Secondary-Storage Confidence Computation for Conjunctive Queries with Inequalities

Dan Olteanu and Jiewen Huang

Computing Laboratory, Oxford University, Oxford, OX1 3QD, UK
{dan.olteanu,jiewen.huang}@comlab.ox.ac.uk

ABSTRACT

This paper investigates the problem of efficiently computing the confidences of distinct tuples in the answers to conjunctive queries with inequalities ($<$) on tuple-independent probabilistic databases. This problem is fundamental to probabilistic databases and was recently stated open.

Our contributions are of both theoretical and practical importance. We define a class of tractable queries with inequalities, and generalize existing results on #P-hardness of query evaluation, now in the presence of inequalities.

For the tractable queries, we introduce a confidence computation technique based on efficient compilation of the lineage of the query answer into Ordered Binary Decision Diagrams (OBDDs), whose sizes are linear in the number of variables of the lineage.

We implemented a secondary-storage variant of our technique in PostgreSQL. This variant 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. Experiments with probabilistic TPC-H data show up to two orders of magnitude improvements when compared with state-of-the-art approaches.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; G.3 [**Mathematics of Computing**]: Probability and Statistics

General Terms

Algorithms, Languages, Management, Performance

Keywords

Query Processing, Decision Diagrams, Probabilistic Databases

1. INTRODUCTION

Probabilistic data management has recently drawn attention of the database research community [3, 5, 12, 10, 15]. Many applications require probabilistic data management support, including data warehousing, data integration, data cleaning, and Web data extraction. Scientific experiments frequently generate probabilistic data, such as incomplete observations or imprecise measurements. Sensor and RFID data are inherently uncertain.

We currently witness a concerted effort from the database research community to provide scalable query evaluation techniques, e.g., [5, 12, 10, 15]. To date, however, little has been achieved towards truly scalable techniques: Most of the existing systems, such as Trio[3], employ exact or approximate main-memory evaluation algorithms that do not scale. Notable exceptions are MystiQ [5] and SPROUT [15], which 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 [12] or Monte Carlo simulations using the Karp-Luby estimator [11]. 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 [2] and social networks [1].

This paper is the first to investigate the problem of efficiently computing 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) [13], 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

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 1: Tuple-independent probabilistic database and the answer to our query from the Introduction.

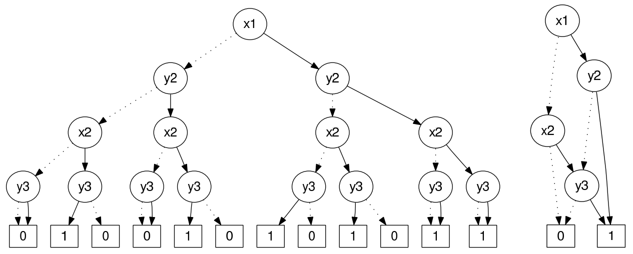


Figure 2: Decision tree (left) and OBDD (right) for the lineage of the answer tuple with DomId=1.

identifier, a domain identifier, and a date of registration for event services. The information on events includes a description and a publication date. Figure 1 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 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 [3]. 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 [5]. Given a decision tree over all variables of this lineage, as in Figure 2(left), the satisfying valuations are represented by the root-to-leaf paths that lead to a leaf labelled 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 [5]. 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 2(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, $p_x = p_y = 0$)

Step 1. $p_y = \Pr(\overline{y_3}) \cdot p_y + \Pr(y_3) \cdot \Pr(1) = \Pr(y_3) = 0.3$

Step 2. $p_x = \Pr(\overline{x_2}) \cdot p_x + \Pr(x_2) \cdot p_y = 0.06$

Step 3. $p_y = \Pr(\overline{y_2}) \cdot p_y + \Pr(y_2) \cdot \Pr(1) = 0.44$

Step 4. $p_x = \Pr(\overline{x_1}) \cdot p_x + \Pr(x_1) \cdot p_y = 0.098$

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.

The main contributions of this paper are as follows.

