Read-Once Functions and Query Evaluation in Probabilistic Databases

Prithviraj Sen [†1], Amol Deshpande [*2], Lise Getoor [*3]

[†]Yahoo! Labs
Bangalore, India

[*]Computer Science Department
University of Maryland, College Park, USA

[1]sen@yahoo-inc.com, [2]amol@cs.umd.edu, [3]getoor@cs.umd.edu

ABSTRACT

Probabilistic databases hold promise of being a viable means for large-scale uncertainty management, increasingly needed in a number of real world applications domains. However, query evaluation in probabilistic databases remains a computational challenge. Prior work on efficient *exact* query evaluation in probabilistic databases has largely concentrated on query-centric formulations (e.g., *safe plans*, *hierarchical queries*), in that, they only consider characteristics of the query and not the data in the database. It is easy to construct examples where a supposedly hard query run on an appropriate database gives rise to a tractable query evaluation problem. In this paper, we develop efficient query evaluation techniques that leverage characteristics of both the query and the data in the database. We focus on tuple-independent databases where the query evaluation problem is equivalent to computing marginal probabilities of Boolean formulas associated with the result tuples. Query evaluation is easy if the Boolean formulas can be factorized into a form that has every variable appearing at most once (called *read-once*); this suggests a naive approach that incorporates previously developed Boolean formula factorization algorithms into the query evaluation. We then develop novel, more efficient factorization algorithms that work for a large subclass of queries (specifically, conjunctive queries without self-joins), by exploiting the unique structure of the result tuple Boolean formulas. We empirically demonstrate that our proposed techniques are (1) orders of magnitude faster than generic inference algorithms when used to evaluate general read-once functions, and (2) for the special case of hierarchical queries, they rival the efficiency of prior techniques specifically designed to handle such queries.

1. INTRODUCTION

The rise in uncertain data in a variety of applications has led to much research in the area of probabilistic databases in recent years [10, 1]. Since query evaluation is #P-complete in probabilistic databases that use the possible world semantics (the dominant semantics on which most systems are based), two broad approaches have been suggested for tractable query execution. Either the system opts to compute approximate query results [21, 18, 25], or the query language is restricted to allow for efficient exact query evaluation. The research on hierarchical queries [11, 25] has identified many such classes of queries (reviewed in the next section) that can be evaluated in PTIME on tuple-independent databases (mutual exclusivity correlations may be permitted in some cases). Unfortunately several negative dichotomy and trichotomy results [11, 21, 26] developed in recent years suggest that the class of hierarchical queries is likely to be too restrictive. More importantly, a query-centric approach is pessimistic by definition; it may be the case that a non-hierarchical query can be tractably evaluated on most probabilistic databases encountered in practice.

In this paper, we develop novel techniques for query evaluation in tuple-independent probabilistic databases by drawing connections to the literature on *read-once functions*. It is well known that in probabilistic databases with independent tuples, every result tuple is associated with a Boolean formula [3, 13] (often called *lineage*), and the query evaluation problem reduces to computing the marginal probabilities for the result tuple Boolean formulas holding true. Further, if a result tuple's Boolean formula can be factorized into a form where every Boolean variable appears at most once, also known as *read-once functions* [15, 17], then the marginal probability can be computed easily in linear time. Hierarchical queries always admit a query evaluation plan such that the result tuple formulas are generated in read-once form, thus providing a connection to efficient query evaluation in probabilistic databases.

Here we further explore and exploit the connection by first showing how to incorporate previously developed Boolean formula factorization algorithms into a probabilistic database query engine. In this naive approach, we run the user-submitted query using any query plan to generate all the result tuples along with their lineages, and then we factorize them into read-once form (if possible) to compute marginal probabilities. This approach by itself allows us to tackle a superset of hierarchical queries more efficiently than has been previously possible. However, it is quite inefficient because prior factorization algorithms require the disjunctive normal form (DNF) of the input Boolean formula; DNF expansion of a formula may result in an exponential blowup (in the size of the query).

We instead develop novel algorithms that allow us to construct the read-once expression for an output tuple (if one exists) on-the-fly during query evaluation, for the class of *conjunctive queries without self-joins*. Our proposed technique subdivides the result tuple lineage into smaller formulas for which we build read-once expressions separately. We then merge these read-once expressions to produce a read-once expression of the result tuple lineage. Both these steps are challenging. In the first step, we have to carefully strategize how to sub-divide the result tuple lineage into smaller components. The divisions should be such that if a read-once expression for a sub-division does not exist then the result tuple may

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

L:

	X
x_1	x_1
x_2	x_2
x_3	x_3

J:

	X	**Y**
z_1	x_1	y_1
z_2	x_1	y_2
z_3	x_2	y_3
z_4	x_3	y_3

R:

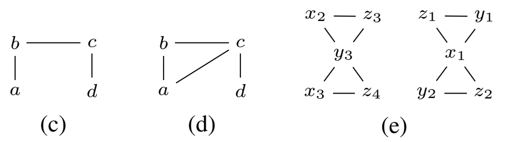
	Y
y_1	y_1
y_2	y_2
y_3	y_3

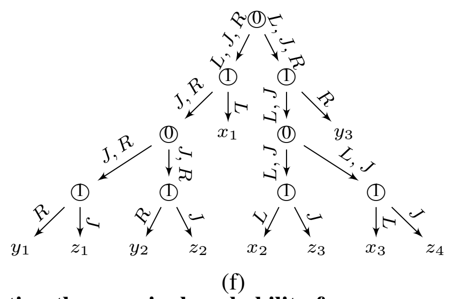
$$q() :- L(\mathbf{X}), J(\mathbf{X},\mathbf{Y}), R(\mathbf{Y})$$
$$r = x_1z_1y_1 + x_1z_2y_2 + x_2z_3y_3 + x_3z_4y_3$$

(a)

$$Pr(e) = \begin{cases} \prod_{c \in ch(e)} Pr(c) & \text{if } v \text{ is } \textcircled{1} \\ 1 - \prod_{c \in ch(e)} (1 - Pr(c)) & \text{if } v \text{ is } \textcircled{0} \\ p_t & \text{if } v = x_t \end{cases}$$

(b)





(f)

Figure 1: (a) A query q and its singleton result r along with its Boolean formula. (b) Computing the marginal probability from a co-tree where $ch(e)$ denotes children of v. (c) $ab + bc + cd$, the P_4 structure. (d) $c(ab + d)$, co-occurrence graph is P_4-free. (e) The co-occurrence graph of r from (a), and (f) its co-tree.

not have a read-once expression. In the main body of the paper, we show that using deep query evaluation plans (where at least one input to every join operator is a base relation) it is possible to divide the result tuple's lineage into such sub-divisions. In the appendix, we show that the same logic can extended to include all query plans. The second step is where we combine the read-once expressions of the various sub-divisions together. Read-once expressions are usually represented using tree-like data structures known as *co-trees* [6]. Thus, this step is an instance of the *tree alignment problem* [20] which in general is MAX SNP-hard; we propose a novel polynomial-time algorithm for solving this problem in our setup. We evaluate our approach by experimenting with several datasets, including the TPC-H benchmark. We show that not only are our proposed techniques orders of magnitude faster than generic inference algorithms when evaluating non-hierarchical queries, they rival techniques specifically developed to handle hierarchical queries also.

Two of the state-of-the-art probabilistic database systems that solve hierarchical queries efficiently are heavily dependant on query compilation techniques. MystiQ [4] looks at the query and constructs a plan that allows efficient evaluation. SPROUT [24] takes the query's hierarchical representation and constructs a *signature* which is used in subsequent operations. These works also propose the use of functional dependencies described in the schema to simplify the query. It is possible that a non-hierachical query results in a hierarchical query when the functional dependencies are taken into account. Our premise, in this paper, is very simple. If the query results in read-once result tuples (either because of the structure in the query or because the data satisfies functional dependencies or for any other reason) then we should be able to compute marginal probabilities easily. In that sense, the techniques we propose in this paper are data-centric and largely disjoint from the techniques employed in SPROUT and MystiQ which are query-centric. This obviates the need for a fair amount of compile time analysis involving the query and functional dependencies. We evaluate against SPROUT and MystiQ extensively in Section 4.

Recently, there have also been proposals to use factorization algorithms within a probabilistic database query engine [25]. However, [25] assumes the DNF of the result tuple is provided as input which, as we mentioned earlier, can be expensive to obtain. Moreover, that work concentrates on approximate evaluation whereas, in this paper, we are more interested in exact query evaluation. Another issue with the current work on hierarchical queries is that most of those deal almost exclusively with equality join predicates [9, 11][*]. Extending the approach to other operators requires effort and dealing with queries composed of different operators is even more cumbersome. On the contrary, viewing result tuples as Boolean formulas allows us to restrict our attention to only two operators, viz. $\wedge$ (`and`) and $\vee$ (`or`). We do not care what kind of join predicate (equality or inequality or anything else) gave rise to the Boolean formula associated with the result tuple. Thus, our techniques are likely to be more widely applicable than prior work.

[*] [23, 22] are notable exceptions.

Outline: In the next section, we discuss preliminaries and show how to incorporate prior factorization algorithms into the query engine. In Section 3, we devise novel and efficient algorithms to construct read-once functions for result tuples produced by conjunctive queries without self-joins. In Section 4, we experimentally demonstrate the efficacy of our proposed techniques. In Section 5, we discuss related work before concluding with Section 6. The techniques proposed in this paper are sound and complete; full proofs appear in the accompanying appendix due to space constraints.

2. PRELIMINARIES

Let R denote a relation defined over a set of attributes $Attr(R)$. Each tuple $t \in R$ is a mapping from $Attr(R)$ to values. We also associate a unique (Boolean-valued) random variable with t denoted by x_t and a probability of existence p_t. Often, when it is clear from the context, we will abuse notation and refer to the tuple's random variable by the tuple itself. A (probabilistic) database $D = \{R_1, \ldots R_m\}$ is a set of relations and represents a distribution over many possible worlds each obtained by choosing a (sub)set of tuples in R_i to be present. If t is present we say x_t is assigned `true` or `t` and `false` or `f`, otherwise. Each possible world w is associated with a probability:

$$Pr(w) = \prod_{i=1}^{m} \prod_{t \in R_i, x_t = \mathtt{t}} p_t \prod_{t \in R_i, x_t = \mathtt{f}} (1 - p_t)$$

Given a query q to be evaluated against database D, the result of the query is defined to be the union of results returned by each possible world along with the marginal probabilities of each result tuple [9]. More precisely, the marginal probability of result tuple t is obtained by adding the probabilities of all possible worlds that return t as a result: $\mu(t) = \sum_{t \in q(w)} Pr(w)$, where $q(w)$ denotes the result of the query on that possible world.

One way to compute the marginal probability of result tuple t produced by (relational algebra) query q is to extend each (relational algebra) operator in q so that it builds a Boolean formula for each (intermediate) tuple generated during query evaluation. We refer to the Boolean formula for t by ϕ_t. Below we show these extended definitions for operators σ, $\times$ and $\prod$.

$$\phi_t = x_t \; \forall t \in R, \quad \phi_{\sigma_c(t)} = \mathtt{if}\ c(t)\ \mathtt{then}\ \phi_t\ \mathtt{else}\ \mathtt{f},$$
$$\phi_{t \times t'} = \phi_t \wedge \phi_{t'}, \quad \phi_{\prod(t_1,\ldots t_k)} = \bigvee_{i=1}^{k} \phi_{t_i}$$

The marginal probability of the result tuple can then be obtained by computing the probability of the corresponding Boolean formula holding true. We refer the interested reader to prior work that illustrates the equivalence between query evaluation under possible world semantics and via Boolean formulas [3, 13, 27]. Figure 1(a) shows a three-relation join query which produces a singleton result and the corresponding result tuple's Boolean formula.

