Faster Query Answering in Probabilistic Databases using Read-Once Functions

Sudeepa Roy Vittorio Perduca Val Tannen

University of Pennsylvania
{sudeepa, perduca, val}@cis.upenn.edu

ABSTRACT

A boolean expression is in *read-once form* if each of its variables appears exactly once. When the variables denote independent events in a probability space, the probability of the event denoted by the whole expression in read-once form can be computed in polynomial time (whereas the general problem for arbitrary expressions is #P-complete). Known approaches to checking read-once property seem to require putting these expressions in disjunctive normal form. In this paper, we tell a better story for a large subclass of boolean event expressions: those that are generated by conjunctive queries without self-joins and on tuple-independent probabilistic databases.

We first show that given a tuple-independent representation and the *provenance graph* of an SPJ query plan without self-joins, we can, without using the DNF of a result event expression, efficiently compute its *co-occurrence graph*. From this, the read-once form can already, if it exists, be computed efficiently using existing techniques. Our second and key contribution is a complete, efficient, and simple to implement algorithm for computing the read-once forms (whenever they exist) directly, using a new concept, that of *co-table graph*, which can be significantly smaller than the co-occurrence graph.

Categories and Subject Descriptors

H.2.m [**Database Management**]: Miscellaneous—*Database Theory*

General Terms

Algorithms, Theory

Keywords

probabilistic databases, read-once functions

1. INTRODUCTION

The computation of distributions for query answers on probabilistic databases is closely related to the manipulation of boolean formulas. This connection has led both to interesting theoretical questions and to implementation opportunities. In this paper we consider the *tuple-independent* model [1] for probabilistic databases. In such a model probabilistic databases are represented by tables whose tuples t are each annotated by a probability value $p_t > 0$, see Fig. 1(a). Each tuple appears in a possible world (instance) of the representation with probability p_t independently of the other tuples. This defines a probability distribution on all possible instances.

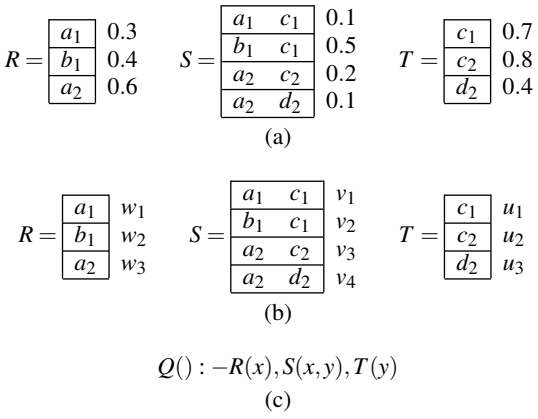

Figure 1: (a) A tuple-independent probabilistic database. (b) Event table representation. (c) An *unsafe* query.

Manipulating all possible instances is impossibly unwieldy so techniques have been developed [27, 11, 37] for obtaining the query answers from the much smaller representation tables. This is where boolean formulas make their entrance. The idea, by now well-understood [8, 2, 10, 1], is to define the relational algebra operators on tables whose tuples are annotated with event expressions. The event expressions are boolean expressions whose variables annotate the tuples in the input tables. The computation of event expressions is the same as that used in c-tables [22] as models for incomplete and probabilistic databases are closely related [16]. Once the event expressions are computed for the tuples in the representation table of the query answer (which is in general *not* tuple-independent), probabilities are computed according to the standard laws.

The event expressions method was called *intensional* "semantics" by Fuhr and Rölleke and they observed that with this method computing the query answer probabilities seems to require exponentially many steps in general [11]. Indeed, the data complexity [2]

ICDT 2011, March 21–23, 2011, Uppsala, Sweden.

[1] This model has been considered as early as [4] as well as in, eg., [11, 37, 13, 8].

[2] Here, and throughout the paper, the data input consists of the

of query evaluation on probabilistic databases is #P-complete, even for conjunctive queries [13], in fact even for quite simple boolean queries [8] such as the query in Fig. 1(c).

But Fuhr and Rölleke also observe that certain event independences can be taken advantage of, when present, to compute answer probabilities in PTIME, with a procedure called *extensional* "semantics". The idea behind the extensional approach is the starting point for the remarkable results [8, 9] of Dalvi and Suciu who discovered that the conjunctive queries can be decidably and elegantly separated into those whose data complexity is #P-complete and those for whom a *safe plan* taking the extensional approach can be found.

Our starting point is the observation that even when the data complexity of a query is #P-complete (i.e. the query is *unsafe* [8]), there may be classes of data inputs for which the computation can be done with the extensional approach, and is therefore in PTIME. We illustrate with a simple example.

EXAMPLE 1. *Consider the tuple-independent probabilistic database and the conjunctive query Q in Fig. 1. Since the query Q is boolean it has just one possible answer and the event expression annotating it is* [3]

$$f = w_1v_1u_1 + w_2v_2u_1 + w_3v_3u_2 + w_3v_4u_3 \quad (1)$$

This was obtained with the standard plan $\pi_{()}((R \bowtie S) \bowtie T)$. *However, it is equivalent to another boolean expression*

$$(w_1v_1 + w_2v_2)u_1 + w_3(v_3u_2 + v_4u_3) \quad (2)$$

which has the property that each variable occurs exactly once.

Event (boolean) expressions in which each variable occurs exactly once are in *read-once form* (see [28]). For read-once forms the events denoted by non-overlapping subexpressions are jointly independent, so we can use the key idea of the extensional approach:

Fact If events $E_1, \ldots, E_n$ are jointly independent then

$$P(E_1 \cap \cdots \cap E_n) = P(E_1) \cdots P(E_n) \quad (3)$$
$$P(E_1 \cup \cdots \cup E_n) = 1 - [1 - P(E_1)] \cdots [1 - P(E_n)]. \quad (4)$$

EXAMPLE 2 (EXAMPLE 1 CONTINUED). *The probability of the answer (2) can be computed as follows*

$$P(f) = P((w_1v_1 + w_2v_2)u_1 + w_3(v_3u_2 + v_4u_3)) = 1 - [1 - P(w_1v_1 + w_2v_2)P(u_1)][1 - P(w_3)P(v_3u_2 + v_4u_3)]$$

where

$$P(w_1v_1 + w_2v_2) = 1 - [1 - P(w_1)P(v_1)][1 - P(w_2)P(v_2)]$$

and

$$P(v_3u_2 + v_4u_3) = 1 - [1 - P(v_3)P(u_2)][1 - P(v_4)P(u_3)].$$

We can extend this example to an entire class of representation tables of unbounded size. For each n, the relations R,T will have $3n$ *tuples while S will have* $4n$ *tuples, and the probability of the answer can be computed in time* $O(n)$ *[33]. It is also clear that there is no relational algebra plan that directly yields (2) above.*

In fact, Fuhr and Rölleke (see [11], Thm.4.5) state that probabilities can be computed by "simple evaluation" (i.e., by the extensional method) if and only if the event expressions computed intensionally are in read-once form. Moreover, the *safe plans* of [8] are such that all the event expressions computed with the intensional method both on intermediary relations and on the final answer, are in read-once form.

But more can be done. Notice that the expression (1) is not in read-once form but it is equivalent to (2) which is. Boolean expressions that are equivalent to read-once forms have been called by various names, eg., separable, fanout-free [20], repetition-free [17], μ-expressions [36], non-repeating [32], but since the late 80's [21] the terminology seems to have converged on *read-once*. Of course, not all boolean expressions are read-once, eg., $xy + yz + zx$ or $xy + yz + zu$ are not.

With this motivation we take the study of the following problem as the goal of this paper:

Problem Given tuple-independent database I and boolean conjunctive query Q, when is $Q(I)$ read-once and if so, can its read-once form be computed efficiently?

It turns out that [12] gives a fast algorithm that takes a formula in irredundant disjunctive normal form, decides whether it is read-once, and if it is, computes the read-once form (which is in fact unique modulo associativity and commutativity). The algorithm is based upon a characterization in terms of the formula's co-occurrence graph given in [18].

Some terminology (taken up again in Section 2). Since we don't have anything to say about negation or difference in queries we work only with *monotone* boolean formulas (all literals are positive, only disjunction and conjunction operations). Disjunctive normal forms (DNFs) are disjunctions of *implicants*, which in turn are conjunctions of *distinct* variables. A *prime* implicant of a formula E is one with a minimal set of variables among all that can appear in DNFs equivalent to E. By absorption, we can retain only the prime implicants. The result is called an *irredundant* DNF (IDNF) of E, and is unique modulo associativity and commutativity. The *co-occurrence graph* of a boolean formula E has its variables as nodes and has an edge between x and y iff they both occur in the same prime implicant of E.

For positive relational queries, the size of the IDNF of the boolean event expressions is polynomial in the size of the table, but often (and necessarily) exponential in the size of the query. This is a good reason for avoiding the explicit computation of the IDNFs, and in particular for not relying on the algorithm in [12]. In recent and independent work Sen et al. [34] proved that for the boolean expressions that arise out of the evaluation of conjunctive queries without self-joins the characterization in [18] can be simplified and one only needs to test whether the co-occurrence graph is a "co-graph" [5] which can be done [4] in linear time [6].