1. To the best of our knowledge, this paper 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 [6]. 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 define a class of hard (ie, with #P-hard data complexity) conjunctive queries with inequalities. This class includes the previously known maximal class of hard conjunctive queries without self-joins, and is defined by generalizing the notion of hierarchical queries based on equality relations on query variables to also include equivalences between query variables and so-called guards. A pair of query variables (X, Y) is a guard for a variable Z, if the inequalities $X < Z < Y$ hold in our query.

3. 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 graph of the inequality conditions present in the query, and show that for particular types of graphs, such as paths or trees, the OBDDs are linear in the number of variables in the lineage. For arbitrary condition graphs, the sizes can be expressed as polynomials with degree bounded in the size of the condition graph. This motivates our choice of OBDDs, for they can naturally exploit the structural regularity in the lineage of answers to tractable queries.

4. 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.

5. We implemented our secondary-storage algorithm and integrated it into the SPROUT query engine [15]. SPROUT is a scalable query engine for probabilistic databases that extends the query engine of PostgreSQL with special physical aggregation operators for confidence computation. It is currently under development at Oxford and publicly available at `http://maybms.sourceforge.net` as part of MayBMS [9].

6. 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 [12]. In cases when the competitors finish the computation within the allocated time, our algorithm outperforms them by up to two orders of magnitude.

The structure of the paper follows the contribution list.

2. PRELIMINARIES

2.1 Tuple-independent Probabilistic Databases

Let $\mathbf{X}$ be a finite set of (independent) Boolean random variables. *A tuple-independent probabilistic table* R^{rep} is a relation of schema $(\overline{A}, V, P)$ with functional dependencies $\overline{A} \rightarrow VP$, $V \rightarrow \overline{A}$. The values in the column V are from $\mathbf{X}$ and the values in the column P are numbers in $(0, 1]$ that represent the probabilities of the corresponding variables being true. *Certain* tables are special tuple-independent tables, where the P-value associated with each variable is 1. Figure 3 gives five probabilistic tables, out of which S and S' are certain. A probabilistic database D is a set of probabilistic tables.

A probabilistic database represents a set of instance databases, here called possible worlds, with one possible world for each total valuation of variables from $\mathbf{X}$. To obtain one instance, fix a total valuation f:

$$f = \Big(\bigwedge_{x \in \mathbf{X}:\, f(x) \text{ true}} x \Big) \wedge \bigwedge_{x \in \mathbf{X}:\, f(x) \text{ false}} \neg x).$$

Under f, the instance of each probabilistic table R^{rep} is the set of tuples $\vec{a}$ such that $(\vec{a}, x, p) \in R^{rep}$ and $f(x)$ is true. The probability of that world is the probability of the chosen total valuation f:

$$Pr[f] = \Big(\prod_{x \in \mathbf{X}:\, f(x) \text{ true}} Pr[x] \Big) \cdot \Big(\prod_{x \in \mathbf{X}:\, f(x) \text{ false}} Pr[\neg x] \Big).$$

2.2 Queries

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

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

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 3 gives four Boolean queries with inequalities.

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

$$\Pr[t \in Q(D)] = \sum_{f:\ t \in Q(D) \text{ in world } f} \Pr[f].$$

The evaluation of queries on probabilistic databases follows the standard semantics, where the columns for vari-

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

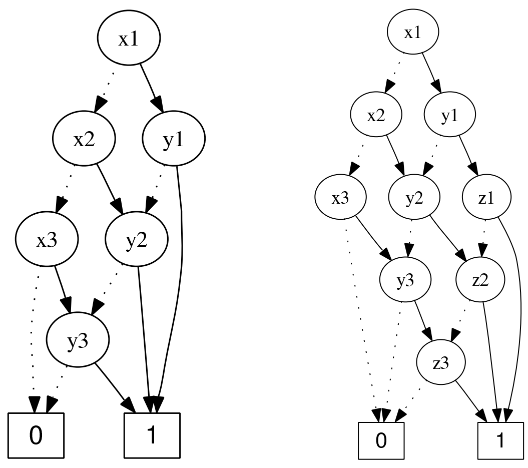


Figure 4: OBDDs used in Examples 2.2 and 4.3.