2.1 Read-Once Functions; Definitions

Although marginal probability computation is #P-Complete in general, efficient computation is possible in many cases.

DEFINITION 1 (**Read-Once Function** [17]). *A Boolean formula ϕ is read-once if there exists a factorization such that each variable appears not more than once.*

Further, the read-once factorized form of the Boolean formula is known as its read-once expression. For instance, r in Figure 1(a) is a read-once function with the read-once expression $x_1(z_1y_1 + z_2y_2) + y_3(x_2z_3 + x_3z_4)$.

THEOREM 1 ([15]). *A Boolean formula is read-once iff it is unate, P_4-free and normal.*

A Boolean formula ϕ is **unate** [15] if every variable only appears in either its positive or negated form. E.g., ab and $\bar{a}b + \bar{a}c$ are unate but $\bar{a}b + ac$ is not. For any Boolean formula ϕ, the **co-occurrence graph** G_ϕ is formed by drawing a vertex for every variable and drawing an edge between two variables if they appear in the same clause in ϕ's disjunctive normal form (DNF). Let X denote a subset of vertices, then the subgraph of G induced by X is the subgraph formed by restricting edges of G to edges with end points in X.

The special graph P_4 denotes a chordless path with 4 vertices and 3 edges (Figure 1(c)). ϕ is P_4-free if no induced subgraph of G_ϕ forms a P_4. Figure 1(c) and (d) show two formulas and the former is not read-once because it forms a P_4. Figure 1(d) is P_4-free because of the presence of the edge $a - c$. Figure 1(e) shows the co-occurrence graph for the result tuple r from Figure 1(a). A formula ϕ is said to be **normal** if every clique in its co-occurrence graph is contained in some clause in its DNF [15]. For instance, even though both $\phi_1 = abc$ and $\phi_2 = ab + bc + ca$ have the same co-occurrence graph (the triangle), ϕ_1 is normal while ϕ_2 is not.

Traditionally, **co-trees** [6] have been used to represent read-once expressions. Co-trees are trees where leaves correspond to Boolean variables while internal node ① represents $\wedge$ and ⓪ represents $\vee$. A given read-once expression can be represented by many co-trees but there exists a canonical co-tree, where ① and ⓪ alternate on every path. See Figure 1(f) for an example. Given the co-tree for a read-once result tuple, the marginal probability can be computed using the simple, bottom-up procedure shown in Figure 1(b).

Our goal in this paper is essentially to construct the co-trees corresponding to all the result tuple lineages efficiently, if they exist.

2.2 Hierarchical Queries

Earlier work on query evaluation in probabilistic databases has identified tractable queries for which probability computation is efficient and this set of queries is referred to as *hierarchical queries* [11]. For the ensuing discussion, we will assume that all queries are projected onto the empty set (in other words, our goal is simply to compute the probability that there exists at least one result tuple). Queries projected onto a non-empty set of attributes can be handled by replacing these attributes with constants. Let q denote a query in Datalog notation. Let a denote an attribute in q and $sg(a)$ denote the set of relations in q that refer to a. In Figure 1(a), $sg(\mathbf{X}) = \{L, J\}$, $sg(\mathbf{Y}) = \{J, R\}$.

DEFINITION 2 (**Hierarchical Query** [11]). *A (conjunctive) query q is hierarchical if, for any two attributes a and b, either $sg(a) \subseteq sg(b)$, $sg(b) \subseteq sg(a)$ or $sg(a) \cap sg(b) = \emptyset$.*

For instance, the query q in Figure 1(a) is not hierarchical ($sg(\mathbf{X}) \cap sg(\mathbf{Y}) = \{J\} \neq \emptyset, sg(\mathbf{X}) \not\subseteq sg(\mathbf{Y}), sg(\mathbf{Y}) \not\subseteq sg(\mathbf{X})$) but $q'() :- S(\mathbf{X}, \mathbf{Y}), T(\mathbf{Y})$ is (because $sg(\mathbf{Y}) = \{S, T\} \supset \{S\} = sg(\mathbf{X})$). [11] (also, [22]) illustrated the connection between hierarchical queries and read-once functions:

PROPOSITION 1 ([11]). *Hierarchical queries produce result tuples with read-once expressions.*

The converse is clearly not true. In Figure 1(a), we show a query that is not hierarchical, but the result tuple it produces has a read-once expression (Figure 1(f)). In this paper, we would like to develop techniques that help us identify such cases efficiently.

2.3 A Naive Approach

One viable approach to evaluating queries is to generate Boolean formulas for result tuples (using the extended operators shown earlier) and then determine whether it is a read-once function. If the result tuple lineage is read-once then we compute its probability from its read-once expression's co-tree, else we resort to a general-purpose inference engine or approximations.

Checking for read-once result tuples is possible in polynomial time. In what follows, let ϕ denote a result tuple's Boolean formula in DNF, $|\phi|$ its length (with different occurrences of the same variable counted multiple times) and $Vars(\phi)$ the set of distinct variables in it. To check for unateness, a linear scan of the formula is sufficient which requires $O(|\phi|)$ time. To check for P_4's in ϕ's co-occurrence graph G_ϕ, a handful of algorithms are available [7, 16, 5]. The common aspect of all of these algorithms is that they all require G_ϕ to be provided as input and they run in time linear in size of G_ϕ ($O(|Vars(\phi)| + |E|)$, where $|E|$ is the number of edges in G_ϕ). G_ϕ can be obtained easily from the formula's DNF. A nice property of these algorithms is that not only do they check for the absence of P_4's, but they also return the co-tree representing the read-once expression which can subsequently be used for probability computation. Checking for normality is also possible in polynomial time. [15] describes a way to do this in $O(|Vars(\phi)||\phi|)$ time using the co-tree obtained in the previous step.

2.4 Limitations

Even though the above approach to query evaluation is both sound and complete, it can be expensive for large probabilistic databases. Two steps in our query evaluation procedure are of particular concern: (1) checking for normality because of its quadratic time complexity, and (2) checking for P_4's. All the algorithms that check for P_4's that we are aware of expect the co-occurrence graph (G_ϕ) as input for which we will likely require the formula to be presented in DNF. Our extended relational algebra operators do not guarantee that a result tuple's formula will be returned in its DNF form (instead they return a more compact representation that we call **lineage-trees**; see next section). Moreover, converting a formula into its DNF form can result in an exponential blowup. More specifically, if a result tuple produced by query q involves n tuples from each relation and there are k relations being joined in q, then the corresponding formula's DNF form could require $O(n^k)$ units of space and time to compute.

3. READ-ONCE EXPRESSIONS FOR CONJUNCTIVE QUERIES

In this section, we present our key technical contribution: an efficient algorithm for directly constructing co-trees corresponding to result tuple lineages (if they exist) for conjunctive queries without self-joins, executed using a deep query evaluation plan. Conjunctive queries form a fairly large subset of database queries, and a number of prior works in probabilistic databases have focused their attention to (subclasses of) conjunctive queries [9, 22].

We begin with formally defining conjunctive queries, and observing a key property of the result tuple lineages generated by a conjunctive query. This property enables us to eliminate the *normality-checking* step, resulting in significant efficiency gains (Section 3.1). We then show how to sub-divide the result tuple lineage so that co-trees can be built for resulting sub-formulas independently of each other; if any of the resulting sub-formulas is not P_4-free, then the result tuple lineage is guaranteed to contain a P_4 (Section 3.2). We then develop an algorithm for merging the co-trees corresponding to the sub-formulas (Section 3.3). We end the section with a discussion of further optimizations (Section 3.4).

3.1 Properties of Conjunctive Queries

Let A denote an attribute. An *atomic predicate* is of the form $A\ op\ B$ where B is either an attribute or a constant conforming to the type of A and op is any binary operator conforming to the same type such as $=, >, <, \neq$ etc. A *conjunctive query* q involving the set of relations $Rels(q) = \{R_1, \dots R_k\}$ is a relational algebra query that involves the three operators σ, $\bowtie$ and $\prod$ such that all selection and join predicates are either atomic predicates or conjunctions of atomic predicates. If $Rels(q)$ does not contain any repeated relations then q is a conjunctive query *without self-joins*.

Since conjunctive queries do not allow negations, the result tuples' Boolean formulas will be unate (variables appear only in their positive form). We next show that for result tuples we will be dealing with, the normality check is also not required. In the ensuing discussion, let $\phi_{DNF} = C_1 + C_2 + \cdots$ denote the DNF of Boolean formula ϕ where each C_i denotes a clause or a conjunction of Boolean variables $x_1x_2\dots$. Often we will treat clauses as sets, and use $x \in C$ to denote a variable x present in the clause C.

For result tuple ϕ produced by conjunctive query q without self-joins, two properties easily follow. First, every clause C in ϕ_{DNF} consists of exactly one tuple existence random variable from each relation in $Rels(q)$. This is often referred to as ϕ_{DNF} being a k-monotone DNF. Second, given a collection T of k $(= |Rels(q)|)$ tuples, one each from a different relation, if T satisfies all the selection and join predicates in q and agrees with ϕ on the final set of projected attributes, then T must form a clause in ϕ_{DNF}. The second property can be restated in terms of ϕ's co-occurrence graph G_ϕ. If G_ϕ contains a k-sized clique then it must form a clause in ϕ_{DNF}. We now state the main result of this subsection (proof can be found in the appendix):

THEOREM 2. *Let ϕ denote a result tuple produced by a conjunctive query without self-joins. If ϕ is P_4-free, then it is normal.*

Theorem 2, along with the guarantee of unateness, means that whenever we are working with result tuples produced by conjunctive queries without self-joins we only need to check if they are P_4-free. As long as they are P_4-free we can rest assured that a read-once expression exists. Recall that, the normality check was the most expensive step while checking for read-once functions. So this represents a significant gain. We next proceed towards constructing an efficient P_4-checking algorithm.

3.2 Piecewise Building of Co-Trees

Like prior P_4-checking algorithms, we would prefer an algorithm that not only checks for the absence of a P_4, but also returns the co-tree representation of the read-once expression (if no P_4 exists). Recall that, unlike other algorithms, we don't want use the co-occurrence graph or the DNF expression as input to our P_4-checking algorithm. Instead we use the representation of the re-

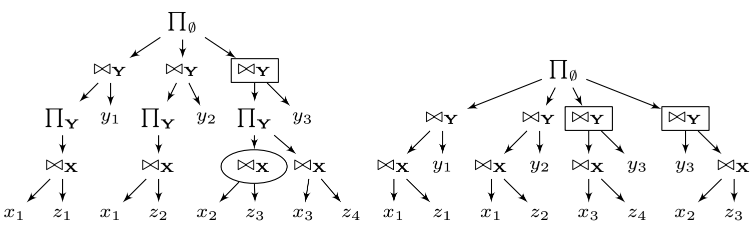


Figure 2: Lineage-trees of r from Figure 1(a) using plans: (a) $\prod_\emptyset(R \bowtie_\mathbf{Y} \prod_\mathbf{Y}(L \bowtie_\mathbf{X} J))$, (b) $\prod_\emptyset(R \bowtie_\mathbf{Y} (L \bowtie_\mathbf{X} J))$.

sult tuple Boolean formula as constructed by the query engine during evaluation. That way we do not have to make any significant changes to the query engine, and further the query optimizer is free to choose the best query evaluation plan (subject to the constraint that the query plan be deep).