It is also stated [34] that even for conjunctive queries without self-joins computing co-occurrence graphs likely requires obtaining the IDNF of the boolean expressions. One of our contributions in this paper is to show that an excursion through the IDNF is in fact not necessary because the co-occurrence graphs can be computed directly from the *provenance graph* [25, 14] that captures the computation of the query on a table. Provenance graphs are DAG representations of the event expressions in such a way that most common subexpressions for the entire table (rather than just each tuple) are not replicated. The smaller size of the provenance graphs likely provides practical speedups in the computations (compared for example with the provenance trees of [34]). Moreover, our ap-

representation tables [13, 8] rather than the collection of possible worlds.

[3] To reduce the size of expressions and following established tradition we use $+$ for $\vee$ and $\cdot$ for $\wedge$, and we even omit the latter in most terms.

[4] Defining cographs seems unnecessary for this paper. It suffices to point out that most cograph recognition algorithms produce (if it exists) something called a "cotree" (sigh) which in the case of co-occurrence graphs associated to boolean formulas is exactly a read-once form!

proach may be applicable to other kinds of queries, as long as their provenance graphs satisfy a simple criterion that we identify.

To give more context to our results, we also note that Hellerstein and Karpinski [21] have shown that if $RP \neq NP$ then deciding whether an arbitrary monotone boolean formula is read-once cannot be done in PTIME in the size of the formula.

The restriction to conjunctive queries without self-joins further allows us to contribute improvements even over an approach that composes our efficient computation of co-occurrence graphs with one of the linear-time algorithms for cograph recognition [6, 19, 3]. Indeed, we show that only a certain subgraph of the co-occurrence graph (we call it the *co-table* graph) is relevant for our stated problem. The co-table graph can be asymptotically smaller than the co-occurrence graph for some classes of queries and instances. To enable the use of only part of the co-occurrence graph we contribute a novel algorithm that computes (when they exist) the read-once forms, using two new ideas: *row decomposition* and *table decomposition*. Using just connectivity tests (eg., DFS), our algorithm is simpler to implement than the cograph recognition algorithms in [6, 19, 3] and it has the potential of affecting the implementation of probabilistic databases.

Moreover, the proof of completeness for our algorithm does not use the cograph characterization on which [34] relies. As such, the algorithm itself provides an alternative new characterization of read-once expressions generated by conjunctive queries without self-joins. This may provide useful insights into extending the approach to handle larger classes of queries.

Having rejected the use of co-occurrence graphs, Sen et al. [34] provide a different approach that derives efficiently the read-once form directly from the computations of the trees underlying the boolean expressions, so called "lineage trees", by merging read-once forms that correspond to partial formulas. They provide a complexity analysis only for one of the steps that their algorithm applies repeatedly. However, to the best of our understanding of the asymptotic complexity of their algorithm, it appears that our algorithm is faster at least by a multiplicative factor of k^2 where k is the number of tables, and the benefit can often be more.

It is also important to note that neither the results of this paper, nor those of [34] provide complexity dichotomies as does, eg. [8]. It is easy to give a family of probabilistic databases for which the query in Fig. 1(c) generates event expressions of the following form:

$$x_1x_2 + x_2x_3 + \cdots + x_{n-1}x_n + x_nx_{n+1}.$$

These formulas are not read-once, but with a simple memoization (dynamic programming) technique we can compute their probability in time linear in n [33].

Roadmap. In Section 2 we review definitions, explain how to compute provenance DAGs for SPJ queries, and compare the sizes of the co-occurrence and co-table graphs. Section 3 presents a characterization of the co-occurrence graphs that correspond to boolean expressions generated by conjunctive queries without self-joins. The characterization uses the provenance DAG. With this characterization we give an efficient algorithm for computing the co-table (and co-occurrence) graph. In Section 4 we give an efficient algorithm that, using the co-table graph, checks if the result of the query is read-once, and if so computes its read-once form. Putting together these two algorithms we obtain an efficient query-answering algorithm that is *complete* for boolean conjunctive queries without self-joins and tuple-independent databases that yield read-once event expressions. In Section 5 we compare the time complexity of this algorithm with that of Sen et al. [34], and other approaches that take advantage of past work in the read-once and cograph literature. Related work, conclusions and ideas for further work ensue. Due to space limitations, we give the details of the proofs of our results in the full version of the paper [33].

2. PRELIMINARIES

A **tuple-independent probabilistic database** is represented by a usual (set-)relational database instance I in which, additionally, every tuple is annotated with a probability in $(0,1]$, see for example Fig. 1(a). We call this the *probability table representation*. We will denote by $\mathbf{R} = \{R_1, \ldots, R_k\}$ the relational schema of the representation. By including/excluding each tuple independently with probability of its annotation, the representation defines a set of $\mathbf{R}$-instances called *possible worlds* and the obvious probability distribution on this set, hence a discrete probability space. For a given tuple $t \in R_i$ this space's event "t occurs" (the set of possible worlds in which t occurs) has probability exactly the annotation of t in the representation. Following the intensional approach [27, 11, 37] we also consider the *event table representation* which consists of the same tables, but in which every tuple is annotated by its unique tuple id, for example see Fig. 1(b).

The **tuple ids** play three distinct but related roles: (1) they identify tuples uniquely over *all* tables and in fact we will often call the tuple ids just tuples, (2) they are boolean variables, (3) they denote the events "the tuple occurs" in the probability space of all possible worlds. The last two perspectives can be combined by saying that the tuple ids are boolean-valued random variables over said probability space. Moreover, an *event expression* is a boolean expression with the tuple ids as variables.

The intensional approach further defines the semantics of the **relational algebra operators** on event tables, i.e., relational instances in which the tuples are annotated with event expressions. In this paper we will only need monotone boolean expressions because our queries only use selection, projection and join and these operators do not introduce negation. Otherwise, joins produce conjunctions, projections produce disjunctions, and selections erase the non-compliant tuples. It is worth observing that the relational algebra on event tables is essentially a particular case of the algebra on c-tables [22], and precisely a particular case of the relational algebra on semiring-annotated relations [15].

Since by now they are well understood (see the many papers we cited so far), we do not repeat here the definition of select, project, and join on tables annotated with boolean event expressions but instead we explain how they produce **provenance graphs**. The concept that we define here is a small variation on the provenance graphs defined in [25, 14] where conjunctive queries (part of mapping specifications) are treated as a black box. It is important for the provenance graphs used in this paper to reflect the structure of different SPJ query plans that compute the same conjunctive query.

A provenance graph (PG) is a directed acyclic graph (DAG) H such that the nodes $V(H)$ of H are labeled by variables or by the operation symbols $\cdot$ and $+$. As we show below, each node corresponds to a tuple in an event table that represents the set of possible worlds of either the input database or some intermediate database computed by the query plan. An edge $u \to v$ is in $E(H)$ if the tuple corresponding to u is computed using the tuple corresponding to v in either a join (in which case u is labeled with $\cdot$) or a projection (in which case u is labeled with $+$). The nodes with no outgoing edges are those labeled with variables and are called leaves while the nodes with no incoming edges are called roots (and can be labeled with either operation symbol). Provenance graphs (PGs) are closely related to the *lineage trees* of [34]. In fact, the lineage trees are tree representations of the boolean event expressions, while PGs

are more economical: they represent the same expressions but without the multiplicity of common subexpressions. Thus, they are associated with an entire table rather than with each tuple separately, each root of the graph corresponding to a tuple in the table. [5]

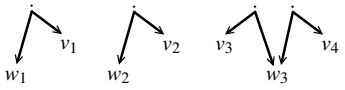

Figure 2: Provenance graph for $R \bowtie S$.

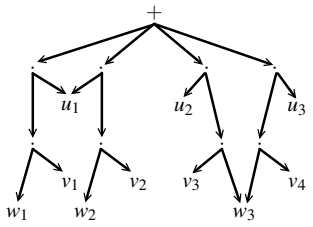

Figure 3: Provenance graph for $\pi_{()}((R \bowtie S) \bowtie T)$.

We explain how the SPJ algebra works on tables with PGs. If tables R_1 and R_2 have PGs H_1 and H_2 then the PG for $R_1 \bowtie R_2$ is constructed as follows. Take the disjoint union H of H_1 and H_2. For every $t_1 \in R_1$ and $t_2 \in R_2$ that do join, add a new root labeled with $\cdot$ and make the root of H_1 corresponding to t_1 and that of H_2 corresponding to t_2 children of this new root. Afterwards, delete (recursively) any remaining roots from H_1 and H_2. For example, referring again to Fig. 1, the PG associated with the table computed by $R \bowtie S$ is shown in Fig. 2.

For selection, delete (recursively) the roots that correspond to the tuples that do not satisfy the selection predicate. For projection, consider a table T with PG H and X a subset of its attributes. The PG for $\pi_X R$ is constructed as follows. For each $t \in \pi_X R$, let $t_1, \ldots, t_m$ be all the tuples in R that X-project to t. Add to H a new root labeled with $+$ and make the roots in H corresponding to $t_1, \ldots, t_m$ the children of this new root. Referring again to Fig. 1, the PG associated with the result of the query plan $\pi_{()}((R \bowtie S) \bowtie T)$ is shown in Fig. 3. Since the query is boolean, this PG has just one root.