ables and probabilities are copied along in the answer tuples. These columns store relationally a DNF formula over Boolean random variables, which is commonly called *lineage*. The lineage of the answers to any of the queries in Figure 3 is $x_1y_1 + x_1y_2 + x_2y_1 + x_2y_3 + x_3y_2$. We denote the lineage of t by $\phi_{t,Q,D}$ (or $\phi_{t,Q}$ if D is clear). If Q is Boolean, we write ϕ_Q as a shorthand for $\phi_{(true),Q}$. For lineage ϕ over variable sets $\overline{x_1}, \ldots, \overline{x_n}$, we denote by $Vars(\phi) \subseteq (\overline{x_1} \cup \ldots \cup \overline{x_n})$ the set of its variables, and by $Vars_{\overline{x_1}}(\phi) \subseteq \overline{x_1}$ the set of its variables that occur in $\overline{x_1}$.

The following result is folklore.

Proposition 2.1. *For any query Q, probabilistic database D, and a distinct tuple t in $Q(D)$, $\Pr[t \in Q(D)] = \Pr[\phi_{t,Q,D}]$.*

Computing $\Pr[\phi_{t,Q,D}]$ is #P-complete in general and the goal of this paper is to define and study classes of queries for which this computation can be done efficiently.

2.3 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) are commonly used to represent compactly large Boolean expressions [13]. We find that OBBDs can naturally represent query lineage.

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 [13]. 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 2.2. Figure 4(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 exclusive 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 QUERIES

We next define classes of tractable queries with inequalities and without self-joins. The confidence computation algorithms developed in Section 4 focus on a particular class of Boolean product queries with a special form of conjunction of inequalities. Sections 3.1 through 3.4 extend the applicability of our algorithms to considerably larger query classes.

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:-$R'(E, F), T(D), T'(G, H), E < D < H$
q_2:-$R'(E, F), T(D), S(B, C), E < D, E < C$
q_3:-$R(A), T(D)$
q_4:-$R(A), T(D), R'(E, F), T'(G, H), A < E, D < E, D < G$ □

We will show in Section 4 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 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

$q(DomId)$:-$Subscr(Id, DomId, RDate), Events(Descr, PDate),$
$RDate < PDate$

has as Boolean reduct the IQ query

q':-$Subscr(Id, RDate), Events(Descr, PDate), RDate < PDate$□

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.*

3.2 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.

3.3 Equality Joins

One important class of efficient-independent queries is represented by the hierarchical queries (without self-joins) whose head variables are maximal [7].

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:

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

This query is hierarchical: subgoals(okey)⊂ subgoals(ckey), subgoals(odate)⊂ subgoals(okey), subgoals(discount)⊂ subgoals(okey), and subgoals(odate) ∩ subgoals(discount)= ∅. The query variable ckey is maximal. □

The tractable conjunctive queries without self-joins on tuple-independent databases are precisely the hierarchical ones [5]. The following result is fundamental to the evaluation of hierarchical queries and used for the generation of safe plans [5] and for query plan optimization in general [15].

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

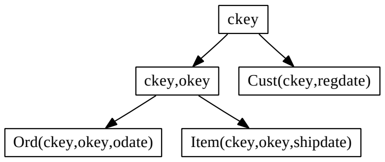


Figure 5: Tree representation of the hierarchical query of Example 3.12.

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 [14]. 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 5 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. □

3.4 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 [15].

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 [15], it follows from the chase procedure that

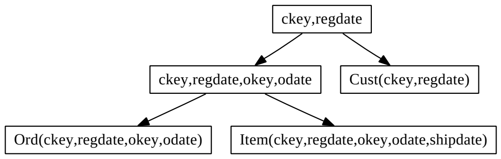

ckey,regdate
ckey,regdate,okey,odate
Cust(ckey,regdate)
Ord(ckey,regdate,okey,odate)
Item(ckey,regdate,okey,odate,shipdate)

Figure 6: Tree representation of the FD-reduct of Example 3.18.