We call such a representation a **lineage-tree**. A lineage-tree depicts how tuples join and project among one another to produce the result tuple. Going back to the example in Figure 1 (a), Figure 2 (a) shows the lineage-tree of r obtained using the query plan $\prod_\emptyset(R \bowtie_\mathbf{Y} \prod_\mathbf{Y}(L \bowtie_\mathbf{X} J))$ for q. Note that, lineage-tree is not the same as the co-tree (x_1 appears twice in Figure 2 (a)). Further note that, the lineage-tree of the result tuple depends on the query plan used. Figure 2 (b) shows the lineage-tree obtained using the plan $\prod_\emptyset(R \bowtie_\mathbf{Y} (L \bowtie_\mathbf{X} J))$ instead. Part of our challenge now is to design an algorithm that constructs the correct read-once expression given any lineage-tree of the result tuple.

In what follows, we need to refer to nodes in the lineage-tree and for this, we need some extra notation. Let ∂ denote a *left/right deep plan* of query q. More specifically, let ∂ denote a plan such that every join in the plan involves at least one base relation. Without loss of generality, we will assume ∂ is of the form $op_1(R_1 \bowtie_{\theta_1} op_2(R_2 \bowtie_{\theta_2} \dots))$ where θ_i denotes a join predicate and op_i denotes a sequence of projections and selections (op_i could also be identity). Lineage-trees produced by left/right deep plans are such that every join node ($\bowtie$) always has at least one base tuple's existence variable as a child. In fact, we can use this as a way of identifying nodes in the lineage-tree.

DEFINITION 3 (JOIN PATH, COFACTOR). *Let $\bowtie^k$ denote a join node in lineage-tree $\mathcal{L}$. Let $\bowtie^{k-1}, \dots \bowtie^1$ denote all join nodes along the path from $\bowtie^k$ to the root of $\mathcal{L}$ such that $\bowtie^{i-1}$ is $\bowtie^i$'s immediate ancestor. Further, let b_i denote (one of) the base tuple child of $\bowtie^i$. Then we refer to the sequence $[b_1, b_2 \dots b_k]$ as a* join path *and $\bowtie^k$ as a join node identified by it. Further, we denote the remaining child of $\bowtie^k$ as $cofactor_\mathcal{L}([b_1, b_2 \dots b_k])$ or $cof_\mathcal{L}([b_1, \dots b_k])$, in short.*

For instance, in $\mathcal{L}$ shown in Figure 2 (a), join path $[y_3]$ identifies the node enclosed in the rectangle, $[y_3, x_2]$ identifies the node enclosed in the ellipse and $cof_\mathcal{L}([y_3, x_2])$ is z_3.

In general, a join path identifies a set of nodes in the lineage-tree. For instance, in lineage-tree $\mathcal{L}$ in Figure 2 (b), join path y_3 identifies the two nodes enclosed in rectangles. In such cases, we define the cofactor to be the *set* of non-base tuple children of the identified join nodes. Alternatively, we can also represent the cofactor by its Boolean formula, in other words the disjunction of the formulas of the (intermediate) tuples representing the nodes the cofactor contains. For instance, in Figure 2 (b), $cof_\mathcal{L}([y_3]) = x_3z_4 + x_2z_3$. Another interesting aspect of cofactors is that they can be expressed in a recursive manner. $cof_\mathcal{L}([\Gamma])$ can be expressed in terms of $cof_\mathcal{L}([\Gamma, b])$, where b denotes a base tuple. For instance, in $\mathcal{L}$ in Figure 2 (b), $cof_\mathcal{L}([y_3]) = x_3cof_\mathcal{L}([y_3, x_3]) + x_2cof_\mathcal{L}([y_3, x_2])$.

1. $\mathcal{T}(b_1, \dots b_n) = \textcircled{0}(b_1, \dots b_n),$ s.t. $b_i \in R$

2. $\cup\,(op(t^1_1, \dots t^1_n), t_2 \dots t_m) = \{t^1_1, \dots t^1_n, \cup(t_2, \dots t_m)\}$

3. $\mathcal{T}(op(t^1_1, \dots t^1_n), t_2, \dots t_m) = \mathcal{T}(\cup(op(t^1_1, \dots t^1_n), t_2, \dots t_m))$

4. $\mathcal{T}(b_1 \bowtie t_1, \dots b_n \bowtie t_m) = \textcircled{1}(b_1, \mathcal{T}(t^{b_1}_1, \dots t^{b_1}_{m_1}))$
$\oplus\, \textcircled{1}(b_2, \mathcal{T}(t^{b_2}_1, \dots t^{b_2}_{m_2})) \dots \oplus\, \textcircled{1}(b_n, \mathcal{T}(t^{b_n}_1, \dots t^{b_n}_{m_n}))$

Figure 3: Co-tree building procedure. op **is** $\prod$ **or** σ**.**

This also illustrates how $cof_{\mathcal{L}}([\Gamma, b])$ is a sub-formula of $cof_{\mathcal{L}}([\Gamma])$ and is contained within it.

Just like many other tree-building algorithms, our co-tree building algorithm is a recursive algorithm that builds co-trees for sub-formulas of the result tuple and combines them to construct the complete read-once expression. However, choosing sub-formulas is tricky – this is because having a P_4 is not a monotonic property. Let $\phi = \phi_1 + \phi_2$. Then ϕ_1 (or ϕ_2) containing a P_4 does not imply that ϕ contains a P_4. This is illustrated by the formula $a_1b_1x_1y_1 + a_1b_2x_1y_2 + a_2b_1x_2y_1 + a_2b_2x_2y_2$ where the sum of any three terms gives us a P_4 even though the complete formula is read-once ($(a_1x_1 + a_2x_2)(b_1y_1 + b_2y_2)$). Figure 10 in the appendix shows the query and database that produce this result tuple.

In the previous example, essentially, picking any three summands gives us a P_4 for which the fourth summand provides a chord. For instance, if we set $\phi_1 = a_1b_1x_1y_1 + a_1b_2x_1y_2 + a_2b_1x_2y_1$, which contains the $P_4 : b_2 - a_1 - b_1 - a_2$, then the fourth summand $\phi_2 = a_2b_2x_2y_2$ contains the edge $a_2 - b_2$. Recall that, a P_4 is a *chordless* path of length 3 edges. Thus, when we consider $\phi_1 + \phi_2$ we no longer have the P_4[†]. To express this more concretely we define the notion of *interference*:

DEFINITION 4 (INTERFERENCE, TYPE 1). *Two formulas* ϕ_1 *and* ϕ_2 interfere *if* $\exists a, b \in Vars(\phi_1), Vars(\phi_2)$ *such that* $a - b \notin G_{\phi_1}$ *and* $a - b \in G_{\phi_2}$.

where $a - b \notin G$ denotes that a and b do not form an edge in G. Interference can also arise due to a triplet of formulas:

DEFINITION 5 (INTERFERENCE TYPE 2). *Three formulas* ϕ_1, ϕ_2 *and* ϕ_3 interfere *if* $\exists a, b$ *such that* $a \in Vars(\phi_1), a \notin Vars(\phi_2), b \notin Vars(\phi_1), b \in Vars(\phi_2)$ *and* $a - b \in G_{\phi_3}$.

Essentially, if we consider $\phi_1 + \phi_2$ as one formula then type 2 interference reduces to type 1 interference.

For our task of building read-once expressions, we would like to divide the result tuple into non-interfering sub-formulas; and this is where the lineage-tree, with its cofactors, comes to our rescue:

LEMMA 1. *Let* ϕ *denote a result tuple,* $\mathcal{L}$ *its lineage and* Γ *a join path. If* ϕ *is* P_4*-free then the set of cofactors* $\{cof_{\mathcal{L}}([\Gamma, b]) | b$ *is a base tuple and a valid extension of* $\Gamma\}$ *is non-interfering.*

For instance, in Figure 2(a), the base tuples y_1, y_2 and y_3 form valid extensions of the empty join path. The lemma says that $cof_{\mathcal{L}}([y_1]) = x_1z_1$, $cof_{\mathcal{L}}([y_2]) = x_1z_2$ and $cof_{\mathcal{L}}([y_3]) = x_2z_3 + x_3z_4$ do not interfere with each other if the lineage of r is read-once.

The lemma immediately gives us a way to sub-divide the result tuple lineage properly, to build its co-tree. Figure 3 outlines a recursive procedure $\mathcal{T}$ that takes a number of (base or intermediate) tuples $t_1, t_2, \cdots, t_n$ as arguments and returns the co-tree representing $\phi_{t_1} \vee \phi_{t_2} \vee \cdots \vee \phi_{t_n}$, if possible (recall that ϕ_t denote the lineage of tuple t). $\mathcal{T}$ begins at the root of the input lineage-tree and proceeds downwards.

[†]ϕ_2 also has a chord for ϕ_1's other P_4, $y_2 - x_1 - y_1 - x_2$.

1. $\mathcal{S}(x, y) = \begin{cases} 1 & \text{if } x \equiv y \\ 0 & \text{otherwise} \end{cases}$

2. $\mathcal{S}(T_1, T_2) = 0,$ if $\neg\mathcal{C}(T_1) \wedge \neg\mathcal{C}(T_2)$

3. $\mathcal{S}(\textcircled{0}(A_1, A_2, \dots A_n), \textcircled{1}(\dots)) = \max\limits_{i=1}^{n}(\mathcal{S}(A_i, \textcircled{1}(\dots)))$

4. $\mathcal{S}(\textcircled{0}(A_1, A_2, \dots A_n), \textcircled{0}(B_1, \dots B_m)) =$
$$\begin{cases} \mathcal{S}(\textcircled{0}(A_2, \dots A_n), \textcircled{0}(B_1, B_2, \dots B_m)), \text{ if } \neg\mathcal{C}(A_1) \\ \max_{i=1}^{m}(\mathcal{S}(A_1, B_i)) \\ \qquad + \mathcal{S}(\textcircled{0}(A_2, \dots A_n), \textcircled{0}(B_1, \dots B_m \setminus B_i)) \end{cases}$$

5. $\mathcal{S}(\textcircled{1}(\mathbf{A_{R_1}}, \mathbf{A_{R_2}}, \dots \mathbf{A_{R_k}}), \textcircled{1}(\mathbf{B_{R_1}}, \mathbf{B_{R_2}}, \dots \mathbf{B_{R_k}})) =$
$$\begin{cases} -\infty, \text{ if } \exists \mathbf{A_R}, \mathbf{B_{R'}} \text{ s.t. } \mathcal{N}(\mathbf{A_R}) \wedge \mathcal{N}(\mathbf{B_{R'}}) \\ \sum_{i=1}^{k} \mathcal{S}(\mathbf{A_{R_i}}, \mathbf{B_{R_i}}) \end{cases}$$

Figure 4: Computing score for $\oplus$ operation. In the last rule bold font denotes sets.

We use prefix notation to denote co-trees. So $\textcircled{0}(b_1, \dots b_n)$ indicates a $\textcircled{0}$ node with children $b_1, \dots, b_n$, where b_i may denote a leaf (base tuple) or another node in the co-tree.

The first rule (the base case) returns a co-tree representing the disjunction of numerous base tuples. The third rule, of which the second rule is a helper, simply goes down a node in the lineage-tree when it encounters a σ or a $\prod$ operator. Note that, rule 2 assumes that if the first tuple fed to $\mathcal{T}$ is a tuple produced by a selection (projection) then the rest of the tuples present in the argument list are also selection (projection) tuples. This assumption is guaranteed to hold because lineage-trees are produced by a query (plan) expressed in a high-level language (e.g., relational algebra).