The boolean event expressions that annotate tuples in the event tables built in the intensional approach can be read off the provenance graphs. Indeed, if t occurs in an (initial, intermediate, or final) table T whose PG is H, then, starting at the root u of H corresponding to t, traverse the subgraph induced by all the nodes reachable from u and build the boolean expression recursively using parent labels as operation symbols and the subexpressions corresponding to the children as operands. For example, we read $(((w_1 \cdot v_1) \cdot u_1) + (u_1 \cdot (w_2 \cdot v_2)) + (u_2 \cdot (v_3 \cdot w_3)) + ((w_3 \cdot v_4) \cdot u_3))$ off the PG in Fig. 3.

The focus of this paper is the case when the boolean (event) expressions are **read-once**, i.e., they are equivalent to expressions in which every variables occurs exactly once, the latter said to be in *read-once form*. For boolean expressions that are read-once, the read-once form is unique (modulo associativity and commutativity) [6]. The interest in read-once formulas derives from the fact that in tuple-independent databases the tuples in the input representation occur *independently* in possible worlds. More complex boolean expressions denote events whose probability needs to be computed from the probabilities of the variables, i.e., the probabilities of the independent "tuple occurs" events in the input. When such event expressions are in read-once form their probability can be computed efficiently in linear time in the number of variables using the rules (3) and (4) in Section 1.

Given a tuple-independent probabilistic database and an SPJ query plan, hence the resulting provenance graph, our objective in this paper is to decide efficiently when the boolean expression(s) read off the PG are read-once, and when they are, to compute their read-once form(s) efficiently, hence the associated probability(es).

In this paper we consider only **boolean conjunctive queries**. We can do this without loss of generality because we can associate to a non-boolean conjunctive query Q and an instance I a set of boolean queries in the usual manner: for each tuple t in the answer relation $Q(I)$, consider the boolean conjunctive query Q_t which is obtained from Q by replacing the head variables with the corresponding values in t. Note that the PGs that result from boolean queries have *exactly one root*. We will also use $Q(I)$ to denote the boolean expression generated by evaluating the query Q on instance I, which may have different (but equivalent) forms based on the query plan.

Moreover, we consider only queries **without self-join**. Therefore our queries have the form

$$Q() : -R_1(\mathbf{x}_1), \ldots, R_k(\mathbf{x}_k)$$

where $R_1, \ldots, R_k$ are all *distinct* table names while the $\mathbf{x}_i$'s are tuples of FO variables [7] or constants, possibly with repetitions, matching the arities of the tables. If the database has tables that do not appear in the query, they are of no interest, so we will always assume that our queries feature all the table names in the database schema $\mathbf{R}$.

As we have stated above, we only need to work with *monotone* boolean formulas (all literals are positive, only disjunction and conjunction operations). Every such formula is equivalent to (many) **disjunctive normal forms** (DNFs) which are disjunctions of conjunctions of variables. These conjunctions are called *implicants* for the DNF. By idempotence we can take the variables in an implicant to be distinct and the implicants of a DNF to be distinct from each other. A *prime* implicant of a formula f is one with a minimal set of variables among all that can appear in DNFs equivalent to f. By absorption, we can retain only the prime implicants in a DNF. The result is called *the irredundant* DNF (IDNF) of f, as it is uniquely determined by f (modulo associativity and commutativity). We usually denote it by f_{IDNF}. Note that in particular the set of prime implicants is uniquely determined by f.

The **co-occurrence graph**, notation G_{co}, of a boolean formula f is an undirected graph whose set of vertices $V(G_{co})$ is the set $\mathtt{Var}(f)$ of variables of f and whose set $E(G_{co})$ of edges is defined as follows: there is an edge between x and y iff they both occur in the same prime implicant of f. Therefore, G_{co} is uniquely determined by f and it can be constructed from f_{IDNF}. This construction is quadratic in the size of f_{IDNF} but of course f_{IDNF} can be exponentially larger than f. Fig. 4 shows the co-occurrence graph for the boolean expression f in equation (1) of Example 1. As this figure shows, the co-occurrence graphs for expressions generated by conjunctive queries without self join are always *k-partite*[8] graphs

[5] Note that to facilitate the comparison with the lineage trees the edge direction here is the opposite of the direction in [25, 14].

[6] This seems to have been known for a long time. We could not find an explicitly stated theorem to this effect in the literature, but, for example, it is clear that the result of the algorithm in [12] is uniquely determined by the input.

[7] FO (first-order) is to emphasize the distinction between the variables in the query subgoals and the variables in the boolean expressions.

[8] A graph $(V_1 \cup \cdots \cup V_k, E)$ is *k-partite*, if for any edge $(u,v) \in E$ where $u \in V_i$ and $v \in V_j$, $i \neq j$.

on tuple variables from k different tables.

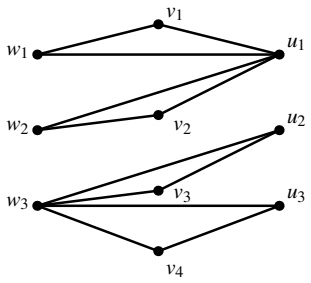

Figure 4: G_{co} for f in Example 1.

We are interested in the co-occurrence graph G_{co} of a boolean formula f because it plays a crucial role in f being read-once. Indeed [18] has shown that a monotone f is read-once iff (1) it is "normal" and (2) its G_{co} is a "cograph". We don't need to discuss normality because [34] has shown that for formulas that arise from conjunctive queries without self-joins it follows from the cograph property. We will also avoid defining what a cograph is (see [5, 6]) except to note that cograph recognition can be done in linear time [6, 19, 3] and that when applied to the co-occurrence graph of f the recognition algorithms also produce, in effect, the read-once form of f, when it exists.

Although the co-occurrence graph of f is defined in terms of f_{IDNF}, we show in Section 3 that when f is the event expression produced by a boolean conjunctive query without self-joins then we can efficiently compute the G_{co} of f from the provenance graph H of any plan for the query. Combining this with any of the cograph recognition algorithms we just cited, this yields one algorithm for the goal of our paper, which we will call a **cograph-help algorithm**.

Because it uses the more general-purpose step of cograph recognition a cograph-help algorithm will not fully take advantage of the restriction to conjunctive queries without self-joins. Intuitively, with this restriction there may be lots of edges in G_{co} that are irrelevant because they link tuples that are not joined by the query. This leads us to the notion of co-table graph defined below.

Toward the definition of the co-table graph we also need that of **table-adjacency graph**, notation G_T. Given a boolean query without self-joins $Q() : -R_1(\mathbf{x}_1),\ldots,R_k(\mathbf{x}_k)$ the vertex set $V(G_T)$ is the set of k table names $R_1,\cdots,R_k$. We will say that R_i and R_j are *adjacent* iff $\mathbf{x}_i$ and $\mathbf{x}_j$ have at least one FO variable in common i.e., R_i and R_j are joined by the query. The set of edges $E(G_T)$ consists of the pairs of adjacent table names. The table-adjacency graph G_T for the query in Example 1 is depicted in Fig. 5.

The table-adjacency graph G_T helps us remove edges irrelevant to a query from the graph G_{co}. For example, if there is an edge between $x \in R_i$ and $x' \in R_j$ in G_{co}, but there is no edge between R_i and R_j in G_T, then (i) either there is no path connecting R_i to R_j in G_T (so all tuples in R_i pair with all tuples in R_j), or, (ii) x and x' are connected in G_{co} via a set of tuples $x_1,\cdots,x_\ell$, such that the tables containing these tuples are connected by a path in G_T. Our algorithm in Section 4 shows that all such edges (x,x') can be safely deleted from G_{co} for the evaluation of the query that yielded G_T.

DEFINITION 1. *The* **co-table graph** *G_C is the subgraph of G_{co} with $V(G_C) = V(G_{co})$ and such that given two tuples $x \in R_i$ and $x' \in R_j$ there is an edge $(x,x') \in E(G_C)$ iff $(x,x') \in E(G_{co})$ and R_i and R_j are adjacent in G_T.*

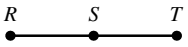

Figure 5: G_T for the relations in Example 1.

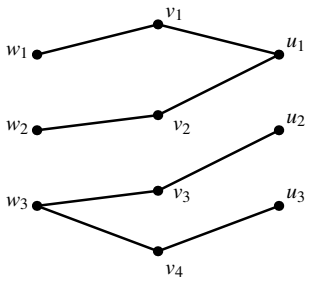

Figure 6: G_C for f in Example 1.

The co-table graph G_C generated by the event tables and query in Fig. 1 is shown in Fig. 6 (it is *not* a cograph!).

Co-occurrence graph vs. co-table graph The advantage of using the co-table graph instead of the co-occurrence graph is most dramatic in the following example:

EXAMPLE 3. *Consider $Q() : -R_1(\mathbf{x}_1),R_2(\mathbf{x}_2)$ where $\mathbf{x}_1$ and $\mathbf{x}_2$ have no common FO variable. Assuming that each of the tables R_1 and R_2 has n tuples, G_{co} has n^2 edges while G_C has none. A cograph-help algorithm must spend $\Omega(n^2)$ time even if it only reads G_{co}.*

On the other hand, G_C can be as big as G_{co}. In fact, when G_T is a complete graph (see next example), $G_C = G_{co}$.

EXAMPLE 4. *Consider $Q() : -R_1(\mathbf{x}_1,y),\ldots,R_k(\mathbf{x}_k,y)$ where $\mathbf{x}_i$ and $\mathbf{x}_j$ have no common FO variable if $i \neq j$. Here G_T is the complete graph on $R_1,\ldots,R_k$ and $G_C = G_{co}$.*

However, it can be verified that both our algorithm and the cograph-help algorithm have the same time complexity on the above example.