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):

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

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

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

Query Q' is hierarchical and can be represented as in Figure 6. 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.*

3.5 Interval Conditions (IC)

Using inequalities, we can express conditions forbidden in IQ queries, such as $X < Z < Y$, with X and Y in the same subgoal and different from the subgoal of Z. We show next that such interval conditions can lead to hard queries. (However, not all queries with interval conditions are hard.)

It is known that non-hierarchical queries are hard (#P-complete) on tuple-independent databases [5]. The prototypical non-hierarchical query is Q_1 of Figure 3, because the subgoals of X and Y do not satisfy the hierarchical property: $\{R, S\} \not\subseteq \{S, T\}$, $\{S, T\} \not\subseteq \{R, S\}$, but $\{S, T\} \cap \{R, S\} \neq \emptyset$. In other words, S can express a many-to-many relationship between tuples of R and T. This query can thus create lineage representing arbitrary bipartite positive 2DNF formulas, for which model counting is #P-complete.

The above pattern of hard queries can be generalized to accommodate inequalities. For this, we first define the notions of interval conditions and variable guards.

DEFINITION 3.20. *For a given query Q, the relation $IC(Q)$ contains the pair of variables (X, Y) if*

- *X and Y occur in the same subgoal of Q, or*
- *$\exists U_1, \ldots, U_n$: $IC(Q)$ contains $(U_1, U_2), \ldots, (U_{n-1}, U_n)$, (X, U_1), and (U_n, Y), or*
- *$\exists (U, V) \in IC(Q)$: $U < X < V$ and $U < Y < V$.*

Then, the pair of variables (X, Y) guards Z *in* Q *if* $X < Z < Y$ *holds in* Q *and* $(X, Y) \in IC(Q)$.

EXAMPLE 3.21. Consider the queries of Figure 3. In Q_2 and Q_3, (E, F) guards B and A respectively, whereas (G, H) guards C and D respectively. (X, Y) guards Z in the queries

$$Q\text{:-}R(X, A), S(Y, A), T(Z), X < Z < Y.$$
$$Q'\text{:-}R_1(X, U_1), R_2(U_2, U_3), R_3(U_3, Y), R_4(Z), R_5(X_1, X_2),$$
$$X_1 < U_1 < X_2, X_1 < U_2 < X_2, X < Z < Y.$$

We explain for Q'. We first note that (X_1, X_2) guards both U_1 and U_2, hence (U_1, U_2) is in $IC(Q')$, then there is a chain of joined subgoals starting from a subgoal of X (here, R_1) and ending with a subgoal of Y (here, R_3). This implies that (X, Y) is also in $IC(Q')$, and because $X < Z < Y$ it follows that (X, Y) guards Z. □

Guards have the property that there exist databases on which the query cannot distinguish between values in the interval (X, Y) and values of Z. In particular, each pair of values (X, Y) can be close enough so as to contain precisely one distinct value of Z from its active domain. For instance, the databases given in Figure 3 establish a one-to-one mapping between the values of guards and guarded variables of queries Q_2 through Q_4. We can thus establish an equivalence relation between guards and guarded variables.

DEFINITION 3.22. *Given a variable* Z *of a query* Q. *The* equivalence class *of* Z, *denoted by* $[Z]_+$, *is the set consisting of* Z *itself, of the variables of all guards of* Z *in* Q, *and of all variables sharing guards with* Z.

We are now ready to generalize the hierarchical property.