The critical rule here is the fourth rule (join rule) which is a direct translation of Lemma 1. We denote by t^b a node in the lineage-tree (representing an intermediate tuple) that joins with base tuple b. In this case, given a set of join tuples we take each base tuple b_i and form its co-tree with the co-tree of its cofactor represented by the set of intermediate tuples it joins with. It is in this rule that the real benefit of Lemma 1 shows up. If a cofactor contains a P_4 then we know that no other cofactor can provide a chord for it since cofactors are non-interfering. This implies that if a cofactor contains a P_4 then the result tuple contains a P_4 and we can bail out immediately. In fact, Lemma 1 guarantees that if we discover a P_4 during the process of merging co-trees produced out of cofactors then the result tuple is not P_4-free. Note that, we form a co-tree per base tuple. Once the various $\mathcal{T}$ calls return, we need to combine them and form the disjunction of the various co-trees. Even though the cofactors do not interfere (if the result tuple is read-once), they can still have variables in common which is why we use the $\oplus$ operator to merge these co-trees and form their disjunction. We describe details of the $\oplus$ operator in the next subsection.

Note that, in Figure 3, at no stage do we compute the DNF of any formula. Also, one of the limitations of the approach presented here is that it only applies to left/right deep plans. The example presented earlier in the section to motivate the need for choosing non-interfering formulas is the result of a plan with a bushy join (Figure 10). See Appendix C for ways to handle such plans.

3.3 Merging Co-Trees

In this section, we discuss how to merge the co-trees generated in the recursive step above (in other words, the implementation of the $\oplus$ operator from Rule 4). For this purpose, we maintain and use an augmented version of a co-tree where every edge from a parent node u to child node v is annotated with a set of relations $\mathcal{A}(u \rightarrow v)$. $R \in \mathcal{A}(u \rightarrow v)$ if some tuple from R forms a leaf in the subtree rooted at v. Figure 1(f) shows the annotated co-tree for result tuple r in Figure 1(a). Two simple properties about annotated co-trees should be apparent. First, if $\exists R \in \mathcal{A}(u \rightarrow v_1), \mathcal{A}(u \rightarrow v_2)$, where u is a $\textcircled{1}$ node, then this would imply a self-join so this is not possible (recall that $\textcircled{1}$ represents an $\wedge$ which corresponds to a join operation). Further, let u denote any $\textcircled{0}$ node and T, the co-tree in which u is present. If T represents a k-monotone DNF, $\mathcal{A}(u \rightarrow v_1)$ has to be equal to $\mathcal{A}(u \rightarrow v_2)$ for any pair of children v_1, v_2.

We implement $\oplus$ as a binary operator using a recursive procedure. Let T_1 and T_2 denote the two input co-trees to be merged, then $\oplus(T_1, T_2)$ returns a pair consisting of a **score** and the resulting merged co-tree. Since the merged co-tree must necessarily have every variable as at most a single leaf, one of the main goals of $\oplus$ is to align the variables common to both T_1 and T_2. The score returned is simply the number of common variables aligned. Soundness of $\oplus$ follows the semantics of the various operations we subsequently describe. Completeness follows by showing that a P_4 exists if any of the assumptions we make do not hold. The score computation is described in Figure 4, where we use $\mathcal{S}(T_1, T_2)$ to denote the score of merging co-trees T_1 and T_2. We describe the creation of the merged co-tree in the text below. We bootstrap the merging process by computing two Boolean quantities for each node in either co-tree: $\mathcal{C}(u)$ is true if the subtree rooted at u contains any variable common to both input co-trees, and $\mathcal{N}(u)$ is true if it contains a variable not present in the other co-tree. At the end of the merging process before returning the result, we check if the returned score matches the number of variables common to both co-trees.

The first rule in Figure 4 is one of the base cases: given a pair of leaves x and y, it returns a score of 1 if they correspond to existence variables for the same tuple and 0 otherwise. In case they represent the same tuple, we return x, else we return $\textcircled{0}(x, y)$ as the merged co-tree. The second rule is always the first thing we check for. Given $\mathcal{S}(T_1, T_2)$, we return the score 0 if the co-trees don't contain any common variables. The merged co-tree is simply $\textcircled{0}(T_1, T_2)$. We may need to make minor adjustments to the returned co-tree so that it is canonical ($\textcircled{0}$ and $\textcircled{1}$ nodes alternate along every path).

Rule 3 in Figure 4 computes the score for merging co-trees $T_1 = \textcircled{0}(A_1, \ldots A_n)$ and $T_2 = \textcircled{1}(\ldots)$, where each A_i represents a sub-co-tree. Note that, T_1 and T_2 must share some common variables otherwise we would have applied rule 2. Here, we match T_2 versus each subtree A_i and pick the one whose score is the highest. If A_k denotes the best match then the returned merged co-tree is $\textcircled{0}(A_1, \ldots A_{k-1}, A_{k+1}, \ldots A_n, A)$ where A denotes the co-tree obtained by merging A_k and T_2. Note that we do not compare T_2 to sets of subtrees $\{A_i\}$. It can be shown that for a successful merge all common variables present in T_2 must be present in the same subtree A_k. In rule 4, we merge two co-trees $T_1 = \textcircled{0}(A_1, \ldots A_n)$ and $T_2 = \textcircled{0}(B_1, \ldots B_m)$. In this case also, we follow similar steps. For any A_i, we first check if $\mathcal{C}(A_i)$ is `true`. If yes, then we find the best matching subtree in T_2. We do this for every subtree A_i. The returned merge result is a co-tree rooted at a $\textcircled{0}$ node with A_i as a subtree, if $\mathcal{C}(A_i)$ is `false`, otherwise we add the co-tree obtained by merging A_i with B_k as a subtree, where B_k is the subtree from T_2 A_i best matches with. The same lemma, referred to above, applies. We do not need to match sets of subtrees of T_1 and T_2.

The last rule merges co-trees of the form $T_1 = \textcircled{1}(A_{\mathcal{A}_1}, \ldots A_{\mathcal{A}_n})$ and $T_2 = \textcircled{1}(B_{\mathcal{A}_1}, \ldots B_{\mathcal{A}_m})$, where the subscripts denote the annotations on the edges connecting roots of the co-trees to roots of the subtrees. In this case, we first attempt to form disjoint minimal subsets of sub-trees $\{A_{\mathcal{A}_{i_1}}, \ldots A_{\mathcal{A}_{i_k}}\}$ and $\{B_{\mathcal{A}_{j_1}}, \ldots B_{\mathcal{A}_{j_l}}\}$ such that $\mathcal{A}_{i_1} \cup \ldots \mathcal{A}_{i_k} = \mathcal{A}_{j_1} \cup \ldots \mathcal{A}_{j_l}$. In other words, each pair of minimal subset of sub-trees should contain variables from the same (sub)set of relations. Let $\mathbf{A_R}$ denote the minimal subset of subtrees from T_1 involving variables from the set of relations $\mathbf{R}$, similarly, let $\mathbf{B_R}$ denote its counterpart from T_2. For example, if $T_1 = \textcircled{1}(A_{R_1}, A_{R_2,R_3})$ and $T_2 = \textcircled{1}(B_{R_1}, B_{R_2}, B_{R_3})$ denote the input co-trees, then $\mathbf{A}_{\{R_1\}} = \{A_{R_1}\}$ and $\mathbf{B}_{\{R_1\}} = \{B_{R_1}\}$ and $\mathbf{A}_{\{R_2,R_3\}} = \{A_{R_2,R_3}\}$ and $\mathbf{B}_{\{R_2,R_3\}} = \{B_{R_2}, B_{R_3}\}$. Subsequently, we attempt to merge each pair $\mathbf{A_R}$ and $\mathbf{B_R}$. The returned score is the sum of scores returned by the recursive calls. If either of $\mathbf{A_R}$ or $\mathbf{B_R}$ is not a singleton set then we first create a dummy $\textcircled{1}$ root to form a single co-tree out of the non-singleton set before making the recursive merge call. It can be shown that if both $\mathbf{A_R}$ and $\mathbf{B_R}$ are non-singleton then a P_4 exists. It can also be shown that if $\mathbf{A_R}$ and $\mathbf{B_{R'}}$ are such that $\mathcal{N}(\mathbf{A_R}) \wedge \mathcal{N}(\mathbf{B_{R'}})$ is true then the merge result has a P_4. Otherwise, the merge result is a co-tree rooted at a $\textcircled{1}$ node with the co-trees returned from the recursive calls as it child subtrees. An exception to the condition involving $\mathbf{A_R}$ and $\mathbf{B_{R'}}$ described above is when none of $\mathbf{A_R}, \mathbf{B_R}, \mathbf{A_{R'}}$ or $\mathbf{B_{R'}}$ have any common variables. Here, we add $\textcircled{0}(\textcircled{1}(\mathbf{A_R}, \mathbf{A_{R'}}), \textcircled{1}(\mathbf{B_R}, \mathbf{B_{R'}}))$ as a subtree to the merge result.

Time Complexity: It can be shown that the time complexity of $\oplus(T_1, T_2)$ is $O(nmk^2)$, where n and m denote the number of nodes from the two co-trees and k denotes the number of relations the leaves from either co-tree belong to. Note that, the most expensive rule in Figure 4 is rule 5, for which we need to do, at most, $k^2 + k$ work (dividing the subtrees based on their annotations can be done in k^2 time). For the simpler case where no new dummy nodes get created (due to rule 5) during the merging process, applying this rule to every pair of nodes from both co-trees incurs a complexity of $O(nmk^2)$. For the more complex case when rule 5 generates dummy nodes also, it is possible to show that this complexity does not increase. However, based on our empirical observations (reported in the next section), this time complexity is rarely, if ever, achieved. In the next subsection, we describe a handful of optimizations that can further reduce the time spent to construct co-trees for result tuples. Coupled with the fact that to construct the co-tree we need simply make one pass over its lineage-tree, means that we have a reasonably efficient approach to computing the marginal probabilities of read-once result tuples produced by conjunctive queries without self-joins.

3.4 Further Optimizations

A number of optimizations are possible in the basic algorithms we presented in this section. One reason why our co-tree building algorithm (Figure 3) slows down is because of the join rule wherein we construct one co-tree per base tuple. Let $\mathbf{T} = \{b_1 \bowtie t_1, \ldots, b_n \bowtie t_m\}$ denote a set of (intermediate) join tuples. We say that $\mathbf{T}$ is **block-structured** if it can be rearranged into the form $B_1 \bowtie T_1, \ldots, B_k \bowtie T_j$ where the B_is and T_is form a disjoint partitioning, or blocks, over the base tuples $\{b_1, \ldots b_n\}$ and the (intermediate) tuples $\{t_1, \ldots t_m\}$, respectively. A corollary of Lemma 3 is that, unless joins are block-structured, the result tuple contains a P_4. It is possible to check if a join is block-structured in linear time. If it is, then we call $\mathcal{T}$ on each block T_i thus producing one co-tree per block which usually results in a smaller number of co-trees.

Another possible optimization lies in the merging process itself. One possible option is to initialize an empty tree and merge each co-tree, produced out of each block of base and intermediate tuples, into it (**sequential merge**). Another option is to arrange the per-block co-trees into a queue and then dequeue a pair of co-trees, merging them and enqueueing the result, in turn (**pairwise merge**). It is possible to trace the pairwise merging process by using a tree where the leaves form the co-trees produced by the join rule and

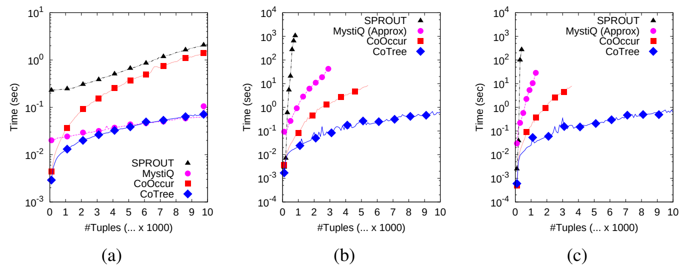


Figure 5: Results of evaluating (a) two, (b) three and (c) four relation Boolean conjunctive queries on synthetic databases generated with $d = 150, \dots 250$. Note that, y-axis is log-scale.

the intermediate nodes represent co-trees produced by merging a pair from the queue. Empirically, we noticed pairwise merge to be significantly faster than sequential merge.