3. COMPUTING THE CO-TABLE GRAPH

In this section we show that given as input the provenance DAG H of a boolean conjunctive query *plan* without self-joins Q on a table-independent database representation I, the co-table graph G_C and the co-occurrence graph G_{co} of the boolean formula $Q(I)$ (see definitions in section 2) can be computed in poly-time in the sizes of H, I and Q.

It turns out that G_C and G_{co} are computed by similar algorithms, one being a minor modification of the other. As discussed in section 1, the co-occurrence graph G_{co} can then be used in conjunction with cograph recognition algorithms (eg., [6, 19, 3]), to find the read-once form of $Q(I)$ if it exists. On the other hand, the smaller co-table graph G_C is used by our algorithm described in section 4 for the same purpose.

We use $\texttt{Var}(f)$ to denote the sets of variables in a monotone boolean expression f. Recall that the provenance DAG H is a layered graph where every layer corresponds to a select, project or join operation in the query plan. We define the *width* of H as the maximum number of nodes at any layer of the DAG H and denote it by β_H. The main result in this section is summarized by the following theorem.

THEOREM 1. *Let $f = Q(I)$ be the boolean expression computed by the query plan Q on the table representation I (f can also be read off the provenance graph of Q on I, H), $n = |\mathtt{Var}(f)|$ be the number of variables in f, $m_H = |E(H)|$ be the number of edges of H, β_H be the width of H, and $m_{co} = |E(G_{co})|$ be the number of edges of G_{co}, the co-occurrence graph of f.*

1. *G_{co} can be computed in time $O(nm_H + \beta_H m_{co})$.*
2. *Further, the co-table graph G_C of f can be computed in time $O(nm_H + \beta_H m_{co} + k^2\alpha\log\alpha)$ where k is the number of tables in Q, and α is the maximum arity (width) of the tables in Q.*

3.1 LCA-Based Characterization of the Co-Occurrence Graph

Here we give a characterization of the presence of an edge (x,y) in G_{co} based on the *least common ancestors* of x and y in the graph H.

Again, let $f = Q(I)$ be the boolean expression computed by the query plan Q on the table representation I. As explained in section 2 f can also be read off the provenance graph H of Q and I since H is the representation of f without duplication of common subexpressions.

The absence of self-joins in Q implies the following.

LEMMA 1. *The DNF generated by expanding f (or H) using only the distributivity rule is in fact the IDNF of f up to idempotency (i.e. repetition of the same prime implicant is allowed).*

PROOF. Let g be the DNF generated from f by applying distributivity repeatedly. Due to the absence of self-joins g every implicant in g will have exactly one tuple from every table. Therefore, for any two implicants in g the set of variables in one is not a strict subset of the set of variables in the other and further absorption (eg., $xy + xyz = xy$) does not apply. (At worst, two implicants can be the same and the idempotence rule reduces one.) Therefore, g is also irredundant and hence *the* IDNF of f (up to commutativity and associativity). □

Denote by f_{IDNF} the IDNF of f, which, as we have seen, can be computed from f just by applying distributivity.

As with any DAG, we can talk about the nodes of H in terms of successors, predecessors, ancestors, and descendants, and finally about the *least common ancestors* of two nodes, denoted $\mathtt{lca}(x,y)$. Because H has a root $\mathtt{lca}(x,y)$ is never empty. When H is a tree, $\mathtt{lca}(x,y)$ consists of a single node. For a node $u \in V(H)$, we denote the set of leaf variables which are descendants of u by $\mathtt{Var}(u)$ (overloaded notation warning!); in other words, a variable x belongs to $\mathtt{Var}(u)$, $u \in V(H)$, if and only if x is reachable from u in H. Now we prove the key lemma of this section:

LEMMA 2. *Two variables $x,y \in \mathtt{Var}(f)$ belong together to a (prime) implicant of f_{IDNF} if and only if the set $\mathtt{lca}(x,y)$ contains a $\cdot$-node.*

PROOF. (if) Suppose $\mathtt{lca}(x,y)$ contains a $\cdot$-node u, i.e., x,y are both descendants of two distinct successors v_1, v_2 of u. Since the $\cdot$ operation multiplies all variables in $\mathtt{Var}(v_1)$ with all variables in $\mathtt{Var}(v_2)$, x and y will appear together in some implicant in f_{IDNF} which will not be absorbed by other implicants by Lemma 1.

(only if) Suppose that x,y appear together in an implicant of f_{IDNF} and $\mathtt{lca}(x,y)$ contains no $\cdot$-node. Then no $\cdot$-node in $V(H)$ has x,y in $\mathtt{Var}(v_1), \mathtt{Var}(v_2)$, where v_1, v_2 are its two distinct successors (note that any $\cdot$-node in a provenance DAG H can have exactly two successors). This implies that x and y can never be multiplied, contradiction. □

Since there are exactly k tables in the query plan, every implicant in f_{IDNF} will be of size k. Therefore:

LEMMA 3. *For every variable $x \in \mathtt{Var}(f)$ and $\cdot$-node $u \in V(H)$, if $x \in \mathtt{Var}(u)$, then $x \in \mathtt{Var}(v)$ for exactly one successor v of u.*

PROOF. If $x \in \mathtt{Var}(f)$ belongs to $\mathtt{Var}(v_1), \mathtt{Var}(v_2)$ for two distinct successors v_1, v_2 of u, then some implicant in f_{IDNF} will have $< k$ variables since $x \cdot x = x$ by idempotence. □

The statement of Lemma 2 provides a criterion for computing G_{co} using the computation of least common ancestors in the provenance graph, which is in often more efficient than computing the entire IDNF. We have shown that this criterion is satisfied in the case of conjunctive queries without self-joins. But it may also be satisfied by other kinds of queries, which opens a path to identifying other cases in which such an approach would work.

3.2 Computing the Table-Adjacency Graph

It is easier to describe the computation of G_T if we use the query in rule form $Q() : -R_1(\mathbf{x}_1), \ldots, R_k(\mathbf{x}_k)$.

The rule form can be computed in linear time from the SPJ query plan. Now the vertex set $V(G_T)$ is the set of table names $R_1, \cdots, R_k$. and an edge exists between R_i, R_j iff $\mathbf{x}_i$ and $\mathbf{x}_j$ have at least one FO variable in common i.e., R_i and R_j are joined. Whether or not such an edge should be added can be decided in time $O(\alpha\log\alpha)$ by sorting and intersecting $\mathbf{x}_i$ and $\mathbf{x}_j$. Here α is the maximum arity (width) of the tables $R_1, \cdots, R_k$. Hence G_T can be computed in time $O(k^2\alpha\log\alpha)$.

3.3 Computing the Co-Table Graph

Recall that co-table graph G_C is a subgraph of the co-occurrence graph G_{co} where we add an edge between two variables x,y, only if the tables containing these two tuples are adjacent in the table-adjacency graph G_T. Algorithm 1 COMPCOTABLE constructs the co-table graph G_C by a single *bottom-up* pass over the graph H.

Algorithm 1 *Algorithm* COMPCOTABLE

Input: Query plan DAG H and table-adjacency graph G_T
Output: Co-table graph G_C.

1: – Initialize $V(G_C) = \mathtt{Var}(f)$, $E(G_C) = \phi$.
2: – For all variables $x \in \mathtt{Var}(f)$, set $\mathtt{Var}(x) = \{x\}$.
3: – Do a *topological sort* on H and *reverse* the sorted order.
4: **for** every node $u \in V(H)$ in this order **do**
5: /* *Update $\mathtt{Var}(u)$ set for both $+$-node and $\cdot$-node u*/*
6: – Set $\mathtt{Var}(u) = \bigcup_v \mathtt{Var}(v)$, where the union is over all successors v of u.
7: **if** $u \in V(H)$ is a $\cdot$-node **then**
8: /* *Add edges to G_C only for a $\cdot$-node*/*
9: – Let v_1, v_2 be its two successors.
10: **for** every two variables $x \in \mathtt{Var}(v_1)$ and $y \in \mathtt{Var}(v_2)$ **do**
11: **if** (i) the tables containing x,y are adjacent in G_T and (ii) the edge (x,y) does not exist in $E(G_C)$ yet **then**
12: – Add an edge between x and y in $E(G_C)$.
13: **end if**
14: **end for**
15: **end if**
16: **end for**

It is easy to see that a minor modification of the same algorithm can be used to compute the co-occurrence graph G_{co}: in Step 11 we simply skip the check whether the tables containing the two tuples are adjacent in G_T. Since this is the only place where G_T is

used, the time for the computation of G_C does not include the time related to computing/checking G_T.

Correctness. By a simple induction, it can be shown that the set $\texttt{Var}(u)$ is correctly computed at every step, i.e., it contains the set of all nodes which are reachable from u in H (since the nodes are processed in reverse topological order and $\texttt{Var}(u)$ is union of $\texttt{Var}(v)$ for over all successors v of u). Next lemma shows that algorithm CompCoTable correctly builds the co-table graph G_C (proof in [33]).