DEFINITION 3.23 (GENERALIZATION OF DEFN. 3.11). *A query is* hierarchical *if for any two equivalence classes of its query variables, either their sets of subgoals are disjoint, or one set is contained in the other.*

EXAMPLE 3.24. All queries of Figure 3 are non-hierarchical. For query Q_1, we have $[X]_+ = \{X\}$ with subgoals $\{R, S\}$, and $[Y]_+ = \{Y\}$ with subgoals $\{S, T\}$. These sets of subgoals do not include each other and are not disjoint. For query Q_2, $[B]_+ = \{E, B, F\}$ with subgoals $\{R', S\}$, and $[C]_+ = \{G, C, H\}$ with subgoals $\{S, T'\}$. Again, these sets are not disjoint nor include each other. Consider the query

$$Q\text{:-}R_1(X_1, A_1), R_2(X_2, A_2), R_3(X_3, A_1), R_4(A_2), X_1 < X_2 < X_3.$$

Then, (X_1, X_3) guards X_2 and $[X_2]_+ = \{X_1, X_2, X_3\}$. This equivalence class covers the subgoals $\{R_1, R_2, R_3\}$, while $[A_2]_+ = \{A_2\}$ covers $\{R_2, R_4\}$. We thus conclude that Q is non-hierarchical. It can be checked that query Q' of Example 3.21 is also non-hierarchical. □

Following an argument similar to the case of hard queries without inequalities, we obtain that

THEOREM 3.25. *Non-hierarchical queries are #P-complete.*

4. 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. In the case of inequality graphs, the OBDD size remains polynomial in the number of variables, where the degree of the polynomial is the minimal number of node cuts needed to transform the inequality graph into a tree.

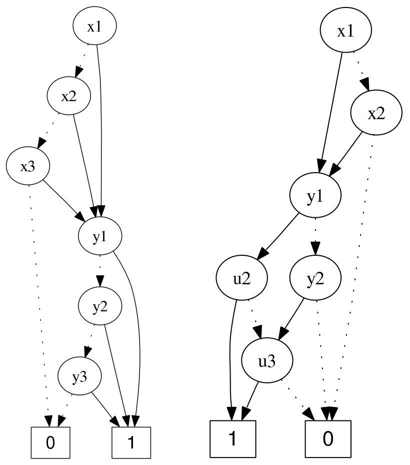


Figure 7: OBDDs used in Examples 4.2 and 4.9.

REMARK 4.1. 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).

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, 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 [14] 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. □

4.1 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 4.2. Let the query $Q\text{:-}R(A), T(D)$ on the database (R, T) of Figure 3. 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 7, is then the concatenation of the OBDDs for the two sums. □

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

4.2 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 4.3. Example 2.2 discusses how the lineage of the query Q_5 above on the database (R, T') of Figure 3 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 3, 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

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

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 4(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 $z_1z_2z_3$, 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 4.4. *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.

4.3 Queries with Inequality Trees

We generalize the results of Section 4.2 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 4.5. Consider the database consisting of tables R', T, and S of Figure 3, where we add variables z_1 to z_5 to the tuples of table S. The lineage of query Q_7 is