A third optimization is possible in the implementation of $\oplus$. Recall from Figure 4 that, in various rules we check if sub-trees contain common variables before calling $\mathcal{S}$ on them (in Rule 4, for instance). In our implementation, we never call $\mathcal{S}$ on two sub-trees unless they contain the same exact number of common variables. This does not in any way affect the correctness of our algorithm since a prerequisite of a successful alignment of two co-trees is that they contain the same number of common variables. Instead, the count of common variables acts as a much more stringent filter, cutting down on the number of calls made to $\mathcal{S}$. An empirical check to determine the effectiveness of this filter showed that in general, each call to $\oplus$ to merge two co-trees of sizes n and m nodes gave rise to $\min(n, m)$ recursive calls which represents a favourable case (much smaller than $O(nm)$).

4. EXPERIMENTS

Our aim is to evaluate how our approach compares with state-of-the-art probabilistic databases with respect to query evaluation time and whether there is any merit to the paradigm of per-result tuple evaluation. To demonstrate scalability, we experiment with synthetic databases of varying sizes. We show that, not only do we far outperform current probabilistic databases when evaluating non-hierarchical queries, but that our techniques also compare favourably to query compilation techniques specifically developed to handle hierarchical queries. We also ran experiments using the TPC-H benchmark [28]. Five of the first TPC-H queries are not hierarchical. Of these, one query's result solely comprised of read-once functions and two others generated $\approx 50\%$ read-once result tuples thus lending credence to the paradigms of database instance-optimal and per-result tuple query evaluation.

We compare our approach (henceforth referred to as **CoTree**) against two state-of-the-art probabilistic databases, **MystiQ** [4] and **SPROUT** [24], and the naive approach (Section 2.3) referred to as **CoOccur** using a prior P_4 checking algorithm [7]. MystiQ contains an implementation of Dalvi et al. [9]'s hierarchical query recognition/safe plan generation technique. For SPROUT, we only consider the `conf()` function for comparison since we are mainly interested in exact confidence computation.

To keep our experimental methodology simple, we only experiment with Boolean conjunctive queries (final set of projected attributes is empty) and hence do not perform the normality check for CoOccur. To generate tuples for the synthetic dataset, we take a domain size d as a parameter and randomly generate attribute values from that domain. For TPC-H queries, we use the *dbgen* program [28] to generate the database. Tuple probabilities are picked randomly. Our code is written in JAVA. All experiments were run on an Intel 2.26GHz Core 2 Duo machine with 2GB RAM.

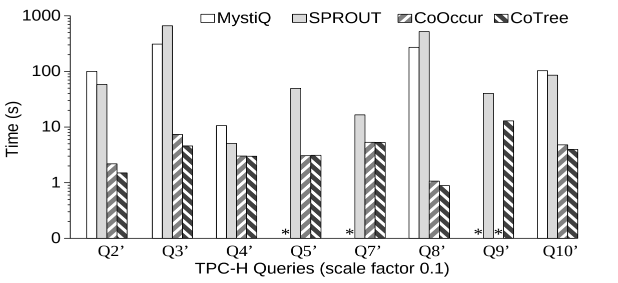


Figure 6: Results on the TPC-H benchmark. "*" indicates the approach ran out of memory. Note that, y-axis is log-scale.

Synthetic Dataset: Figures 5(a), (b) and (c) depict the results of running the four approaches on two, three and four relation Boolean join queries over synthetic databases generated with d varying from $150, \dots 250$. All three plots report runtimes (in seconds) on the y-axis averaged over ten different queries and number of tuples per relation (in thousands) along the x-axis. Partial plots indicate the approach ran out of memory or took too long to run. For the case of two relation joins (Figure 5 (a)), the queries are guaranteed to be hierarchical [11] so MystiQ does very well. The interesting thing in this case is that CoTree matches MystiQ's performance toe-to-toe. CoOccur's performance deteriorates drastically with larger databases because the result tuples produced have larger and denser co-occurrence graphs. Perhaps surprisingly, SPROUT performs worst even though it has specialized algorithms to handle hierarchical queries. The likely reason[‡] for this is that SPROUT prefers plans which defer marginal probability computation till the end (*lazy* plans [24]) as opposed to MystiQ, which computes marginals as soon as an (intermediate) tuple is produced (*eager* plans [24]). In this case, the eager plan works better than the lazy plan. Note that, SPROUT is capable of running both lazy and eager plans. For three and four relation joins, Figures 5 (b) and (c) respectively, it is no longer necessary that the queries be hierarchical. However, it is possible to extend the idea behind the running example (Figure 1 (a)) to larger relations and larger number of joins to generate read-once result tuples. For these queries, MystiQ immediately switches to its approximate inference method (a slow MCMC technique). SPROUT fares the worst in these cases also (SPROUT is grazing the y-axis which is in log-scale). CoTree performs the best, providing performance that is orders of magnitude faster than any of the other approaches. Varying d alters performance in expected ways. Upon reducing d (thus increasing join selectivity), the co-occurrence graphs of the result tuples become denser thus deteriorating performance of SPROUT, MystiQ and CoOccur even further, while increasing d reduces the difference among the approaches.

TPC-H: We also experimented with queries Q1 through Q10 in the TPC-H specification on a 0.1GB database. Since we do not consider running aggregation nor self-joins, we modified the queries slightly. We added random probabilities to all TPC-H relations. This leads to Q2, Q5, Q7, Q8 and Q9 being non-hierarchical. Interestingly, despite the queries being non-hierarchical, all of Q2's result tuples turned out to be read-once, for which SPROUT, CoOccur and CoTree were able to compute marginals easily. Moreover, Q8 and Q9's result comprise 52% and 43.4% read-once result tuples, respectively. This illustrates the merit of evaluating on a per-result tuple basis. However, all of these queries still present fairly easy marginal probability computation problems (no query spent more than 15% of the time computing probabilities). Thus, we picked the queries with more than 1 joining relation and modified them even further by dropping attributes from the SELECT clause and/or predicates from the WHERE clause all the while ensuring

[‡]Dan Olteanu, personal communication.

that all generated result tuples were read-once. Figure 6 denotes the performance of the four approaches on these modified queries where we use bars to denote run-time and Qi' denotes the modified query obtained from TPC-H Query Qi. In Figure 6, except for Q4' (a two relation join query), all other queries were non-hierarchical. MystiQ ran out of memory while running Q5' and Q7', and both MystiQ and CoOccur ran out of memory while running Q9'. The missing bars are denoted by "*". CoTree performs best on all the queries, with CoOccur coming a close second. For five of these queries, CoTree outperforms SPROUT by at least an order of magnitude and on two, CoTree is two orders of maginitude faster.

Discussion: Besides the experiments described above, we extensively evaluated our co-tree merging algorithm and our pairwise approach to merging multiple co-trees. We also conducted an experiment to check if we are repeating work while factorizing each result tuple in isolation by possibly rediscovering the same query plan. Sometimes, by running the correct query plan, the lineage-tree of the result tuple itself turns out to be its co-tree (this is the main idea behind MystiQ [9]). To check for this, we took the three relation join Boolean query (Figure 5 (b)) and ran it with all possible query plans (various early projections, various join orders). For each query plan, we measured the hardness of computing the result tuple's marginal probability from the generated lineage-tree by measuring *treewidth* [2]. If a query plan recovered the co-tree then the treewidth would be 1. Invariably, the query plans produced very hard inference problems with treewidth sometimes approaching 250. This implies that to exactly evaluate factorizable result tuples, only employing query rewriting techniques is insufficient and our approach of considering each result tuple separately is justified.

5. RELATED WORK

Olteanu and Huang [22, 23] showed that hierarchical queries can be extended to include inequality operators, $\neq$, $>$, $<$, in certain cases. These queries may not produce read-once functions. Dalvi et al. [8] propose an approach to efficiently handle disjunctive queries with self-joins. Jha et al. [19] consider unifying hierarchical queries with graphical models-based query evaluation [27]; these techniques however, do not cover the class of read-once functions. Olteanu et al. [25] use a generalized version of co-trees known as *d-trees* but require input in DNF to construct them. Darwiche [12] proposes using Boolean formula factorization algorithms to compile a probabilistic model into a more tractable form. However, he relies on the use of an exponential-sized intermediate representation called *multi-linear formula*. Our work on merging co-trees is related to tree alignment [20] which comes in two flavours, ordered and unordered. Ordered trees have an ordering defined over the childred as opposed to unordered trees. Unordered tree alignment is MAX SNP-hard [20]. So it is noteworthy that, even though our co-trees contain unordered nodes (⓪ nodes), we can still merge them in polynomial time.

6. CONCLUSION

In summary, we considered the problem of efficiently evaluating queries over tuple-level uncertainty probabilistic databases. Earlier approaches have mainly concentrated on query-centric notions of solvability (hierarchical queries). In this paper, we went beyond just looking at the query to decide whether it is PTIME-solvable or not. We first showed how to incorporate Boolean formula factorization techniques into the query engine that exploit structure present in both query and data to efficiently evaluate result tuples' marginal probabilities. We also proposed novel, especially efficient algorithms to evaluate a large class of queries, viz. conjunctive queries without self-joins. We empirically showed that our proposed techniques are much faster than prior Boolean formula factorization techniques, techniques specifically developed for hierarchical queries, and generic inference algorithms.

Even though our discussion mainly involved tuple-independent probabilistic databases, the techniques we proposed are likely to be useful for databases with correlated tuples also. In that case, by plugging in the factorized co-tree generated by our algorithms into the graphical model describing the correlations among the base tuples [27], we should be able to obtain a combined graphical model of lower treewidth. As part of our future work, we intend to extend our techniques to query probabilistic databases with tuple and attribute uncertainty. Also, we would like to explore various generalizations such as P4-tidy graphs [14] to see if these can be advantageously used for querying probabilistic databases.

Acknowledgments: We would like to thank the anonymous reviewers for their suggestions and Dan Olteanu for patiently answering all our questions about the SPROUT codebase. This work was supported in part by NSF grants IIS-0546136 and IIS-0916736.

7. REFERENCES

[1] C. C. Aggarwal. *Managing and Mining Uncertain Data*. Springer, 2009.

[2] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. In *BIT*, 1985.

[3] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[4] J. Boulos, N. Dalvi, B. Mandhani, C. Re, S. Mathur, and D. Suciu. Mystiq: A system for finding more answers by using probabilities. In *SIGMOD*, 2005.

[5] A. Bretscher, D. Corneil, M. Habib, and C. Paul. A simple linear time lexbfs cograph recognition algorithm. *SIAM Journal on Discrete Mathematics*, 2008.

[6] D. Corneil, H. Lerchs, and L. Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 1981.

[7] D. Corneil, Y. Perl, and L. Stewart. A linear recognition algorithm for cographs. *SIAM Journal of Computing*, 1985.

[8] N. Dalvi, K. Schnaitter, and D. Suciu. Computing query probability with incidence algebras. In *PODS*, 2010.

[9] N. Dalvi and D. Suciu. Efficient query eval. on prob. databases. In *VLDB*, '04.

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

[11] N. N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *PODS*, 2007.

[12] A. Darwiche. A logical approach to factoring belief networks. In *Intl. Conf. on Principles and Know. Rep. and Reasoning*, 2002.

[13] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. on Info. Systems*, 1997.

[14] V. Giakoumakis, F. Roussel, and H. Thuillier. On P4-tidy graphs. *Discrete Maths and Theoretical Comp. Science*, 1997.

[15] M. 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 Applied Mathematics*, 2006.

[16] M. Habib and C. Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 2005.

[17] J. Hayes. Fanout structure of switching functions. *JACM*, 1975.

[18] 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 *SIGMOD*, 2008.

