Lineage Processing over Correlated Probabilistic Databases

Bhargav Kanagal
bhargav@cs.umd.edu

Amol Deshpande
amol@cs.umd.edu

Dept. of Computer Science, University of Maryland, College Park MD 20742

ABSTRACT

In this paper, we address the problem of scalably evaluating conjunctive queries over correlated probabilistic databases containing tuple or attribute uncertainties. Like previous work, we adopt a two-phase approach where we first compute *lineages* of the output tuples, and then compute the probabilities of the lineage formulas. However unlike previous work, we allow for arbitrary and complex correlations to be present in the data, captured via a forest of *junction trees*. We observe that evaluating even read-once (tree structured) lineages (e.g., those generated by *hierarchical* conjunctive queries), polynomially computable over tuple independent probabilistic databases, is #P-complete for lightly correlated probabilistic databases like *Markov sequences*. We characterize the complexity of exact computation of the probability of the lineage formula on a correlated database using a parameter called *lwidth* (analogous to the notion of *treewidth*). For lineages that result in low lwidth, we compute exact probabilities using a novel message passing algorithm, and for lineages that induce large lwidths, we develop approximate Monte Carlo algorithms to estimate the result probabilities. We scale our algorithms to very large correlated probabilistic databases using the previously proposed INDSEP data structure. To mitigate the complexity of lineage evaluation, we develop optimization techniques to process a batch of lineages by sharing computation across formulas, and to exploit any independence relationships that may exist in the data. Our experimental study illustrates the benefits of using our algorithms for processing lineage formulas over correlated probabilistic databases.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Management, Performance

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

Keywords

Conjunctive Queries, Indexing, Junction Trees, Lineage, Probabilistic Databases

1. INTRODUCTION

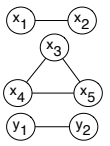
Large amounts of correlated probabilistic data are being generated at a rapidly increasing pace in a wide variety of application domains, including data integration [9], information extraction [15, 13], RFID/sensor network applications [24, 16] and other applications which use machine learning techniques [23] for reasoning over large datasets [7]. Hence, there is a need for systems that can effectively store and query such correlated uncertain data. Although there has been much work in recent years on uncertain data management, most of that work has either restricted the types of correlations that can be represented, or limited the type of queries that can be evaluated efficiently. Further, the increasing scale of such data has led to many challenges that have not been addressed before. We begin with two motivating applications to illustrate the key challenges that we address in this paper.

Information Extraction/Integration: Consider an information extraction/integration system [13, 15, 21, 9] that scans used car advertisements from multiple different sources, such as `cars.com`, `craigslist.com`, and `autotrader.com` and populates a relational database with structured entities (Figure 1). To cope with the enormous amounts of data on the web, the system employs automatic extractors to detect potential tuples. Since most web data is in natural language format, incorrect tuples might also be extracted; hence machine learning algorithms based on CRFs [13] and Bayesian inference techniques [15] are used to assign probabilities of correctness/existence to the extracted tuples.

Figure 1 shows three relations extracted automatically from the web, along with their probabilities. Such data may exhibit significant correlations; e.g., both tuples x_1 and x_2 cannot belong to the relation simultaneously, since they correspond to the same car (since the VIN is same), but their prices/sellers are different. The system models this by including a mutual exclusion correlation between the tuples x_1 and x_2. We may denote such correlations using a graph (Figure 1(b)). Note that x_1 is connected to x_2 indicating that they are correlated. Additional correlations occur due to *attribute uncertainty*. If the attribute Car Model was missing for the extracted tuple, the system adds separate tuples for each Car Model offered by the seller (e.g., x_3, x_4, x_5), along with a mutual exclusion dependency among them (x_3, x_4 and x_5 in Figure 1(b)). In general, complex correlations can

tid	VIN	Seller	Model	Price	prob
x_1	1A0	239	Honda	3500	0.3
x_2	1A0	231	Honda	4500	0.8
x_3	2B1	231	Honda	4500	0.8
x_4	2B1	231	Toyota	4500	0.8
x_4	2B1	231	Ford	4500	0.8

(a) CarAds



(b) Correlation

tid	Seller	Address	prob
y_1	239	12344	0.3
y_2	239	12345	0.3
y_3	231	12207	0.8
y_4	340	12209	0.9

(c) Location

tid	Seller	Reputed	prob
z_1	239	Good	0.3
z_2	231	Bad	0.7
z_3	340	Good	0.9

(d) Reputation

Figure 1: Data extracted by an information extraction engine. The correlations are indicated in part (b).

arise in any application that uses sophisticated learning and inference techniques like Bayesian networks [16, 24, 20].