$$\begin{array}{l} x_1(y_1+y_2+y_3)(z_1+z_3+z_2+z_5+z_4)+ \\ x_2(y_2+y_3)(z_2+z_5+z_4)+ \\ x_3(y_3)(z_4). \end{array}$$

```
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 $\pi_{V_1,V_2,\ldots,V_n}$(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 8: Incremental computation of variable orders for IQ queries with inequality trees.

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 $x_1x_2x_3y_1z_1z_3y_2z_2z_5y_3z_4$. Figure 12 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 $y_1z_1z_3$ after x_1 and before x_2, and similarly for $y_2z_2z_5$: We then obtain $x_1y_1z_1z_3x_2y_2z_2z_5x_3y_3z_4$. 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 4.5 (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 8.

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 rediscover 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 **latestVarInVO**. 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 4.6. Consider the lineage of Example 4.5. 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 4.5 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 $y^i_1, \ldots, y^i_{m_i}$, 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^i_j. The OBDD nodes that can point to y^i_j-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 4.7. *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 12 shows two OBDDs for a fragment of the lineage of query Q_7 of Example 4.5, one as constructed by our algorithm (right), and a reduced version of it (left).

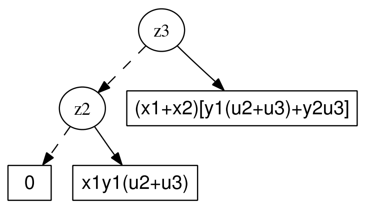


Figure 9: Partial OBDD used in Example 4.9.

4.4 Queries with Inequality Graphs

We next consider IQ queries with inequality graphs.

In case the inequality graph is cyclic, then the query is unsatisfiable, as an inequality of the form $A < B < A$ can be derived from the transtivity of inequality.

An inequality graph with several unconnected components means that the query is a product of independent subqueries, one subquery per unconnected component. This case is approached as described in Section 4.1.

In case of connected inequality graphs, the structure of the lineage for such queries differs from that of inequality trees. Our approach is to simplify the inequality graph by eliminating all its sink nodes, i.e., all nodes that have more than one incoming edge. After elimination, we are left with an inequality tree, which can be processed as presented in Section 4.3, or with several unconnected inequality trees, which can then be processed independently.

The elimination of a node from an inequality graph corresponds to the construction of a variable order where all (random) variables of the table corresponding to that node occur together and before the variables of other tables. The order of these variables has to follow the inclusion of their cofactors such that the variable x_{i+1} occurs before x_i if $f_{x_i} \subset f_{x_{i+1}}$. This order is the order of the indices of variables x_i, assuming the variables x_i are initially sorted according to the values of the attribute mapped to that inequality node.

If several nodes in the inequality graph need to be eliminated, then their elimination order is irrelevant. In case of two such nodes, say with corresponding tables R_1 and R_2, their elimination leads to variable orders starting with all the variables $x_n, \ldots, x_1$ of the table R_1, followed by all the variables $y_m, \ldots, y_1$ of the table R_2, and finally completed with the variables from the remaining tables. Because the elimination of a variable from R_1 may not affect the truth of any variable from R_2, the OBDD fragment constructed under the variable order $x_n \ldots x_1 y_m \ldots y_1$ can have size $n \cdot m$: for each variable x_i, the same variable y_j can occur under both of its branches. (Note again that variables x_l with $l < i$ cannot occur under the positive branch of x_i because their cofactors are subsumed by the cofactor of x_i.)

This property generalizes to k inequality sink nodes to be removed from an inequality graph.

THEOREM 4.8. *Let ϕ be the lineage of any IQ query on any tuple-independent database. Let g be the inequality graph, whose sink nodes correspond to tables $T_1, \ldots, T_k$. 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 $O(2^{|g|-k} \cdot |Vars(\phi)| \cdot \prod_{i=1}^{k}(|Vars_{T_i}(\phi)|))$.*

We next exemplify with an inequality graph that has one sink node.

```
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 10: Data structures used by the algorithm of Figure 11.

Example 4.9. Let the IQ query with inequality graph

$$Q_9 :\!- R(A), T(D), R'(E,F), T'(G,H), A < E, D < E, D < G$$

on the database (R, R', T, T') of Figure 3, with the modification that R' has variables z_1, z_2, z_3 and T' has variables u_1, u_2, u_3. Its inequality graph has the sink node labeled E corresponding to the table R'. The lineage is

$$x_1y_1(z_2 + z_3)(u_2 + u_3) + x_1y_2z_3u_3+$$
$$x_2y_1z_3(u_2 + u_3) + x_2y_2z_3u_3+$$
$$x_3y_2z_3u_3.$$

We remove from the inequality graph the sink node E and obtain two unconnected graphs: one isolated node A, and a path $D \rightarrow G$. This removal operation corresponds to the elimination of variables z_3 and z_2 from the lineage. The formulas remaining after these variable eliminations can then be (separately) compiled as for unconnected graphs. The partial OBDD structure obtained by eliminating the variables z_3 and z_2 is shown in Figure 9. We also show the OBDD for the cofactor of z_3 in Figure 7(right). □

5. CONFIDENCE COMPUTATION IN SECONDARY STORAGE

This section introduces a secondary-storage algorithm for confidence computation for IQ queries with inequality trees. For queries with arbitrary inequality graphs, we follow the node elimination algorithm given in Section 4.4, which transforms arbitrary inequality graphs into (possibly unconnected) trees. We then apply the algorithm of this section.