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

[20] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *Annual Symposium on Combinatorial Pattern Matching*, 1994.

[21] C. Koch. Approximating predicates and expressive queries on probabilistic databases. In *PODS*, 2008.

[22] D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *SUM*, 2008.

[23] D. Olteanu and J. Huang. Secondary-storage conf. computation for conjunctive queries with inequalities. In *SIGMOD*, 2009.

[24] D. Olteanu, J. Huang, and C. Koch. SPROUT: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, 2009.

[25] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, 2010.

[26] C. Ré and D. Suciu. The trichotomy of having queries on a probabilistic database. *VLDB J.*, 18(5):1091–1116, 2009.

[27] P. Sen, A. Deshpande, and L. Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.

[28] TPC-H Benchmark. http://www.tpc.org/.

APPENDIX

A. PROOFS

Theorem 2 crucially depends on the following lemma where $x - y \in G$ denotes variables x and y form an edge in G.

LEMMA 2. *Let C_1, C_2 denote two clauses in ϕ_{DNF} s.t. $C_1 \cap C_2 = \mathbf{z} = \{z_1, \dots z_l\}$, $x \in C_1, x \notin C_2, y \notin C_1, y \in C_2$. If ϕ is P_4-free and $x{-}y \in G_\phi$ then $\exists C_3$ in ϕ_{DNF} s.t. $\{x, y, z_1, \dots z_k\} \subseteq C_3$.*

PROOF. Let q denote the conjunctive query (without self-joins) that produced result tuple ϕ. Notice that $x, y, \mathbf{z}$ form $l + 2$ sized clique in G_ϕ. If $|Rels(q)| = l + 2$ then this clique will form a clause in which case we have found C_3. We will assume this is not the case and $|Rels(q)| > l + 2$. The basic idea is to show that a selection of the remaining variables from C_1 and C_2 along with x, y and $\mathbf{z}$ form a $|Rels(q)|$-sized clique in G_ϕ. This, in turn will imply the presence of the clause containing x, y and $\mathbf{z}$. Let us first complete the cliques C_1 and C_2:

- $C_1 \setminus (\{x\} \cup \mathbf{z}) = \{y', w_1, \dots w_n\}$
- $C_2 \setminus (\{y\} \cup \mathbf{z}) = \{x', v_1, \dots v_n\}$

where $w_i \not\equiv v_i$ and $w_i, v_i \in R_i \in Rels(q)\ \forall i = 1 \dots n$ (re-index $Rels(q)$ if necessary). First, notice that one of the edges $x - v_i$ or $y - w_i$ (or both) has to exist in $G_\phi\ \forall i = 1 \dots n$ if ϕ is to be P_4-free else $w_i - x - y - v_i$ is a P_4 (w_i and v_i cannot form an edge because they belong to the same relation R_i). This gives us a selection procedure: if $x - v_i$ then pick v_i else pick w_i. Further, two variables v_i and w_j ($i \neq j$) selected this way must have an edge between them otherwise ϕ has a $P_4 : w_j - x - v_i - v_j$ (v_j and x do not form an edge otherwise we would have selected v_j not w_j). Thus, $x, y, \mathbf{z}$ along with the selected w's and v's form a $|Rels(q)|$-sized clique in G_ϕ. $\square$

THEOREM 2. Let ϕ denote the result tuple produced by a conjunctive query q without self-joins. If ϕ is P_4-free, it is also normal.

PROOF. To prove the proposition, we need to show that any clique in G_ϕ is contained in some clause in ϕ_{DNF}. Notice that a clique of size greater than $|Rels(q)|$ is not possible since this would imply a self-join in q and a clique of size equal to $|Rels(q)|$ directly translates to a clause in ϕ_{DNF}. Proving the proposition for cliques of size $< |Rels(q)|$ is slightly trickier. We prove this by induction on the size k of the clique. The base case is when $k = 3$ ($k = 2$ is simply an edge which has to be contained in a clause). Let $\{a, b, c\}$ denote the clique. Assume that clause C_1 contains a and b but not c and C_2 contains b and c but not a. (If either C_1 contained c or C_2 contained a then we have found the clause we are looking for and the base case is proved.) Now we invoke Lemma 2 with $x = a, y = c$ and $\mathbf{z} = \{b\}$. Thus, there must exist a clause containing all three variables. For the inductive case, we will assume that any $k - 1$ sized clique in G_ϕ is fully contained in some clause in ϕ_{DNF}. Now we need to show that given a k-sized clique $\{a_1, \dots a_k\}$, a clause containing it exists. Notice that $\{a_1, \dots a_{k-1}\}$ forms a $k - 1$ sized clique and thus there must be a clause C_1 containing it (by inductive hypothesis). Similarly, there must also be a clause C_2 containing $\{a_2, \dots a_k\}$. Notice that if $a_k \in C_1$ or $a_1 \in C_2$ then we have shown what was needed. Assuming that is not true, invoke Lemma 2 with $x = a_1, y = a_k$ and $\mathbf{z} = \{a_2, \dots a_{k-1}\}$. Thus, a clause containing $\{a_1, \dots a_k\}$ must exist. $\square$

LEMMA 1. Let ϕ denote a result tuple, $\mathcal{L}$ its lineage-tree and Γ a join path. If ϕ is P_4-free then the set of cofactors $\{cof_{\mathcal{L}}([\Gamma, b]) | b \in R\}$ is non-interfering.

PROOF. It is straightforward to show that type 1 interference is not possible. Let us assume that $cof_{\mathcal{L}}([\Gamma, b_i])$ and $cof_{\mathcal{L}}([\Gamma, b_j])$ interfere, where $b_i, b_j \in R$. In other words, $\exists x, y$ such that $x - y \in G_{cof_{\mathcal{L}}([\Gamma, b_i])}$ and $x \not- y \in G_{cof_{\mathcal{L}}([\Gamma, b_j])}$ even though $x, y \in Vars(cof_{\mathcal{L}}([\Gamma, b_j]))$. This implies that the DNF of $cof_{\mathcal{L}}([\Gamma, b_j])$ must contain two distinct clauses one of which contains x but not y and the other y but not x. Coupled with the fact that $cof_{\mathcal{L}}([\Gamma, b_j])$ joins with b_j and tuples in join path Γ, this implies that ϕ_{DNF} contains two distinct clauses one of which contains the set $\{\gamma_1, \dots \gamma_k, b_j, x\}$ and the other $\{\gamma_1, \dots \gamma_k, b_j, y\}$, where γ_i denotes a tuple from Γ. Since $x - y \in G_{cof_{\mathcal{L}}([\Gamma, b_i])}$, we can now set $\mathbf{z} = \{\gamma_1, \dots \gamma_k, b_j\}$ and invoke Lemma 2 to claim the presence of a clause in ϕ_{DNF} that contains all of $\gamma_1, \dots \gamma_k, b_j, x, y$. If that is true, then there should be a clause in DNF of $cof_{\mathcal{L}}([\Gamma, b_j])$ that contains both x and y and these two variables should form an edge in $G_{cof_{\mathcal{L}}([\Gamma, b_j])}$ which is not true. Hence we have a contradiction.

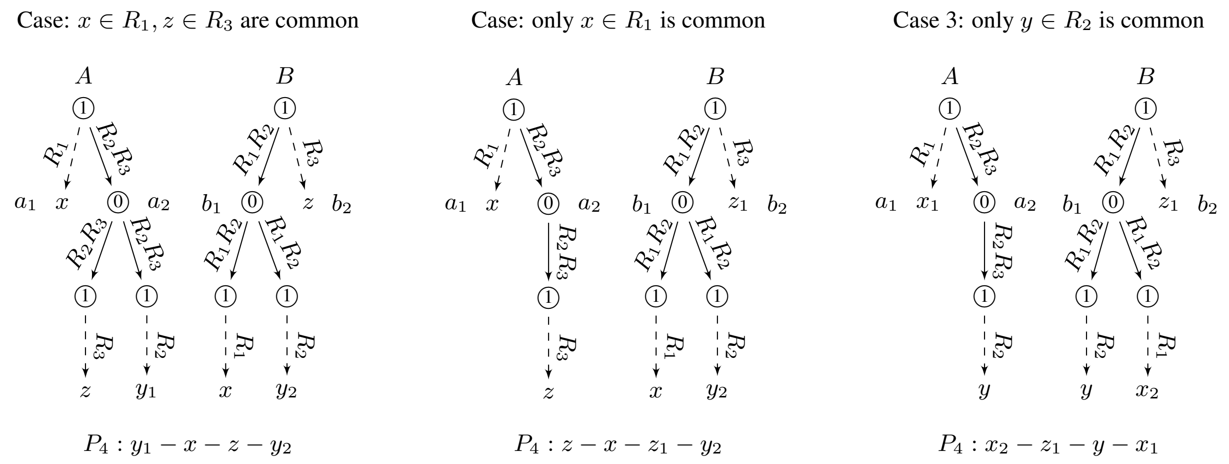
For type 2 interference, assume that $\exists x, y$ such that $x - y \in G_{cof_{\mathcal{L}}([\Gamma, b_h])}, x \in Vars(cof_{\mathcal{L}}([\Gamma, b_i])), x \notin Vars(cof_{\mathcal{L}}([\Gamma, b_j]))$, $y \notin Vars(cof_{\mathcal{L}}([\Gamma, b_i]))$ and $y \in Vars(cof_{\mathcal{L}}([\Gamma, b_j]))$, where b_h, b_i, b_j belong to the same relation R. Note that, this gives us a P_4 in $\phi : b_i - x - y - b_j$, unless $b_i - y$ or $x - b_j$ exists in G_ϕ (b_i and b_j belong to the same relation so they cannot form an edge). Without loss of generality, let us assume $b_i - y$ exists in G_ϕ so we do not have the P_4. Clearly, this edge is not present in $G_{cof_{\mathcal{L}}([\Gamma, b_i])}$ (since $cof_{\mathcal{L}}([\Gamma, b_i])$ does not contain y) so it has to come from some other part of $\mathcal{L}$. We now backtrack along $\Gamma = \gamma_1, \dots \gamma_k$ in the reverse direction. Let $k' \leq k$ denote the maximum index such that $\exists \gamma'_{k'} \neq \gamma_{k'}$ and $b_i - y \in G_{cof_{\mathcal{L}}([\gamma_1, \dots \gamma_{k'-1}, \gamma'_{k'}])}$. Note that, $cof_{\mathcal{L}}([\gamma_1, \dots \gamma_k, b_i])$ is a subformula of $cof_{\mathcal{L}}([\gamma_1, \dots \gamma_{k'-1}, \gamma_{k'}])$ (due to the recursive nature of cofactors). Let Γ' denote the join path $\gamma_1, \dots \gamma_{k'-1}$ then this implies $cof_{\mathcal{L}}(\Gamma', \gamma_{k'})$, which does not contain an edge between y and b_i, and $cof_{\mathcal{L}}(\Gamma', \gamma'_{k'})$, which does, exhibit type 1 interference. As we have already seen, this is not possible. Hence we have a contradiction. $\square$

LEMMA 3 (BLOCK-STRUCTURED CO-FACTORS). *Let Γ denote a join path in lineage-tree $\mathcal{L}$ of result tuple ϕ. If ϕ is P_4-free then $cof_{\mathcal{L}}([\Gamma]) = b_1 cof_{\mathcal{L}}([\Gamma, b_1]) + \ldots + b_n cof_{\mathcal{L}}([\Gamma, b_n])$, where b_i denotes a base tuple, is block-structured. In other words, if $\exists$ (intermediate) tuples t_i, t_j such that $t_i \in cof_{\mathcal{L}}([\Gamma, b_i])$, $t_i \in cof_{\mathcal{L}}([\Gamma, b_j])$ but $t_j \in cof_{\mathcal{L}}([\Gamma, b_i])$, $t_j \notin cof_{\mathcal{L}}([\Gamma, b_j])$ then ϕ has a P_4.*