Given such a database, a query of interest to a user may be: (Q) *List all Honda cars priced under $5000, sold by a highly reputed seller in the 12344/12345 area.* Q is a conjunctive query with a single output tuple (1A0) since seller 231 has bad reputation. It can be generated in two ways, either by joining tuples x_1, y_1 and z_1 or by joining tuples x_1, y_2 and z_1. Hence, it has lineage given by $(x_1 \wedge y_1 \wedge z_1) \vee (x_1 \wedge y_2 \wedge z_1)$, equivalently $(x_1 \wedge (y_1 \vee y_2) \wedge z_1)$. To answer the query, we need to determine the probability of this boolean formula.

Event Monitoring Application: Consider an event monitoring application such as an RFID ecosystem [30] that uses data collected by RFID devices (installed in a building) to detect different types of activities. Since the RFID data is noisy, incomplete and error-prone, it is subjected to probabilistic modeling using dynamic Bayesian networks [16, 24] and thereby, probabilities are assigned to the detected events. For instance, `obs(`X`,'PC',10:00 a.m)` is an event that specifies that the laptop PC was found near the RFID device X at 10:00 a.m and it is associated with a probability of occurrence. The detected events exhibit *spatial* and *temporal* correlations [17, 24] owing to the spatial locality of the sensors and temporal nature of the modeling process. Typical queries in such a system involve computing the probabilities of *compound events* expressed as compositions of simple events. For instance, a query of interest is: *What is the probability that the PC was transferred correctly from its starting location at Room A to the final location at the conference room?* If there are three RFID devices X_1, X_2 and X_3 between Room A and the conference room, then the compound event for which we need to know the probability is given by the boolean conjunction: `obs(`X_1`,'PC',10:00)` $\wedge$ `obs(`X_2`,'PC',10:05)` $\wedge$ `obs(`X_3`,'PC',10:10)`.

Many probabilistic database systems have been developed in recent years to handle large-scale uncertain data [8, 25, 27, 1, 28, 3]. While this prior work has made great strides in our understanding of how to manage large-scale uncertain data and how to evaluate various types of queries (including lineages) on them, only a handful of these systems can handle correlated data effectively. Sen et al. [28, 29] and Antova et al. [1] have addressed issues in representing and querying complex correlations in probabilistic databases. However, their proposed techniques are not scalable to large databases. Letchner et al. [24, 20] have developed techniques for processing queries over probabilistic event streams that have special correlation structure, *Markov sequences* – where the tuple (event) at time t is only directly influenced by the tuple (event) at time t–1. However, many of their queries were limited to simple event detection queries (corresponding to 1 level boolean queries) and decomposable aggregation queries. In prior work [18], we developed an index structure called *INDSEP* that enables scalable query processing over large correlated probabilistic databases. However our main focus was on the class of *inference* queries and aggregation queries and we did not consider conjunctive queries.

In this paper, we fill this gap by designing highly scalable algorithms and data structures for processing conjunctive queries over large correlated probabilistic databases. We follow the prevalent approach to conjunctive query evaluation in probabilistic databases by first computing the lineages [27] of the output tuples of the query (i.e. the boolean formulas corresponding to the existence of the tuples), and then computing the probabilities of the lineage formulas. The second task is in general very hard in presence of correlations. For instance, to compute the probability of a simple boolean formula $(a_1 \vee a_2 \cdots \vee a_{100})$ exactly, we need to capture all possible dependencies that exist among these random variables efficiently, preferably without constructing the full joint distribution (containing 2^{100} probabilities).We formally prove that the above problem is #P complete in general even in the simplest case of processing *read-once* (tree-structured) lineages on *lightly correlated* probabilistic databases such as *Markov sequences*. Despite this negative result, we show that we can process lineage formulas without having to construct very large probability distributions by developing algorithms based on *message passing* in *junction trees* – a concise data structure to represent correlations in the data. We characterize the complexity of processing lineages by introducing a parameter called *lwidth*, short for lineage-width (similar to the graph theoretic notion of *treewidth* [26]). The value of lwidth depends on the input lineage formula and the nature of the correlations in the data. If the lwidth induced by the formula is small, we attempt to process the formula exactly. For the other case, we develop an Monte Carlo approximation algorithm to estimate the probability. We scale our lineage processing algorithms to large-scale correlated probabilistic databases using our previously proposed *INDSEP* data structure [18]. To effectively integrate our algorithms with INDSEP, we introduce a lineage planning phase to our approach. This planning phase additionally enables us to perform several optimizations, such as exploiting any independence relationships that exist in the underlying data to reduce lwidths of the lineage formulas. Further, it allows us to process a batch of lineage formulas efficiently by sharing computation across formulas that have common subexpressions. The research contributions of the paper are:

- We extend the INDSEP data structure proposed in prior work for scalable evaluation of inference and aggregation queries [18] to conjunctive queries over *lightly correlated* probabilistic databases.
- We develop a novel algorithm for computing the probability of boolean formulas over junction trees (and a forest of junction trees), a fundamental problem that has not been considered before.
- We show how to process a batch of lineages efficiently by exploiting the common subexpressions in the formulas.

