for each buckets, namely, $P(B_i)$ is the exact probability of bucket B_i. We prove the case of heuristic Independent and Exact-k can be proved in the same way.

WLOG, suppose $P(B_1) = \max_{i=1}^{n} P(B_i)$. The probability of Φ is

$$P(\Phi) = P(B_1) + P(B_2 \vee ... \vee B_n) - P(B_1 \wedge (B_2 \vee ... \vee B_n))$$

Since $P(B_2 \vee ... \vee B_n) - P(B_1 \wedge (B_2 \vee ... \vee B_n)) \geq 0$, we get

$$P(\Phi) \geq P(B_1) = \max_{i=1}^{n} P(B_i)$$

The probability of Φ can also be expressed as

$$P(\Phi) = \sum_{i=1}^{n} P(B_i) - P(\bigvee_{1 \leq i < j \leq n} B_i \wedge B_j)$$

So $P(\Phi) \leq \sum_{i=1}^{n} P(B_i)$. In addition, any probability should not exceed 1. Therefore,

$$P(\Phi) \leq \min(1, \sum_{i=1}^{n} P(B_i))$$

□

Proposition 5.3. If a d-tree d for a DNF Φ has bounds $[L, U]$, then it holds that $L \leq P(\Phi) \leq U$. □

Proof. We prove it by induction.

Basis: If the node is a leaf, this is Proposition 5.2.

Inductive step: Assume that an inner node N is associated with DNF ϕ and its children $N_1, ..., N_n$ are associated with DNFs $\phi_1, ..., \phi_n$. In addition, $\forall 1 \leq i \leq n$, it holds that $L_i \leq P(\phi_i) \leq U_i$, where N_i has bounds $[L_i, U_i]$.

N may be any of $\odot$, $\otimes$ or $\oplus$. We discuss them separately below.

Case of $\odot$: Compute the exact probability of ϕ and the bounds of N.

$$P(\phi) = \prod_{i=1}^{n} P(\phi_i)$$

$$L_N = \prod_{i=1}^{n} L(\phi_i)$$

$$U_N = \prod_{i=1}^{n} U(\phi_i)$$

Under the assumption, $L_N \leq P(\phi) \leq U_N$.

Case of $\otimes$: Compute the exact probability of ϕ and the bounds of N.

$$P(\phi) = 1 - \prod_{i=1}^{n} (1 - P(\phi_i))$$

$$L_N = 1 - \prod_{i=1}^{n} (1 - L(\phi_i))$$

$$U_N = 1 - \prod_{i=1}^{n} (1 - U(\phi_i))$$

Under the assumption, $L_N \leq P(\phi) \leq U_N$.

Case of $\oplus$: Compute the exact probability of ϕ and the bounds of N.

$$P(\phi) = \sum_{i=1}^{n} P(\phi_i)$$

$$L_N = \sum_{i=1}^{n} L(\phi_i)$$

$$U_N = \sum_{i=1}^{n} U(\phi_i)$$

Under the assumption, $L_N \leq P(\phi) \leq U_N$.

Therefore, in any of the cases, it holds that

$$L_N \leq P(\phi) \leq U_N$$

□

Proposition 5.6. Given a DNF Φ, a fixed error ϵ, and a d-tree for Φ with bounds $[L, U]$.

- If $U - \epsilon \leq L + \epsilon$, then any value in $[U - \epsilon, L + \epsilon]$ is an absolute ϵ-approximation of $P(\Phi)$.
- If $(1 - \epsilon) \cdot U \leq (1 + \epsilon) \cdot L$, then any value in $[(1 - \epsilon) \cdot U, (1 + \epsilon) \cdot L]$ is a relative ϵ-approximation of $P(\Phi)$. □

Proof. Let p be the probability of Φ. According to Proposition 5.3, $p \in [L, U]$. For error ϵ it holds that $1 - \epsilon \geq 0$.