Lemma 4. *Algorithm* CompCoTable *adds an edge (x,y) to G_C if and only if x,y together appear in some implicant in f_{IDNF} and the tables containing x,y are adjacent in G_T.*

Time Complexity. Here we give a sketch of the time complexity analysis, details can be found in the full version [33]. Computation of the table adjacency graph takes $O(k^2\alpha\log\alpha)$ time as shown in Section 3.2. The total time complexity of algorithm CompCoTable as given in Theorem 1 is mainly due to two operations: (i) computation of the $\texttt{Var}(u)$ set at every internal node $u \in V(H)$, and (ii) to perform the test for pairs x,y at two distinct children of a $\cdot$-node, whether the edge (x,y) already exists in G_C, and if not, to add the edge.

We show that the total time needed for the first operation is $O(nm_H)$ in total: for every internal node $u \in V(H)$ we can scan the variables sets of all its immediate successor in $O(nd_u)$ time to compute $\texttt{Var}(u)$, where d_u is the outdegree of node u in H. This gives total $O(nm_H)$ time. On the other hand, for adding edges (x,y) in G_C, it takes total $O(m_{co}\beta_H)$ time during the execution of the algorithm, where m_{co} is the number of edges in the co-occurrence graph (and not in the co-table graph, even if we compute the co-table graph G_C) and β_H is the width of the graph H. To show this, we show that two variables x,y are considered by the algorithm at Step 10 if and only if the edge (x,y) already exists in the co-occurrence graph G_{co}, however, the edge may not be added to the co-table graph G_C if the corresponding tables are not adjacent in the table adjacency graph G_T. We also show that any such edge (x,y) will be considered at a unique level of the DAG H. In addition to these operations, the algorithm does initialization and a topological sort on the vertices which take $O(m_H+n_H)$ time ($n_H = |V(H)|$) and are dominated by the these two operations.

4. COMPUTING THE READ-ONCE FORM

Our algorithm CompRO (for *Compute Read-Once*) takes an instance I of the schema $\mathbf{R} = R_1,\cdots,R_k$, a query $Q() :- R_1(\mathbf{x_1}), R_2(\mathbf{x_2}), \cdots, R_k(\mathbf{x_k})$ along with the table adjacency graph G_T and co-table graph G_C computed in the previous section as input, and outputs whether $Q(I)$ is read-once. (if so it computes its unique read-once form).

Theorem 2. *Suppose we are given a query Q, a table-independent database representation I, the co-table graph G_C and the table-adjacency graph G_T for Q on I as inputs. Then*

1. *Algorithm* CompRO *decides correctly whether the expression $Q(I)$ generated by evaluating Q on I is read-once, and if yes, it returns the unique read-once form of the expression, and,*
2. *Algorithm* CompRO *runs in time*

$$O(m_T\alpha\log\alpha + (m_C+n)\min(k,\sqrt{n})),$$

where $m_T = |E(G_T)|$ is the number of edges in G_T, $m_C = |E(G_C)|$ is the number of edges in G_C, n is the total number of tuples in I, k is the number of tables, and α is the maximum size of any subgoal.

4.1 Algorithm CompRO

In addition to the probabilistic database with tables $R_1,\cdots,R_k$ and input query Q, our algorithm also takes the *table-adjacency graph* G_T and the *co-table graph* G_C computed in the first phase as discussed in Section 3. The co-table graph G_C also helps us to remove *unused tuples* from all the tables which do not appear in the final expression – every unused tuple won't have a corresponding node in G_C. So from now on we can assume wlog. that every tuple in every table appears in the expression $Q(I)$.

The algorithm CompRO uses two decomposition operations: *Row decomposition* is a *horizontal* decomposition operation which partitions the rows or tuples in every table into the same number of groups and forms a set of sub-tables from every table. On the other hand, *Table decomposition* is a *vertical* decomposition operation. It partitions the set of tables into groups and a *modified sub-query* is evaluated in every group. For convenience, we will represent the instance I as $R_1[T_1],\cdots,R_k[T_k]$, where T_i is the set of tuples in table R_i. Similarly, for a subset of tuples $T_i' \subseteq T_i$, $R_i[T_i']$ will denote the instance of relation R_i containing exactly the tuples in T_i'. The algorithm CompRO is given in Algorithm 2.

Row Decomposition. The row decomposition operation partitions the tuples variables in *every* table into ℓ disjoint groups. In addition, it decomposes the co-table graph G_C into $\ell \geq 2$ disjoint *induced subgraphs*[9] corresponding to the above groups. For every pair of distinct groups j, j', and for every pair of distinct tables $R_i, R_{i'}$, no tuple in group j of R_i ever joins with a tuple in group j' of $R_{i'}$ (recall that the query does not have any self-join operation). The procedure for row decomposition is given in Algorithm 3[10].

Algorithm 3 $\text{RD}(\langle R_1[T_1],\cdots,R_k[T_k]\rangle, G_C')$

Input: Tables $R_1[T_1],\cdots,R_k[T_k]$, and induced subgraph G_C' of G_C on $\bigcup_{i=1}^k T_i$
Output: If successful, the partition of G_C' and tuple variables of every input tables into $\ell \geq 2$ connected components: $\langle\langle T_{1,j},\cdots,T_{k,j}\rangle, G_{C,j}'\rangle$, $j \in [1,\ell]$

1: – Run BFS or DFS to find the connected components in G_C'.
2: – Let ℓ be the number of connected components.
3: **if** $\ell = 1$ **then** {/* *there is only one connected component* */}
4: **return** with failure: "Row decomposition is not possible".
5: **else**
6: – Let the tuples (vertices) of table R_i in the j-th connected component j of G_C' be $T_{i,j}$
7: – Let the induced subgraph for connected component j be $G_{C,j}$.
8: **return** $\langle\langle T_{1,1},\cdots,T_{k,1}\rangle, G_{C,1}'\rangle$, $\cdots$, $\langle\langle T_{1,\ell},\cdots,T_{k,\ell}\rangle, G_{C,\ell}'\rangle$ with success.
9: **end if**

[9] A subgraph H of G is an induced subgraph, if for any two vertices $u,v \in V(H)$, if $(u,v) \in E(G)$, then $(u,v) \in E(H)$.

[10] It should be noted that the row decomposition procedure may be called on $R_{i_1}[T_{i_1}'],\cdots,R_{i_p}[T_{i_p}']$ and G_C', where $R_{i_1},\cdots,R_{i_p}$ is a subset of the relations from $R_1,\cdots,R_k$, $T_{i_1}',\cdots,T_{i_p}'$ are subsets of the respective set of tuples $T_{i_1},\cdots,T_{i_p}$, and G_C' is the induced subgraph of G_C on $T_{i_1}',\cdots,T_{i_p}'$. For simplicity in notations, we use $R_1[T_1],\cdots,R_k[T_k]$. This holds for table decomposition as well.

Algorithm 2 COMPRO*(Q, $I = \langle R_1[T_1], \cdots, R_k[T_k]\rangle$, G_C, G_T,* FLAG*)*

Input: Query Q, tables $R_1[T_1], \cdots, R_k[T_k]$, co-table graph G_C, table-adjacency graph G_T, and a boolean parameter FLAG which is true if and only if row decomposition is performed at the current step.
Output: If successful, the unique read-once form f^* of the expression for $Q(I)$

if $k = 1$ **then**
  return $\sum_{x \in T_1} x$ with success. (*/* all unused tuples are already removed */*)
end if
if FLAG = TRUE **then** {*/* Row decomposition */*}
  – Perform RD($\langle R_1[T_1], \cdots, R_k[T_k]\rangle, G_C$).
  if row decomposition returns with success **then** {*/** RD *partitions every table and G_C into $\ell \geq 2$ disjoint groups*/*}
    – Let the groups returned be $\langle \langle T_1^j, \cdots, T_k^j \rangle, G_{C,j} \rangle$, $j \in [1, \ell]$.
    – $\forall j \in [1, \ell]$, let $f_j =$ COMPRO(Q, $\langle R_1[T_1^j], \cdots, R_k[T_k^j]\rangle$, $G_{C,j}, G_T,$ FALSE).
    return $f^* = f_1 + \cdots + f_\ell$ with success.
  end if
else {*/* Table decomposition */*}
  – Perform TD($\langle R_1[T_1], \cdots, R_k[T_k]\rangle$, Q, G_T, G_C).
  if table decomposition returns with success **then** {*/** TD *partitions I, G_C and G_T into $\ell \geq 2$ disjoint groups, $\sum_{j=1}^{\ell} k_j = k$ */*}
    Let the groups returned be $\langle \langle R_{j,1}, \cdots, R_{j,k_j} \rangle, \widehat{Q_j}, G_{C,j}, G_{T,j} \rangle$, $j \in [1, \ell]$.
    – $\forall j \in [1, \ell]$, $f_j =$ COMPRO($\widehat{Q_j}$, $\langle R_1[T_1], \cdots, R_k[T_k]\rangle$, $G_{C,j}, G_{T,j},$ TRUE).
    return $f^* = f_1 \cdot \ldots \cdot f_\ell$ with success.
  end if
end if
if the current operation is not successful **then** {*/* Current row or table decomposition is not successful and $k > 1$ */*}
  return with failure: "$Q(I)$ is not read-once".
end if

Table Decomposition. On the other hand, the table decomposition operation partitions the set of tables $\mathbf{R} = R_1, \cdots, R_k$ into $\ell \geq 2$ disjoint groups $\mathbf{R_1}, \cdots, \mathbf{R}_\ell$. It also decomposes the table-adjacency graph G_T and co-table graph G_C into ℓ disjoint induced subgraphs $G_{T,1}, \cdots, G_{T,\ell}$, and, $G_{C,1}, \cdots, G_{C,\ell}$ respectively corresponding to the above groups. The groups are selected in such a way that all tuples in the tables in one group join with all tuples in the tables in another group. This procedure also modifies the sub-query to be evaluated on every group by making the subqueries of different groups mutually independent by introducing free variables, i.e., they do not share any common variables after a successful table decomposition. Algorithm 4 describes the table decomposition operation. Since the table decomposition procedure changes the input query Q to $\widehat{Q} = \widehat{Q_1}, \cdots, \widehat{Q_\ell}$, it is crucial to ensure that changing the query to be evaluated does not change the answer to the final expression. This is shown in the full version [33].