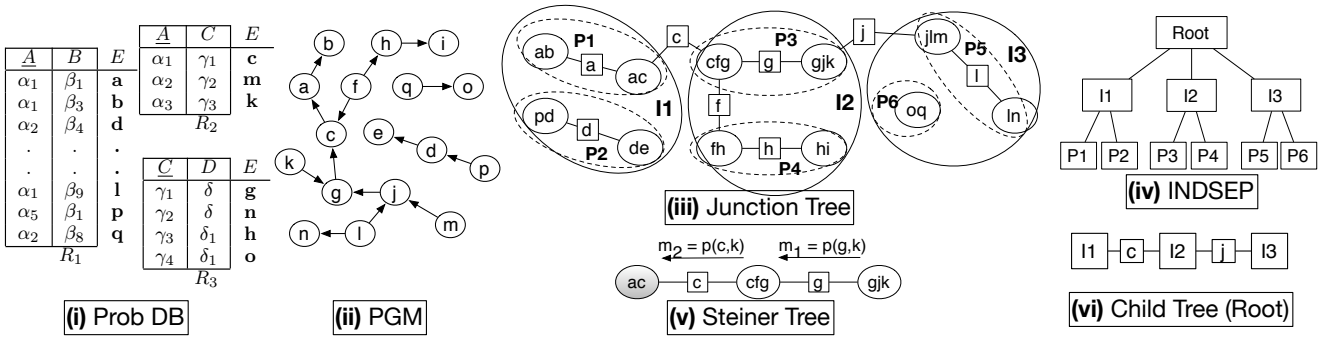


Figure 2: (i) A tuple uncertain probabilistic database; (ii) the graphical model that captures the correlations among the various tuples; (iii) the junction tree corresponding to the PGM in (ii); (iv) the INDSEP data structure corresponding to the junction tree in (iii); (v) Steiner tree computed while evaluating the inference query $\{a, k\}$. Pivot clique (ac) is shaded; (vi) The childTree stored in the root is shown here. Note that I_2 is connected to I_1 via c and to I_3 via j as indicated in (iii)

- We devise approximation algorithms based on Gibbs sampling for estimating the probability of boolean formulas over junction trees.

Outline: In Section 2, we provide background for junction trees and INDSEP. In Section 3, we describe our algorithms for processing lineage formulas over junction trees. Following this, we illustrate how to scale these algorithms by using the INDSEP data structure, in Sections 4 and 5. The approximation algorithms for lineage processing are illustrated in Section 5.3. We conclude with experiments in Section 7.

2. PRELIMINARIES

In this section, we briefly discuss the tuple uncertain probabilistic database model that we use in the paper and how we capture correlations in the data using a forest of junction trees. We then discuss how different types of queries may be evaluated over junction trees, and briefly describe the INDSEP data structure, proposed in our prior work, to speed up *inference* queries over junction trees.

2.1 Tuple Uncertainty Model

Although our system can handle attribute uncertainty, for clarity of presentation, we focus on tuple uncertainty in the bulk of the paper. In a tuple uncertain probabilistic database, each tuple is annotated with an *existence* probability (Figure 1(a)). We model such uncertainty by introducing a boolean random variable for each tuple t that denotes the existence of the tuple. It takes value 1 if t belongs to the database and 0 otherwise. Figure 2(i) shows a tuple uncertain probabilistic database on three relations R_1, R_2 and R_3. The random variables corresponding to the existence of the tuples are indicated in column E.

We allow representation of arbitrary correlations among the tuples by constructing a PGM on the tuple existence variables [28]. One possible set of correlations among the tuples in Figure 2(i) is shown in the PGM of Figure 2(ii). The edges in the graphical model capture the correlations among the random variables. Owing to space constraints, we do not discuss PGMs here. PGMs can be equivalently represented using *junction trees*, which we discuss next.