For absolute ϵ-approximation, we have that

$$p - \epsilon \leq U - \epsilon \leq \hat{p} \leq L + \epsilon \leq p + \epsilon.$$

For relative ϵ-approximation, we have that

$$(1 - \epsilon) \cdot p \leq (1 - \epsilon) \cdot U \leq \hat{p} \leq (1 + \epsilon) \cdot L \leq (1 + \epsilon) \cdot p.$$

□

Lemma 5.9. For a d-tree d, $L(d)$ is the pair of bounds $[L, U]$ that maximizes each of $U - L$ and $(1 - \epsilon) \cdot U - (1 + \epsilon) \cdot L$ over the entire bound space of d.□

Proof. Consider the point interval of each open leaf be $[x_i, x_i]$, where x_i is a distinct variable. The upper and lower bounds of d can be then expressed as functions f_U and f_L, respectively, of such variables. We show that for each such variable x, $\frac{\delta(f_U - f_L)}{\delta x} \leq 0$ and hence $f_U - f_L$ is maximized when x is minimized. That is, when $x = L$, where L is the lower bound of that open leaf.

Base case: We are at the open leaf with variable x. Let us denote by n the level of this leaf. We have $f_U^n = a_U^n \cdot x + b_U^n$ and $f_L^n = a_L^n \cdot x + b_L^n$, where $a_U^n = a_L^n = 1$ and $b_U^n = b_L^n = 0$. It then holds that

$$\frac{\delta(f_U^n - f_L^n)}{\delta x} = a_U^n - a_L^n \leq 0.$$

Assume now the property holds at a node c at level $j + 1$, and c is an ancestor of the open leaf with x. We show that the property it then also holds at the parent of c.

Case 1: The parent of c is a $\oplus$ node: $\oplus(c_1, \ldots, c_k)$, 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

$$\frac{\delta(f_U^j - f_L^j)}{\delta x} = a_U^{j+1} - a_L^{j+1} \leq 0.$$

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 $\otimes$ node: $\oplus(c_1, \ldots, c_k)$, where c is one of $c_1, \ldots, c_k$. Let

$$\alpha_L = \prod_{i=1, c_i \neq c}^{k} (1 - L(c_i)), \quad \alpha_U = \prod_{i=1, c_i \neq c}^{k} (1 - U(c_i))$$

where $L(c_i)$ and $U(c_i)$ represent the formulas for the lower and upper bounds, respectively, of node c_i. It is easy to see that $\alpha_L \leq \alpha_U$, given that $L(c_i) \leq U(c_i)$ for each node c_i. Then,

$$\begin{aligned} f_U^j &= 1 - \alpha_U \cdot (1 - f_U^{j+1}) \\ &= \alpha_U \cdot a_U^{j+1} \cdot x + 1 - \alpha_U + \alpha_U \cdot b_U^{j+1} \\ f_L^j &= 1 - \alpha_L \cdot (1 - f_L^{j+1}) \\ &= \alpha_L \cdot a_L^{j+1} \cdot x + 1 - \alpha_L + \alpha_L \cdot b_L^{j+1} \\ \frac{\delta(f_U^j - f_L^j)}{\delta x} &= \alpha_U \cdot a_U^{j+1} - \alpha_L \cdot a_L^{j+1} \leq 0. \end{aligned}$$

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 maximizes $(1 - \epsilon) \cdot U - (1 + \epsilon) \cdot L$. 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. □

Proposition 5.10. Given a d-tree d for a DNF Φ, and a fixed error ϵ. If the bounds $L(d)$ satisfy the sufficient condition for an ϵ-approximation in Proposition 5.6, then there is a refinement of d that is an ϵ-approximation of Φ. □

Proof. Suppose $L(d) = [L_d, U_d]$. According to Proposition 5.3, for any open leaf N with DNF ϕ in d, N's bounds $[L_N, U_N]$ satisfy that $L_N \leq P(\phi) \leq U_N$. If N is expanded completely, then its bounds become a point interval $[P(\phi), P(\phi)]$. If all the open leaves are expanded completely, the pair of bounds $[L', U']$ of the new d-tree d' is in the bound space of d. It follows Lemma 5.9 that $U' - L' \leq U_d - L_d$ and $(1-\epsilon) \cdot U' - (1+\epsilon) \cdot L' \leq (1-\epsilon) \cdot U_d - (1+\epsilon) \cdot L_d$. Therefore, bounds $[L', U']$ satisfy Proposition 5.6 and any value in $[L', U']$ is an ϵ-approximation of Φ.