The following lemma shows that if row-decomposition is successful, then table decomposition cannot be successful and vice versa. However, both of them may be unsuccessful in case the final expression is not read-once (proof in [33]).

LEMMA 5. *At any step of the recursion, if row decomposition is successful then table decomposition is unsuccessful and vice versa.*

Therefore, in the top-most level of the recursive procedure, we can verify which operation can be performed – if both of them fail, then the final expression is not read-once which follows from the correctness of our algorithm. If the top-most recursive call performs a successful row decomposition initially the algorithm COMPRO is called as COMPRO(Q, $\langle R_1[T_1], \cdots, R_k[T_k]\rangle$, G_C, G_T, TRUE). The last boolean argument is TRUE if and only if row decomposition is performed at the current level of the recursion tree. If in the first step table decomposition is successful, then the value of the last boolean variable in the initial call will be FALSE.

Correctness. The following two lemmas respectively show the soundness and completeness of the algorithm COMPRO (proofs are in [33]).

LEMMA 6. ***(Soundness)*** *If the algorithm returns with success, then the expression f^* returned by the algorithm* COMPRO *is equivalent to the expression $Q(I)$ generated by evaluation of query Q on instance I. Further, the output expression f^* is in read-once form.*

LEMMA 7. ***(Completeness)*** *If the expression $Q(I)$ is read-once, then the algorithm* COMPRO *returns the unique read-once form f^* of the expression.*

For completeness, it suffices to show that if $Q(I)$ is read-once, then the algorithm does not exit with error. Indeed, if the algorithm returns with success, as showed in the soundness lemma, the algorithm returns an expression f^* in read-once form which is the unique read-once form of $Q(I)$ [12, 5].

Time Complexity. Consider the recursion tree of the algorithm COMPRO. Lemma 5 shows that at any level of the recursion tree, either all recursive calls use the row decomposition procedure, or all recursive calls use the column decomposition procedure. The time complexity of COMPRO given in Theorem 2 is analyzed in the following steps. If $n' =$ the total number of input tuples at the current recursive call and $m'_C =$ the number of edges in the induced subgraph of G'_C on these n' vertices, we show that row decomposition takes $O(m'_C + n')$ time and, *not considering the time needed to compute the modified queries $\widehat{Q_j}$* (Step 13 in Algorithm 4), the table decompositions procedure takes $O(m'_C + n')$ time. Then we consider the time needed to compute the modified queries and show that these steps over *all* recursive calls of the algorithm take $O(m_T \alpha \log \alpha)$ time in total, where α is the maximum size of a subgoal in the query Q. Finally, we give a bound of $O(\min(k, \sqrt{n}))$ on the height of the recursive tree for the algorithm COMPRO. However, note that at every step, for row or table decomposition, every tuple in G'_C goes

Algorithm 4 TD($\langle R_1[T_1],\cdots,R_k[T_k]\rangle$, $\langle Q() : -R_1(\mathbf{x_1}),\cdots,R_k(\mathbf{x_k})\rangle$, G'_C, G'_T)

Input: Tables $R_1[T_1],\cdots,R_k[T_k]$ **query** $Q() : -R_1(\mathbf{x_1}),\cdots,R_k(\mathbf{x_k})$ **induced subgraph** G'_T **of** G_T **on** $\bigcup_{i=1}^{k} R_i$**, induced subgraph** G'_C **of** G_C **on** $\bigcup_{i=1}^{k} T_i$

Output: If successful, a partition of input tables, G'_T**,** G'_C **into** ℓ **groups, and an updated sub-query for every group**

for all edges $e = (R_i, R_j)$ in G'_T **do**
  – Annotate the edge e with common variables C_e in the vectors $\mathbf{x_i}$, $\mathbf{x_j}$.
  – Mark the edge e with a "+" if for every pair of tuple variables $x \in T_i$ and $y \in T_j$, the edge (x,y) exists in G'_C. Otherwise mark the edge with a "−".
end for
– Run BFS or DFS to find the connected components in G_T w.r.t "−" edges
– Let ℓ be the number of connected components.
if $\ell = 1$ **then** {*/* there is only one connected component */*}
  return with "Failure: Table decomposition is not possible".
else
  – Let $G'_{T,1},\cdots,G'_{T,\ell}$ be the induced subgraphs of ℓ connected components of G'_T and $G'_{C,1},\cdots,G'_{C,\ell}$ be the corresponding induced subgraph for G'_C.
  – Let $\mathbf{R_p} = \langle R_{p,1},\cdots,R_{p,k_p}\rangle$ be the subset of tables in the p-th component of G'_T, $p \in [1,\ell]$.
  / Compute a new query for every component */*
  for every component p **do**
    for every table R_i in this component p **do**
      – Let $C_i = \bigcup_e C_e$ be the union of common variables C_e over all edges e from R_i to tables in different components of $G_{T'}$ (all such edges are marked with '+')
      – For every common variable $z \in C_i$, generate a *new (free)* variable z^i, and replace *all* occurrences of z in vector $\mathbf{x_i}$ by z'. Let $\widehat{\mathbf{x_i}}$ be the new vector.
      – Change the query subgoal for R_i from $R_i(\mathbf{x_i})$ to $R_i(\widehat{\mathbf{x_i}})$.
    end for
    Let $\widehat{Q_p}() : -R_{p,1}(\widehat{\mathbf{x_{p,1}}}),\cdots,R_{p,k_p}(\widehat{\mathbf{x_{p,k_p}}})$ be the new query for component p.
  end for
  return $\langle\langle R_{1,1},\cdots,R_{1,k_1}\rangle,\widehat{Q_1},G_{C,1},G_{T,1}\rangle,\cdots,\langle\langle R_{\ell,1},\cdots,R_{\ell,k_\ell}\rangle,\widehat{Q_\ell},G_{C,\ell},G_{T,\ell}\rangle$ with success.
end if

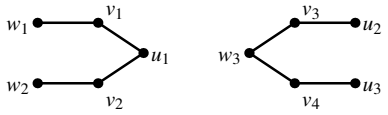

Figure 7: $G_{C,1}$ and $G_{C,2}$.

to exactly one of the recursive calls, and every edge in G'_C goes to at most one of the recursive calls. So for both row and table decomposition at every level of the recursion tree the total time is $O(m_C + n)$. Combining all these observations, the total time complexity of the algorithm is $O(m_T \alpha \log \alpha + (m_C + n)\min(k, \sqrt{n}))$ as stated in Theorem 2. The details can be found in the full version [33].

Example. Here we illustrate our algorithm. Consider the query Q and instance I from Example 1 in the introduction. The input query is $Q() : -R(x)S(x,y)T(y)$. In the first phase, the table-adjacency graph G_T and the co-table graph G_C are computed. These graphs are depicted in Fig. 5 and Fig. 6 respectively.

Now we apply COMPRO. There is a successful row decomposition at the top-most recursive call that decomposes G_C into the two subgraphs $G_{C,1}, G_{C,2}$ shown in Fig. 7. So the final expression f^* annotating the answer $Q(I)$ will be the *sum* of the expressions f_1, f_2 annotating the answers of Q applied to the relations corresponding to $G_{C,1}$ and $G_{C,2}$ respectively.

The relations corresponding to $G_{C,1}$ are

$$R = \begin{array}{|c|}\hline a_1 \\ \hline b_1 \\ \hline \end{array}\ \begin{array}{l} w_1 \\ w_2 \end{array} \qquad S = \begin{array}{|cc|}\hline a_1 & c_1 \\ \hline b_1 & c_1 \\ \hline \end{array}\ \begin{array}{l} v_1 \\ v_2 \end{array} \qquad T = \begin{array}{|c|}\hline c_1 \\ \hline \end{array}\ u_1$$

and, the relations corresponding to $G_{C,2}$ are

$$R = \begin{array}{|c|}\hline a_2 \\ \hline \end{array}\ w_3 \qquad S = \begin{array}{|cc|}\hline a_2 & c_2 \\ \hline a_2 & d_2 \\ \hline \end{array}\ \begin{array}{l} v_3 \\ v_4 \end{array} \qquad T = \begin{array}{|c|}\hline c_2 \\ \hline d_2 \\ \hline \end{array}\ \begin{array}{l} u_2 \\ u_3 \end{array}$$

Now we focus on the first recursive call at the second level of recursion tree with input co-table subgraph $G_{C,1}$. Note that the table-adjacency graph for this call is the same as G_T. At this level the table decomposition procedure is invoked and the edges of the table-adjacency graph are marked with $+$ and $-$ signs, see Fig. 8. In this figure the common variable set for R,S on the edge (R,S) is $\{x\}$, and for S,T on the edge (S,T) is $\{y\}$. Further, the edge (S,T) is marked with a "+" because there are all possible edges between the tuples in S (in this case tuples v_1, v_2) and the tuples in T (in this case u_1). However, tuples in R (here w_1, w_2) and tuples in S (here v_1, v_2) do not have all possible edges between them so the edge (R,S) is marked with a "−".