This algorithm is in essence our algorithm for incremental computation of variable orders for queries with inequality trees given in Figure 8 of Section 4. 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 11 and uses data structures described in Figure 10: The code in the topmost box should replace the inner box of the algorithm for variable order computation given in Figure 8.

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 4.3, 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.

```
Code to replace the inner box in Figure 8:
    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;
```

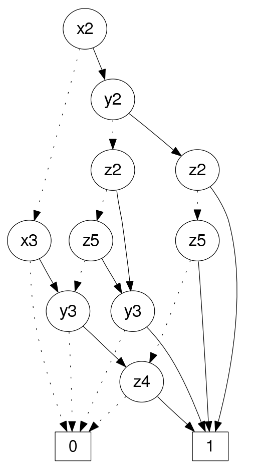
```
get_node(bool vector bv, int i, bool is_true)
bv[i] = is_true; crtLevel = the first Level in obdd_levels;
while(true)
    if (crtLevel.index = 0 OR (!bv[crtLevel.index]
       AND all_ancestors_set_true(bv, crtLevel.index)))
       foreach OBDD_Node no in crtLevel.nodes do
          if (no.bv = bv)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

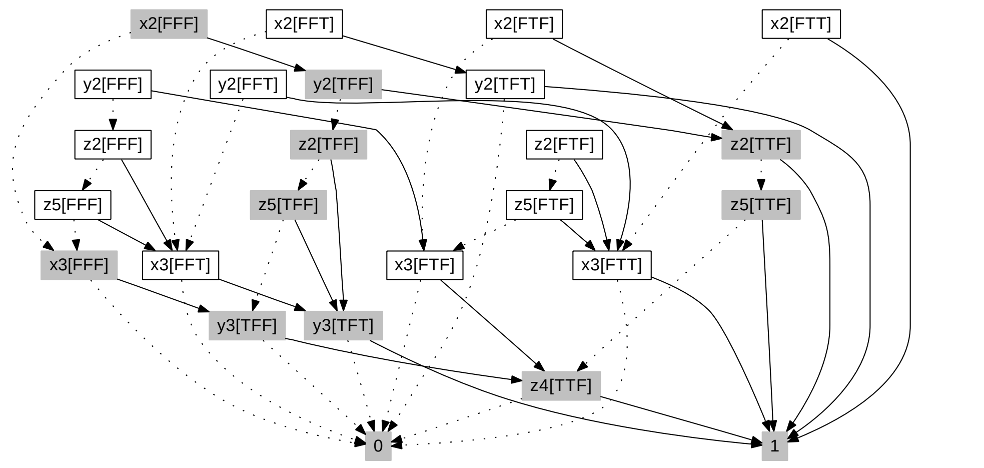
```
all_ancestors_set_true(bool vector bv, int i)
foreach ancestor A of get_ineqtree_node_with_index(i) do
    if (!bv[A.index]) return false;
return true;
```

Figure 11: Secondary-storage algorithm for confidence computation.

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 12, 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 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.





Left: Reduced OBDD used in Section 4. Right: Partially reduced OBDD as constructed by the algorithm of Section 5. 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 12: OBDDs for the lineage discussed in Example 4.5.

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 11.

Example 5.1. We show how to compute the probability of the lineage of query Q_7 of Example 4.5. Figure 12 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 12) 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 $\Pr(x_3)$ × (node with vector [T,F,F] in y_3 level).p + $\Pr(\overline{x}_3)$ × 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. □

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 13: Queries used in the experiments.

6. EXPERIMENTS

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.

Prototype. We implemented our secondary-storage algorithm and integrated it into SPROUT [15]. SPROUT is a scalable query engine for probabilistic databases that extends the query engine of PostgreSQL with a new physical aggregation operator for confidence computation.

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.

Queries. We evaluated the six queries shown in Figure 13. The aggregate construct `conf()` specifies confidence computation of distinct tuples in the query answer.

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 [9], which is a publicly available extension of the PostgreSQL backend (`http://maybms.sourceforge.net`).

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 14: Lineage Characteristics (scale factor 1).