□

B Hard TPC-H Queries

B2
```
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';
```

B9
```
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%';
```

B20
```
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';
```

B21
```
    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';
```

C Tractable TPC-H Queries

1
```
select
    l_returnflag,
    l_linestatus,
    conf()
from
    lineitem
where
    l_shipdate <= date '1998-09-01'
group by
    l_returnflag, l_linestatus;
```

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;
```

B1
```
select
    conf()
from
    lineitem
where
    l_shipdate <= date '1998-09-01';
```

B6
```
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;
```

B16
```
    select
        conf()
    from
        partsupp,
        part
    where
        part.p_partkey = partsupp.ps_partkey
        and p_brand <> 'Brand#45'
        and p_type like 'MEDIUM POLISHED%';
```

B17
```
    select
        conf()
    from
        lineitem,
        part
    where
        part.p_partkey = lineitem.l_partkey
        and p_brand = 'Brand#23'
        and p_container = 'MED BOX';
```

D Random Graph Queries

```
drop table node;
drop table inout;
drop table total_order;
drop table to_subset;
drop table edge0;
drop table no_edge0;
drop table edge;
drop table no_edge;

create table node (n integer);
insert into  node values (1);
insert into  node values (2);
insert into  node values (3);
insert into  node values (4);
......
insert into  node values (n); /* n is the number of nodes in the graph */


/* Here we specify the probability that an edge is in the graph.
   Note: The explanation above assumes that p is 0.5.
*/
create table inout (bit integer, p float);
/* prob is the probability that edge is in the graph */
insert into  inout values (1, prob);
/* (1 - prob) is the probability that edge is missing */
insert into  inout values (0, 1 - prob);

create table total_order as
(
   select n1.n as u, n2.n as v
   from node n1, node n2
   where n1.n < n2.n
);


/* This table represents all subsets of the total order over node as possible
   worlds. We use the same probability -- from inout -- for each edge, but
```

```
   in principle we could just as well have a different (independent)
   probability for each edge.
*/
create table to_subset as
(
   repair key u,v
   in (select * from total_order, inout)
   weight by p
);


create table edge0    as (select u,v from to_subset where bit=1);
create table no_edge0 as (select u,v from to_subset where bit=0);

create table edge     as (select *              from edge0);
insert into  edge        (select v as u, u as v from edge0);

create table no_edge  as (select *              from no_edge0);
insert into  no_edge     (select v as u, u as v from no_edge0);

/* Triangle query */
select
    conf()
from
    edge0 e1,
    edge0 e2,
    edge0 e3
where
    e1.v = e2.u
    and e2.v = e3.v
    and e1.u = e3.u
    and e1.u < e2.u
    and e2.u < e3.v;

/* Path-of-length-2 query */
select
    conf()
from
    edge e1,
    edge e2,
```

```
    no_edge e3
where
    e1.v = e2.u
    and e2.v = e3.u
    and e3.v = e1.u
    and e1.u <> e2.u
    and e1.u <> e3.u
    and e2.u <> e3.u;
```

E Social Network Queries

We show below the queries for Dolphin social network construction and asking for probabilities of different graph properties. Karate social network can be constructed similarly.

```
drop table Dolphin_certain;
drop table Dolphin_temp;
drop table Dolphin;
drop table Dolphin_triangle;
drop table Dolphin_path2;
drop table Dolphin_path3;
drop table Dolphin_separation2;
drop table edge;
drop table no_edge;

create table Dolphin_certain(u varchar, v varchar, p real);

COPY Dolphin_certain FROM 'the_directory_of_the_data_file' WITH DELIMITER AS E'\t';

create table Dolphin_temp(u varchar, v varchar, inout boolean, p real);

insert into Dolphin_temp select u, v, TRUE, p from Dolphin_certain;

insert into Dolphin_temp select u, v, FALSE, 1 - p from Dolphin_certain;

create table Dolphin as repair key u,v in Dolphin_temp weight by p;

create table edge as select * from Dolphin where inout = TRUE;

create table no_edge as select * from Dolphin where inout = FALSE;

/* Triangle query */
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-of-length-2 query */
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-of-length-3 query */
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 */
select
    conf()
from
    edge e1,
    edge e2
where
    e1.v = e2.u
    and e1.u <> e2.v;
```