Table decomposition procedure performs a connected component decomposition using "−"-edges, that decomposes G_T in two components $\{R,S\}$ and $\{T\}$. The subset C of common variables collected from the "+"-edges across different components will be the variables on the single edge (S,T), $C = \{y\}$. This variable y is replaced by new *free variables* in all subgoals containing it, which are S and T in our case. So the modified queries for disjoint components returned by the table decomposition procedure are $\widehat{Q_1}() : -R(x)S(x,y_1)$ and $\widehat{Q_2}() : -T(y_2)$. The input graph $G_{C,1}$ is decomposed further into $G_{C,1,1}$ and $G_{C,1,2}$, where $G_{C,1,1}$ will have the edges (w_1, v_1) and (w_2, v_2), whereas $G_{C,1,2}$ will have no edges and a single vertex u_1. Moreover, the expression f_1 is the product of f_{11} and f_{12} generated by these two queries respectively. Since the

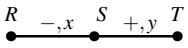

Figure 8: Marked table-adjacency graph for R, S, T.

number of tables for $\widehat{Q_2}$ is only one, and T has a single tuple, by the base step (Step 2) of COMPRO, $f_{12} = u_1$. For expression f_{11} from $\widehat{Q_1}$, now the graph $G_{C,1,1}$ can be decomposed using a row decomposition to two connected components with single edges each ((w_1, v_1) and (w_2, v_2) respectively). There will be recursive subcalls on these two components and each one of them will perform a table decomposition (one tuple in every table, so the single edges in both calls will be marked with "+"). Hence f_{11} will be evaluated to $f_{11} = w_1v_1 + w_2v_2$. So $f_1 = f_{11} \cdot f_{12} = (w_1v_1 + w_2v_2)u_1$.

By a similar analysis as above, it can be shown that the same query Q evaluated on the tables R, S, T given in the above tables give $f_2 = w_3(v_3v_4 + u_2u_3)$. So the overall algorithm is successful and outputs the read-once form $f^* = f_1 + f_2 = (w_1v_1 + w_2v_2)u_1 + w_3(v_3u_2 + v_4u_3)$.

5. DISCUSSION OF TIME COMPLEXITY OF QUERY-ANSWERING ALGORITHM

Putting together our results from Sections 3 and 4, we propose the following algorithm for answering boolean conjunctive queries without self-joins on tuple-independent probabilistic databases.

Phase 0 (Compute provenance DAG)
Input: query Q, event table rep I
Output: provenance DAG H
Complexity: $O((\frac{ne}{k})^k)$

Phase 1 (Compute co-table graph)
Input: H, Q
Output: table-adjacency graph G_T, co-table graph G_C
Complexity: $O(nm_H + \beta_H m_{co} + k^2 \alpha \log \alpha)$ (Thm. 1)

Phase 2 (Compute read-once form)
Input: event table rep I, Q, G_T, G_C
Output: read-once form f^* or FAIL
Complexity: $O(m_T \alpha \log \alpha + (m_C + n)\min(k, \sqrt{n}))$ (Thm. 2)

Size of the provenance DAG H. Let f be the boolean event expression generated by some query plan for Q on the database I. The number of edges m_H in the DAG H represents the size of the expression f. Since there are exactly k subgoals in the input query Q, one for every table, every prime implicant of f_{IDNF} will have exactly k variables, so the size of f_{IDNF} is at most $\binom{n}{k} \leq (\frac{ne}{k})^k$. Further, the size of the input expression f is maximum when f is already in IDNF. So size of the DAG H is upper bounded by $m_H \leq (\frac{ne}{k})^k$. Again, the "leaves" of the DAG H are exactly the n variables in f. So $m_H \geq n - 1$, where the lower bound is achieved when the DAG H is a tree (every node in H has a unique predecessor); in that case f must be read-once and H is the unique read-once tree of f.

Therefore $n - 1 \leq m_H \leq (\frac{ne}{k})^k$. Although the upper bound is quite high, it is our contention that for practical query plans the size of the provenance graph is much smaller than the size of the corresponding IDNF.

Data complexity. The complexity dichotomy of [8] is for data complexity, i.e., the size of the query is bounded by a constant. This means that our k and α are $O(1)$. Hence the time complexities of the Phase 1 and Phase 2 are $O(nm_H + \beta_H m_{co})$ and $O(m_C + n)\min(k, \sqrt{n}))$ respectively. As discussed above, m_H is $\Omega(n)$ and $O((\frac{ne}{k})^k)$. So one of these two terms may dominate the other based on the relative values of m_H, m_C and m_{co} and of β_H. For example, when $m_H = \theta(n)$, $m_{co} = \theta(m_C) = \theta(n^2)$, and $\beta_H = O(1)$, the first phase takes $O(n^2)$ time, whereas the second phase may take $O(n^{\frac{5}{2}})$ time. However, when $m_H = \Omega(n^{\frac{3}{2}})$, the first phase always dominates.

In any case, we take the same position as [34] that for unsafe [8] queries the competition comes from the approach that does *not* try to detect whether the formulas are read-once and instead uses probabilistic inference [26] which is in general EXPTIME. In contrast, our algorithm runs in PTIME, and works for a larger class of queries than the safe queries [8] (but of course, not on all instances).

Comparisons with other algorithms. For these comparisons we do not restrict ourselves to data complexity, instead taking the various parameters of the problem into consideration.

First consider the **general read-once detection algorithm**. This consists of choosing some plan for the query, computing the answer boolean event expression f, computing its IDNF, and then using the (so far, best) algorithm [12] to check if f is read-once and if so to compute its read-once form. The problem with this approach is that the read-once check is indeed done in time a low polynomial, but *in the size of f_{IDNF}*. For example, consider a boolean query like the one in Example 3. This is a query that admits a plan (the safe plan!) that would generate the event expression $(x_1 + y_1) \cdots (x_n + y_n)$ on an instance in which each R_i has two tuples x_i and y_i. This is a read-once expression easily detected by our algorithm, which avoids the computation of the IDNF.

Next consider the **cograph-help algorithm** that we have already mentioned and justified in Section 2. This consists of our Phase 0 and a slightly modified Phase 1 that computes the co-occurrence graph G_{co}, followed by checking if G_{co} is a cograph using one of the linear-time algorithms given in [6, 19, 3] which also outputs the read-once form if possible. Since Phase 0 and Phase 1 are common we only need to compare the last phases.

The co-graph recognition algorithms will all run in time $O(m_{co} + n)$. Our Phase 2 complexity is better than this when $m_C \min(k, \sqrt{n}) = o(m_{co})$. Although in the worst case this algorithm performs at least as well as our algorithm (since m_C may be $\theta(m_{co})$), (i) almost always the time required in first phases will dominate, so the asymptotic running time of both these algorithms will be comparable, (ii) as we have shown earlier, the ratio $\frac{m_{co}}{m_C}$ can be as large as $\Omega(n^2)$, and the benefit of this could be significantly exploited by caching co-table graphs computed for other queries (see discussions in Section 7), and (iii) these linear time algorithms use complicated data structures, whereas we use simple graphs given as adjacency lists and connectivity-based algorithms, so our algorithms are simpler to implement and may run faster in practice.

Finally we compare our algorithm against that given in [34]. Let us call it the **lineage-tree algorithm** since they take the lineage tree of the result as input as opposed to the provenance DAG as we do. Although [34] does not give a complete running time analysis of the lineage tree algorithm, for the brief discussion we have, we can make, to the best of our understanding, the following observations.

Every join node in the lineage tree has two children, and every project node can have arbitrary number of children. When the recursive algorithm computes the read-once trees of every child of a project node, every pair of such read-once trees are merged which may take $O(n^2k^2)$ time for every single merge (since the variables in the read-once trees to be merged are repeated). Without counting the time to construct the lineage tree this algorithm may take $O(Nn^2k^2)$ time in the worst case, where N is the number of nodes

in the lineage tree.

Since [34] does not discuss constructing the lineage tree we will also ignore our Phase 0. We are left with comparing N with m_H. It is easy to see that the number of edges in the provenance DAG H, $m_H = \theta(N)$, where N is the number of nodes in the lineage tree, when both originate from the same query plan[11] Since the lineage-tree algorithm takes $O(Nn^2k^2)$ time in the worst case, and we use $O(nm_H + \beta_H m_{co} + k^2\alpha\log\alpha) + O((m_C+n)\min(k,\sqrt{n})) = O(nN + \beta_H n^2 + k^2\alpha\log\alpha + n^{\frac{5}{2}})$. The width β_H of the DAG H in the worst case can be the number of nodes in H. So our algorithm *always* gives an $O(k^2)$ improvement in time complexity over the lineage-tree algorithm given in [34] whereas the benefit can often be more.