The first algorithm (denoted by "conf") is an exact confidence computation algorithm with good behaviour on randomly generated data [12]. It compiles the lineage into a weak form of d-NNF (decomposable negation normal form) on which probability computation can be done linearly [8].

The second algorithm (denoted by "aconf") is a Monte Carlo simulation for confidence computation [16, 5] based on the Karp-Luby (KL) fully polynomial randomized approximation scheme for DNF counting [11]. 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 [12], we used in the experiments the optimal Monte-Carlo estimation algorithm of [4].

In contrast to our technique, both competitors can process lineage of arbitrary queries on complete probabilistic database models (such as U-relations [12]), 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 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 with a warm cache obtained by running a query once and then reporting the average runtime over five subsequent, identical executions. 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).

Sizes of query lineage. We experimented with queries that produce large lineage. Figure 14 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 15 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.

Comparison with state-of-the-art algorithms. Figure 15 shows the results of our experimental comparison. For

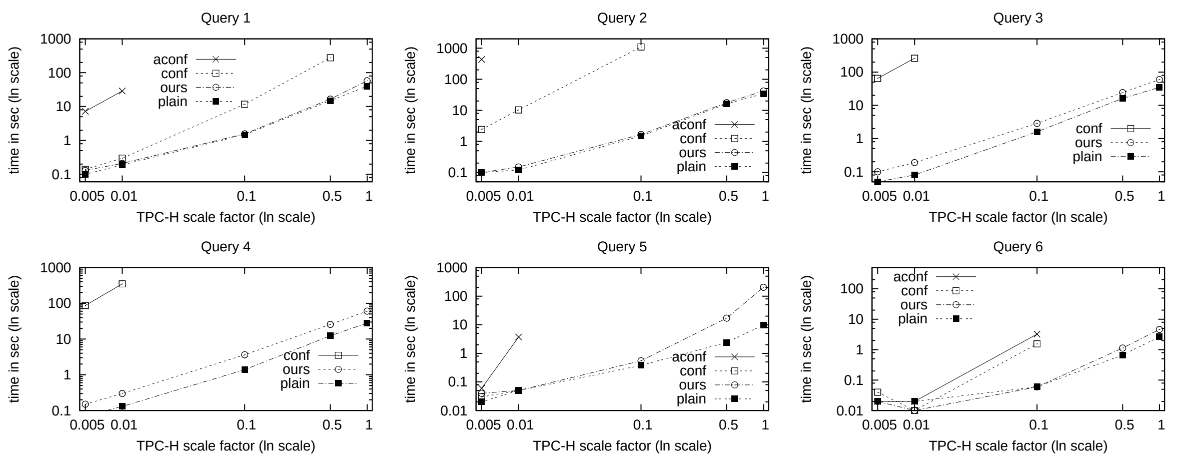


Figure 15: Effect of varying the scale factor on query evaluation using our technique, plain, conf, and aconf.

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 [4] 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.

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. CONCLUSION AND FUTURE WORK

This paper gives a syntactical characterization of (in)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. This technique is fully integrated into the SPROUT query engine and achieves orders of magnitude improvement over state-of-the-art exact and approximate confidence computation algorithms.

Exciting followup research can be centered around secondary-storage algorithms for exact and approximate confidence computation for various classes of queries and probabilistic database models. Of foremost importance is to chart the tractability frontier of conjunctive queries with inequalities.

8. REFERENCES

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

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

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

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

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

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

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

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

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

[10] 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.

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

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

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

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

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

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