PROOF. Notice that, since b_j joins with t_i, G_ϕ contains an edge between b_j and any variable present in the formula of t_i. Also, for a variable x that is present in t_j but not in t_i we have a P_4 in ϕ if $b_j \not- x \in G_\phi$ (the P_4 is $b_j - x' - b_i - x$, where x' belongs to the same relation as x and is present in t_i). Thus, unless b_j is connected to all variables in t_j in G_ϕ, ϕ has a P_4 and there is nothing left to prove. Now, we go back to $cof_{\mathcal{L}}(\Gamma)$. Since Γ joins with b_j and Γ joins with any clause in DNF of t_j, we can apply Lemma 2 repeatedly to show that b_j joins with t_j obtaining a contradiction. $\square$

B. LEMMAS FOR MERGING ALGORITHM

Recall from Section 3.2 that, if two cofactors interfere then the result tuple has a P_4. Also recall that, in the description of the $\oplus$ operator (Section 3.3) we check to see if the returned score matches the number of common variables. This check is a crucial step because it helps detects P_4s in the result tuple. It is possible to show (by tracing $\mathcal{S}$) that if T_1 and T_2 (co-trees being merged) are such that $x - y \in G_{\phi_{T_1}}$ and $x, y \in Vars(T_2)$ but $x \not- y \in G_{\phi_{T_2}}$ then the returned score will always be less than the number of variables



Case: $x \in R_1, z \in R_3$ are common
Case: only $x \in R_1$ is common
Case 3: only $y \in R_2$ is common
$P_4 : y_1 - x - z - y_2$
$P_4 : z - x - z_1 - y_2$
$P_4 : x_2 - z_1 - y - x_1$

Figure 8: When merging two co-trees rooted at ① nodes, the child sub-trees must be divisible into sets that cater to the same relations, else the merging has a P_4. Moreover, at least one such subset should be singleton.

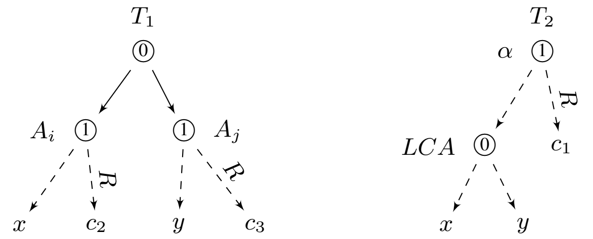


Figure 7: Separated common nodes indicate a P_4.

common to T_1 and T_2. A simple intuitive explanation is by considering least common ancestors (LCA) of the two variables in the co-trees T_1 and T_2. One of the basic properties of a co-tree is that given a pair of variables, their ancestor is a ① node if the variables form an edge and a ⓪ node, if not [15]. This implies that the LCA of x and y in T_1, LCA_1, is a ① node and their LCA in T_2, LCA_2, is a ⓪ node. Now, as per merging rules 3 and 4 (Figure 4), we only match one rooted subtree against another. This implies that whenever we match the subtrees rooted at LCA_1 and LCA_2 we are bound to misalign at least one of either x or y thus detecting that the result tuple contains a P_4. In what follows, we prove three lemmas required to justify our algorithm for $\oplus(T_1, T_2)$ presented in Section 3.3. In none of these proofs do we consider cases leading to edge discrepancies where two variables x, y form an edge in T_1 (T_2) and do not form an edge in T_2 (T_1) because, as we just showed, these cases lead to P_4s and by comparing the returned score we can easily capture such cases.

LEMMA 4. *Let $T_1 = ⓪(A_1, \dots A_n)$ such that $x \in Vars(A_i)$ and $y \in Vars(A_j)$, where $i \neq j$. Let T_2 denote another co-tree rooted at ① node such that $x, y \in Vars(T_2)$. If T_1 and T_2 involve variables from the same set of relations then $\phi_{T_1} \vee \phi_{T_2}$ contains a P_4.*

PROOF. If LCA of x and y in T_2 is a ① node then we have an edge discrepancy between T_1 and T_2 which implies interference and a P_4. Assuming LCA of x and y in T_2 is a ⓪ node, Figure 7 shows the setup described in the lemma where LCA denotes LCA of x and y in T_2 and α denotes root of T_2. Dashed edges indicate ancestor-child (not necessarily parent-child) relationships. Notice that, besides LCA, α must have at least one other descendant (otherwise there is no point of having α). Let c_1 denote one such descendant leaf and let R denote the relation c_1 belongs to. Now, the merging rules (Figure 4) ensure that sub-trees being aligned involve variables from the same relations. This implies that variables from R are present in both child sub-trees of A. Notice that c_1's presence in either of these sub-trees will lead to edge discrepancies that would indicate the presence of a P_4 and there would be nothing left to prove. Thus, let c_2 ($\not\equiv c_1$) denote the variable from R that is present in the child sub-tree containing x, and let c_3 ($\not\equiv c_1$) denote the variable from R that is present in the child sub-tree containing y. Again, neither c_2 nor c_3 can be present in B because if they were then we would have edge discrepancies indicating a P_4 and there would be nothing left to prove. Now, we have a $P_5 : c_2 - x - c_1 - y - c_3$ that contains two P_4's. □

The next two lemmas relate to rule 5 in Figure 4. The first one shows that when aligning two co-trees rooted at ① nodes, the child sub-trees need to be such that we can divide them into minimal subsets such that each pair of subsets involves variables from the same (sub)set of relations (see Section 3.3). This will not be possible if there exists relations R_1, R_2, R_3, two child sub-trees in the first co-tree rooted at a_1, a_2 and two child sub-trees in the second co-tree rooted at b_1, b_2 such that $R_1 \in \mathcal{A}(a \to a_1), \{R_2, R_3\} \subseteq \mathcal{A}(a \to a_2), \{R_1, R_2\} \subseteq \mathcal{A}(b \to b_1), R_3 \in \mathcal{A}(b \to b_2)$ where a and b denote the roots of the two input co-trees, respectively. The proof of the lemma considers various cases that satisfy this property and shows that a P_4 exists in each case. The second lemma deals with the placement of variables that are not present in both co-trees.

LEMMA 5. *Let a and b denote two ① roots of two co-trees A and B, respectively. Let a_1, a_2 and b_1, b_2 denote two children of a and b, respectively, such that there is at least one common variable present in the sub-trees rooted at a_1, a_2 and b_1, b_2. If $\exists R_1, R_2, R_3$ s.t. $R_1 \in \mathcal{A}(a \to a_1)$, $\{R_2, R_3\} \subseteq \mathcal{A}(a \to a_2)$, $\{R_1, R_2\} \subseteq \mathcal{A}(b \to b_1)$, $R_3 \in \mathcal{A}(b \to b_2)$ then the merging of A and B contains a P_4.*

PROOF. The assumption of sub-trees rooted at a_1, a_2 and b_1, b_2 having at least one common variable is important because if this is not the case then we have work arounds. For instance, if the input co-trees rooted at a and b do not have any variable in common then we apply merging rule 2 not 5 (Figure 4). Figure 8 explains the proof. In Figure 8, x, x_1, x_2 denote variables from R_1, y, y_1, y_2 de-

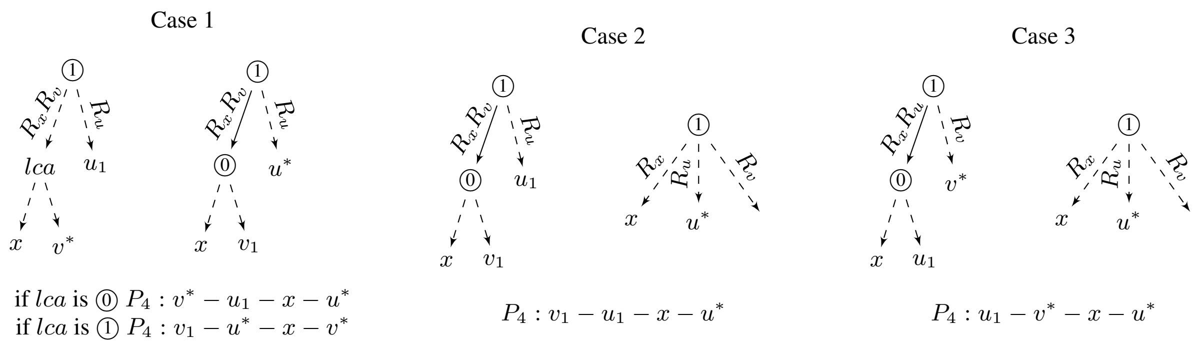

Case 1
Case 2
Case 3
①
R_xR_v
R_u
lca
u_1
x
v^*
⓪
u^*
v_1
u_1
R_x
R_u
R_v
v^*
if lca is ⓪ $P_4 : v^* - u_1 - x - u^*$
if lca is ① $P_4 : v_1 - u^* - x - v^*$
$P_4 : v_1 - u_1 - x - u^*$
$P_4 : u_1 - v^* - x - u^*$

Figure 9: Proof showing that merging co-trees after application of rule 5 in Figure 4 is simple. If the presence of new variables complicate things then a P_4 exists in the merge result.

note variables from R_2 and z, z_1 denote variables from R_3. There are three cases depicted in Figure 8:

- In the first case, we assume that x and z are common (y may or may not be common), i.e., x is present in both sub-trees rooted at a_1 and b_1, and z is present in both sub-trees rooted at a_2 and b_2. Note that in this case, if y_1 is present under b_1 or y_2 under a_2 then we will introduce an edge discrepancy, in which case we know there is a P_4 and there is nothing left to prove. The figure depicts the P_4 when y_1 exists only under a_2 and y_2 exists only under b_1.
- In the second case, we assume only one of x or z are common. Since both cases are symmetric, we assume x is common. Notice that in this case, a common variable y can exist in both sub-trees rooted at a_2 and b_1 but this does not contradict the existence of y_2 which cannot exist in the sub-tree rooted at a_2 without introducing an edge discrepancy. The figure depicts the P_4 obtained.
- In the last case, we assume y is the only variable common to both co-trees. If either x or z or both are also common then we fall under the previous cases. Since there are not common variables x or z present in sub-trees rooted at a_1, a_2 and b_1, b_2, x_1, x_2, z_1 must exist such that x_1 is not present in the sub-tree rooted at b_1, x_2 does not exist in the sub-tree rooted at a_1 and z_1 does not exist in the sub-tree rooted at a_2. This gives us a P_4 (see Figure 8).

□

The second lemma which deals with merging rule 5 in Figure 4, has implications on how we actually merge co-trees formed by the various recursive calls once they have returned. We first illustrate an example that explains the kind of problem we might run into had the following lemma not been true. Assume that $\{x_1, x_2\}$, $\{v_1, v_2\}$ and $\{u_1, u_2\}$ belong to the same relations. Now, consider merging ①(⓪(①(x_1, v_1), ①(x_2, v_2)), u_1) (or $x_1v_1u_1 + x_2v_2u_1$) and ①(x_1, u_2, v_1) (or $x_1u_2v_1$). When we apply $\mathcal{S}$ on these two, we will need to apply rule 5. Merging rule 5 will then make two recursive calls $\mathcal{S}$(⓪(①(x_1, v_1), ①(x_2, v_2)), ①(x_1, v_1)) and $\mathcal{S}(u_1, u_2)$. Assume that the co-trees returned, ⓪(①(x_1, v_1), ①(x_2, v_2)) and ⓪(u_1, u_2), are correctly formed. Thus, we now need only worry about putting these together. The simple thing that we do is to put a ① node as root and make the roots of the co-trees returned by the recursive calls its children. But in the case of this example this would not work. This is because in doing so, we would introduce an edge between v_2 and u_2 (among other spurious edges that will also get introduced). Taking a closer look at the correct merge result, $x_1v_1u_1 + x_2v_2u_1 + x_1u_2v_1$, we should notice that it contains a $P_4 : v_2 - u_1 - x_1 - u_2$ (among others). This kind of a tricky merge situation occurs everytime we produce two separate recursive calls one of which involves variables from the first co-tree not present in the second and the other containing variables from the second co-tree not present in the first (e.g., v_2 and u_2, respectively, in the example). In the case of this example, as it turned out we were lucky and the result was not read-once. The question is, is this always the case? The next lemma answers this question in the affirmative.