6. RELATED WORK

The beautiful complexity dichotomy result of [8] classifying conjunctive queries without self-joins on tuple-independent databases into "safe" and "unsafe" has spurred and intensified interest in probabilistic databases. Some papers have extended the class of safe relational queries [9, 29, 30, 7]. Others have addressed the question of efficient query answering for unsafe queries on *some* probabilistic databases. This includes mixing the intensional and extensional approaches, in effect finding subplans that yield read-once subexpressions in the event expressions [23]. The technique identifies "offending" tuples that violate functional dependencies on which finding safe plans relies and deals with them intensionally. It is not clear that this approach would find the read-once forms that our algorithm finds. The OBDD-based approach in [29] works also for some unsafe queries on some databases. The SPROUT secondary-storage operator [31] can handle efficiently some unsafe queries on databases satisfying certain functional dependencies.

Exactly like us, [34] looks to decide efficiently when the extensional approach is applicable given a conjunctive query without self-joins and a tuple-independent database. We have made comparisons between the two papers in various places, especially in Section 5. Here we only add that that our algorithm deals with different query plans uniformly, while the lineage tree algorithm needs to do more work for non-deep plans. The graph structures used in our approach bear some resemblance to the graph-based synopses for relational selectivity estimation in [35].

The read-once property has been studied for some time, albeit under various names [20, 17, 36, 21, 18, 24, 32]. It was shown [21] that if RP≠NP then read-once cannot be checked in PTIME for arbitrary monotone boolean formulas, but for formulas in IDNF (as input) read-once can be checked in PTIME [12]. Our result here sheds new light on another class of formulas for which such an efficient check can be done.

7. CONCLUSIONS AND FUTURE WORK

We have investigated the problem of efficiently deciding when a conjunctive query *without self-joins* applied to a *tuple-independent* probabilistic database representation yields result representations featuring *read-once* boolean event expressions (and, of course, efficiently computing their read-once forms when they exist). We have given a complete and simple to implement algorithm of low polynomial data complexity for this problem, and we have compared our results with those of other approaches.

As explained in the introduction, the results of this paper do not constitute complexity dichotomies. However, there is some hope that the novel proof of completeness that we give for our algorithm may be of help for complexity dichotomy results in the space coordinated by the type of queries and the type of databases we have studied.

Of independent interest may be that we have also implicitly performed a study of an interesting class of monotone boolean formulas, those that can be represented by the provenance graphs of conjunctive queries without self-joins (characterizations of this class of formulas that do not mention the query or the database can be easily given). We have shown that for this class of formulas the read-once property is decidable in low PTIME (the problem for arbitrary formulas is unlikely to be in PTIME, unless RP=NP). Along the way we have also given an efficient algorithm for computing the co-occurrence graph of such formulas (in all the other papers we have examined, computing the co-occurrence graph entails an excursion through computing a DNF; this, of course, may be the best one can do for arbitrary formulas, if RP≠NP). It is likely that nicely tractable class of boolean formulas may occur in other database applications, to be discovered.

For further work one obvious direction is to extend our study to larger classes of queries and probabilistic databases [9, 7]. Recall from the discussion in the introduction however, that the class of queries considered should not be able to generate arbitrary monotone boolean expressions. Thus, SPJU queries are too much (but it seems that our approach might be immediately useful in tackling unions of conjunctive queries without self-joins, provided the plans do the unions last).

On the more practical side, work needs to be done to apply our approach to non-boolean queries, i.e., they return actual tables. Essentially, one would work with the provenance graph associated with each table (initial, intermediate, and final) computing simultaneously the co-table graphs of the event expressions on the graph's roots. It is likely that these co-table graphs can be represented together, with ensuing economy.

However we believe that the most practical impact would have the *caching of co-table graphs* at the level of the system, over batches of queries on the same database, since the more expensive step in our algorithm is almost always the computation of the co-table graph (see discussion in Section 5).

This would work as follows, for a fixed database I. When a (let's say boolean for simplicity) conjunctive query Q_1 is processed, consider also the query $\bar{Q}_1$ which is obtained from Q_1 by replacing each *occurrence* of constants with a distinct fresh FO variable. Moreover if an FO variable x occurs several times in a subgoal $R_i(\mathbf{x}_i)$ of Q but does *not* occur in any of the other subgoals (i.e., x causes selections but not joins), replace also each occurrence of x with a distinct fresh FO variable. In other words, Q_1 is doing what $\bar{Q}_1$ is doing, but it first applies some selections on the various tables of I. We can say that $\bar{Q}_1$ is the "join pattern" behind Q_1. Next, compute the co-table graph for $\bar{Q}_1$ on I and *cache* it together with $\bar{Q}_1$. It is not hard to see that the co-table graph for Q_1 can be efficiently computed from that of $\bar{Q}_1$ by a "clean-up" of those parts related to tuples of I that do not satisfy the select conditions of Q_1.

When another query Q_2 is processed, check if its join-pattern $\bar{Q}_2$ *matches* any of the join-patterns previously cached (if not, we further cache *its* join-pattern and co-table graph). Let's say it matches $\bar{Q}_1$. Without defining precisely what "matches" means, its salient property is that the co-table graph of $\bar{Q}_2$ can be efficiently obtained from that of $\bar{Q}_2$ by another clean-up, just of edges, guided by the table-adjacency graph of $\bar{Q}_2$ (which is the same as that of Q_2). It can be shown that these clean-up phases add only an $O(n\alpha)$ to the running time.

There are two practical challenges in this approach. The first one

[11] If we "unfold" provenance DAG H to create the lineage tree, the tree will have exactly m_H edges, and the number of nodes in the tree will be $N = m_H + 1$.

is efficiently finding in the cache some join-pattern that matches that of an incoming query. Storing the join-patterns together into some clever data structure might help. The second one is storing largish numbers of cached co-table graphs. Here we observe that they can all be stored with the same set of nodes and each edge would have a list of the co-table graph it which it appears. Even these lists can be large, in fact the number of all possible joint-patterns is exponential in the size of the schema. More ideas are needed and ultimately the viability of this caching technique can only be determined experimentally.

Acknowledgement. We thank Amol Deshpande and Dan Suciu for useful discussions.

8. REFERENCES

[1] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, pages 983–992, 2008.

[2] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

[3] A. Bretscher, D. Corneil, M. Habib, and C. Paul. A simple linear time LexBFS cograph recognition algorithm. *SIAM J. Discrete Math.*, 22(4):1277–1296, 2008.

[4] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *VLDB*, pages 71–81, 1987.

[5] D. G. Corneil, H. Lerchs, and L. S. Burlingham. Complement reducible graphs. *Discrete Appl. Math.*, 3(3):163–174, 1981.

[6] D. G. Corneil, Y. Perl, and L. K. Stewart. A linear recognition algorithm for cographs. *SIAM J. Comput.*, 14(4):926–934, 1985.

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

[8] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

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

[10] N. N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *PODS*, pages 1–12, 2007.

[11] 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.

[12] M. C. Golumbic, A. Mintz, and U. Rotics. Factoring and recognition of read-once functions using cographs and normality and the readability of functions associated with partial k-trees. *Discrete Appl. Math.*, 154(10):1465–1477, 2006.

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

[14] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, pages 675–686, 2007.

[15] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[16] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *EDBT Workshops*, pages 278–296, 2006.

[17] V. A. Gurvich. Repetition-free Boolean functions. *Uspehi Mat. Nauk*, 32(1(193)):183–184, 1977.

[18] V. A. Gurvich. Criteria for repetition-freeness of functions in the algebra of logic. *Soviet Math. Dokl.*, 43(3):721–726, 1991.

[19] M. Habib and C. Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 145(2):183 – 197, 2005.

[20] J. P. Hayes. The fanout structure of switching functions. *J. Assoc. Comput. Mach.*, 22(4):551–571, 1975.

[21] L. Hellerstein and M. Karpinski. Learning read-once formulas using membership queries. In *COLT*, pages 146–161, 1989.

[22] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[23] A. Jha, D. Olteanu, and D. Suciu. Bridging the gap between intensional and extensional query evaluation in probabilistic databases. In *EDBT*, pages 323–334, 2010.

[24] M. Karchmer, N. Linial, I. Newman, M. E. Saks, and A. Wigderson. Combinatorial characterization of read-once formulae. *Discrete Mathematics*, 114(1-3):275–282, 1993.

[25] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.

[26] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA, 2009.

[27] L. V. S. Lakshmanan, N. Leone, R. B. Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3), 1997.

[28] I. Newman. On read-once boolean functions. In M. S. Paterson, editor, *Boolean Function Complexity*, pages 25–34. Cambridge University Press, 1992.

[29] D. Olteanu and J. Huang. Using obdds for efficient query evaluation on probabilistic databases. In *SUM*, pages 326–340, 2008.

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

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

[32] J. Peer and R. Pinter. Minimal decomposition of boolean functions using non-repeating literal trees. In *IFIP Workshop on Logic and Architecture Synthesis*, 1995.

[33] S. Roy, V. Perduca, and V. Tannen. Faster query answering in probabilistic databases using read-once functions. *CoRR*, abs/1012.0335, 2010.

[34] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. In *To appear in VLDB*, 2010.

[35] J. Spiegel and N. Polyzotis. Graph-based synopses for relational selectivity estimation. In *SIGMOD Conference*, pages 205–216, 2006.

[36] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[37] E. Zimányi. Query evaluation in probabilistic relational databases. *Theor. Comput. Sci.*, 171(1-2):179–219, 1997.