LEMMA 6. *Let application of merging rule 5 (Figure 4) produce two recursive calls $\mathcal{S}(T_1^A, T_1^B)$ and $\mathcal{S}(T_2^A, T_2^B)$, where A and B denote the input co-trees, T_1^A, T_2^A minimal subsets of sub-trees formed from A and T_1^B, T_2^B minimal subsets formed from B. If $\mathcal{N}(T_1^A) \wedge \mathcal{N}(T_2^B) \vee \mathcal{N}(T_2^A) \wedge \mathcal{N}(T_1^B)$ is true then the merge result contains a P_4.*

PROOF. Just as in the previous proof of Lemma 5, we will assume that A and B share at least one common variable (otherwise we will not apply merging rule 5 but rule 2 instead). Moreover, if at least one common node is not present in T_1^A, T_2^A and T_1^B, T_2^B then we still have work arounds that will not allow the $\mathcal{S}$ calls stated in the lemma to occur. We will denote by x the common variable and by R_x the relation it belongs to. Let v^* and u^* denote the new variables from A and B, respectively, such that v^* is not present in B and u^* is not present in A. Moreover, in at least one co-tree, either A or B, the child sub-tree containing the new variable must contain a common variable in it. Otherwise, we have work arounds that do not let the $\mathcal{S}$ calls stated in the lemma to occur. WLOG, we will assume that the new variable present with the common variable in the same sub-tree is v^* (present in co-tree A) and if the common variable is not x then rename it. Now, we have three cases, they are shown in Figure 9. In Figure 9, v_1 and u_1 belong to relations R_v and R_u, respectively, where R_u contains u^* and R_v contains v^*. The first case contains two subcases, depending on whether the LCA of x and v^* is a ⓪ or ① node. In either case, we find a P_4. Also, note that in the first case when LCA is a ⓪ node, v^* could belong to R_x so there is no separate R_v but the stated P_4 still exists. The rest of the figure should be self-explanatory. □

<u>Time Complexity</u>: Recall that in Section 3.3, we showed that for the simple case when no new dummy ① nodes are introduced by rule 5 in Figure 4, $\oplus(T_1, T_2)$ incurs a time complexity of $O(nmk^2)$, where n and m denote the number of nodes in T_1 and T_2, respectively, and k denotes the total number of relations. For the more

R_1:

	A
a_1	1
a_2	2

R_2:

	B
b_1	1
b_2	2

R_3:

	A
x_1	1
x_2	2

R_4:

	B
y_1	1
y_2	2

$$q\text{:-} \prod_{\emptyset}\{(R_1(\mathbf{A}) \times R_2(\mathbf{B})) \bowtie_{\mathbf{A},\mathbf{B}} (R_3(\mathbf{A}) \times R_4(\mathbf{B}))\}$$

$$\begin{aligned} r &= (a_1b_1)(x_1y_1) + (a_1b_2)(x_1y_2) + (a_2b_1)(x_2y_1) + (a_2b_2)(x_2y_2) \\ &= (a_1x_1 + a_2x_2)(b_1y_1 + b_2y_2) \end{aligned}$$

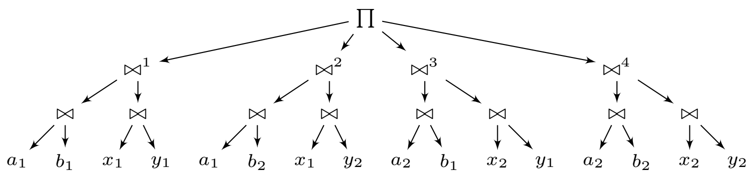


Figure 10: Query plan with a bushy join. Finding non-interfering sub-formulas is an issue.

general case when dummy nodes are introduced, we first note that two dummy nodes generated from, say, T_1 introduced by an invocation of rule 5 never get matched with the same node in T_2. Also, due to Lemma 5, dummy nodes always get matched against real nodes that were present in the original co-tree. This implies that extra work incurred per ① node in T_1 is at most $O(mk^2)$, which implies the total extra work done due to dummy nodes cannot exceed $O(nmk^2)$. So, the total time complexity remains $O(nmk^2)$.

C. PLANS WITH BUSHY JOINS

Plans with bushy joins can also be handled but they require slightly more work. Let us first consider an example that helps illustrate the issues in this case. Figure 10 shows how a plan with a bushy join between $R_1 \times R_2$ and $R_3 \times R_4$ can cause complications. More specifically, the subtree in the lineage-tree rooted at $\bowtie^2$ contains variable y_2 but not x_2, the subtree rooted at $\bowtie^3$ contains x_2 but not y_2 whereas the subtree rooted at $\bowtie^4$ contains both variables with an edge between them in the corresponding co-occurrence graph (type 2 interference). Another way of stating the issue is to notice that, assuming we manage to construct co-trees for subtrees rooted at $\bowtie^1$, $\bowtie^2$, $\bowtie^3$ and $\bowtie^4$, merging any three of these leads to a P_4 (e.g., merging graphs of $\bowtie^1$, $\bowtie^2$ and $\bowtie^3$ would give us P_4 $y_2 - x_1 - y_1 - x_2$) but the result tuple r in Figure 10 is read-once.

We now describe a technique to handle bushy joins. Let $\mathcal{L}$ denote the lineage-tree of result tuple ϕ. Let $n_\bowtie$ denote an internal node in lineage-tree $\mathcal{L}$ that represents a bushy join (e.g., $\bowtie^1$). Moreover, let $n_{\bowtie\leftarrow}$ and $n_{\bowtie\rightarrow}$ represent $n_\bowtie$'s left and right child respectively. Let us consider computing $\mathcal{T}(n_{\bowtie,1}, \ldots n_{\bowtie,m})$ where each of $n_{\bowtie,i}$ represents a node in $\mathcal{L}$ representing a tuple produced by a bushy join. Essentially, in $\mathcal{L}$ produced by plans with bushy joins interference can occur but we will try to come up with an ordering over the co-tree merging operations that circumvents this, i.e., does not detect a P_4 unless the result tuple has one. First consider type 1 interference. Suppose $n_{\bowtie,i\leftarrow}$ contains variable $a \in R$. Repeating arguments presented in Lemma 1 we can show that if $n_{\bowtie,i\rightarrow}$ is such that it contains variables x, y such that $x - y \in G_\phi$ but $x \not- y$ in $n_{\bowtie,i\rightarrow}$ then there must exist another node $n_{\bowtie,j}$ such that a is a leaf in $n_{\bowtie,j\leftarrow}$ and x, y form an edge in $n_{\bowtie,j\rightarrow}$. Thus, if we do not separate the tuples represented by $n_{\bowtie,i}$ and $n_{\bowtie,j}$ then the type 1 interference can never cause a problem because the potential P_4 and the chord are always present together. Again, by repeating arguments presented in Lemma 1, if we avoid type 1 interference then we avoid type 2 interference also. The complete ordering procedure is as follows. Given set of tuples $N = \{n_{\bowtie,1}, \ldots n_{\bowtie,m}\}$ to be merged and formed a co-tree for, first pick two relations $R_\leftarrow$ and $R_\rightarrow$ such that base tuples from $R_\leftarrow$ and $R_\rightarrow$ form leaves in N's left children and right children, respectively. Now, let $N_{a,b} \subseteq N$ represent the set of nodes such that $\forall n_{\bowtie,i} \in N_{a,b}$ $a \in R_\leftarrow$ is a leaf in $n_{\bowtie,i\leftarrow}$ and $b \in R_\rightarrow$ forms a leaf in $n_{\bowtie,i\rightarrow}$. The idea is to take all such pairs a, b and form co-trees for $N_{a,b}$ separately. Subsequently, we merge the various co-trees obtained by merging co-trees for all $N_{a,b}$ with the same a first. To form the co-tree for some $N_{a,b}$ we recurse, i.e., we pick another pair of relations $R'_\leftarrow$ and $R'_\rightarrow$ different from $R_\leftarrow$ and $R_\rightarrow$ and repeat the whole procedure. This merge operation ordering approach is a simple way to extend the approach for plans with left/right deep joins presented in the main body of the paper to the case of plans involving bushy joins. In the case of bushy joins, we no longer have join paths easily determined from the lineage-tree but need to construct them ourselves which we do through recursion.

X:

	$\mathbf{A_1}$	$\mathbf{B_1}$
x_1	3	4
x_2	1	2

Y:

	$\mathbf{A_2}$
y_1	1
y_2	3

Z:

	$\mathbf{B_2}$
z_1	2
z_2	4

$$q() \text{:-} X(\mathbf{A_1}, \mathbf{B_1}), Y(\mathbf{A_2}), Z(\mathbf{B_2}), A_1 = A_2 \vee B_1 = B_2$$

$$\begin{aligned} r = \;& x_1y_1z_2 + x_1y_2z_1 + x_1y_2z_2 + x_2y_1z_1 + x_2y_1z_2 \\ & + x_2y_2z_1 \end{aligned}$$

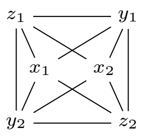


Figure 11: A disjunctive query producing a result tuple that is not normal.

Going back to the example presented Figure 10, one viable ordering is determined by setting $R_\leftarrow = R_1$ and $R_\rightarrow = R_3$. Now we have, $N_{a_1,x_1} = \{\bowtie^1, \bowtie^2\}$ whose co-tree is easily formed, let's call this T_{a_1,x_1}, and $N_{a_2,x_2} = \{\bowtie^3, \bowtie^4\}$ whose co-tree we will denote by T_{a_2,x_2}. Finally, we merge T_{a_1,x_1} and T_{a_2,x_2}. Another ordering that would also work (besides others) is to set $R_\leftarrow = R_1$ and $R_\rightarrow = R_4$, in which case, $N_{a_1,y_1} = \{\bowtie^1\}$, $N_{a_1,y_2} = \{\bowtie^2\}$, $N_{a_2,y_1} = \{\bowtie^3\}$, $N_{a_2,y_2} = \{\bowtie^4\}$. Let $T_{a,y}$ denote the co-tree obtained from $N_{a,y}$. We then perform the remaining merges in the following order: $(T_{a_1,y_1} \oplus T_{a_1,y_2}) \oplus (T_{a_2,y_1} \oplus T_{a_2,y_2})$. Note that, had we chosen to perform $(T_{a_1,y_1} \oplus T_{a_2,y_1}) \oplus (T_{a_1,y_2} \oplus T_{a_2,y_2})$ instead, then that would have also worked.

D. OPEN QUESTIONS

Having considered the case of conjunctive queries in depth, the next obvious question is whether our techniques extend to larger classes of queries. Consider the case of queries with disjunctions, some disjunctions can be allowed without breaking any of our results. But if the query contains a disjunctive join predicate that involves attributes from more than two relations and cannot be broken down into smaller conjunctive predicates then our techniques do not apply. Figure 11 shows one such case with a disjunctive join predicate that leads to a result tuple which is P_4-free but not normal (e.g., x_1, y_1, z_1 is a clique in G_{ϕ_r} but no single clause contains all three variables). Of course, the case for queries with self-joins also remains open.