2.2 Junction Trees

A junction tree is an equivalent representation of a PGM [10] and is also known as a *tree decomposition*, or a *clique tree*. It is a concise representation of the joint probability distribution of a set of random variables. We will not discuss the actual construction of the junction tree owing to space constraints; however we describe some of its main properties. In a junction tree, there are 2 types of nodes, clique nodes (represented by circles) and separator nodes (represented by square). The clique nodes correspond to maximal cliques in the *triangulated* PGM and the separator nodes correspond to the cut vertices that separate the maximal cliques [14]. After fully constructing the junction tree, each clique/separator in the tree stores the joint probability distribution of the variables in the clique/separator. An example of a junction tree (for the PGM in Figure 2(ii)) is shown in Figure 2(iii). In this junction tree, clique (ab) stores the probability distribution $p(a, b)$ and separator c stores the probability distribution $p(c)$. The variables in a clique are directly correlated with each other and the separators encode the *conditional independences* that are present among the variables in the junction tree (though not all the conditional independences in the original PGM may be preserved). Given the knowledge of variable a, variable b becomes independent of c (i.e., conditional independence) since a is the separator between cliques (ab) and (ac). The conditional independence relationship can be mathematically expressed as:

$$p(a, b, c) = \frac{p(a,b)p(a,c)}{p(a)}$$

Also note the disconnection between clique (oq) and the rest of the tree. Disconnection implies independence, i.e., o and q are correlated with each other but are independent of the other variables in the junction tree. The overall joint distribution among all the random variables in the junction tree can be computed by multiplying the probabilities of all the clique pdfs and dividing by the product of all the separator pdfs. The joint distribution for the above example is shown below:

$$\frac{p(ab)p(ac)p(pd)p(de)p(cfg)\ldots p(ln)p(oq)}{p(a)p(d)p(c)\ldots p(l)} \quad (1)$$

Junction trees whose maximum clique size (*treewidth* of PGM) can be bounded by a small constant are typically called *thin* junction trees [22, 2]. We call probabilistic databases that can be represented using thin junction trees *lightly correlated probabilistic databases*. Examples of lightly correlated probabilistic databases include Markov sequences [20, 17].

2.3 Why Junction Trees ?

In the machine learning community, junction trees are considered a very useful tool since they can be used to compute *all marginals*, i.e., the probability distributions of all

the random variables, in two passes over the tree. Furthermore, junction trees are very effective at reusing computation among different inference queries. Hence in our context, where we typically have few updates to existing data, but many queries, a junction tree is a natural intermediate structure to build since it speeds up query evaluation by orders of magnitude. While junction trees can be built efficiently for lightly correlated probabilistic databases, there have been concerns about the feasibility of building a junction tree over *highly correlated* probabilistic databases, that generate large cliques. Since the clique's pdf is exponential in the number of its constituent variables, it is impractical to build junction trees with large cliques. There have been many proposals to alleviate this problem by performing approximations. We mention two such approaches here. Firstly, Choi et al. [4] provide techniques for selectively removing edges from the underlying PGM, so as to generate smaller cliques in the junction tree. The authors propose to adjust the values of the remaining parameters to compensate for the deleted edges. In another technique called *HUGS*, proposed by Kjaerluff et al. [19], the cliques store approximate versions of their pdfs. Each clique approximates its pdf with a set of factors which correspond to the relevant factors in the associated PGM. The authors adapt the junction tree message passing algorithms (discussed next) for this representation via Gibbs sampling. More recently, Chechetka et al. [2] proposed new algorithms for directly learning thin junction trees from data. Such techniques can be used to scale junction trees to highly correlated probabilistic databases.

2.4 Query Processing over Junction Trees

We first describe a simple operation on probability distribution functions which we use throughout the paper.

Elimination: Eliminating a variable v from a joint pdf $p(V)$ is the process of computing a new pdf over $V \setminus v$, i.e., $p(V \setminus v)$, by summing the entries which have the same value for $V \setminus v$. Suppose we have pdf $p(A, B)$, where A and B are boolean random variables. Then, after eliminating B, $p(A)$ can be computed, (for $a \in \{0, 1\}$) using the equation:

$$p(A = a) = p(A = a, b = 0) + p(A = a, b = 1)$$

Next, we describe how to evaluate various types of queries over junction trees.

Inference queries: An inference query is specified by a set of variables V and the output is defined to be the joint probability distribution over the variables in V.

The naive way to answer such an inference query is to construct a joint probability distribution over all the variables using equation (1), and then to eliminate the non-query variables. This is clearly not feasible. However this can be made much more efficient by eliminating the non-query variables earlier during the execution. This is the essence of the HUGIN algorithm [6, 18] which we illustrate now. We first compute the smallest subtree (Steiner tree) of the junction tree that covers all the query variables (the best case scenario being that a single clique contains all the query variables). For example, to evaluate the inference query {a,k} over the junction tree shown in Figure 2(iii), we first construct the Steiner tree between cliques (ac) (since it contains variable a) and (gjk) (since it contains k) as shown in Figure 2(v). We then choose a *pivot clique*, the node to which all other cliques in the Steiner tree send their message. Suppose we choose clique (ac) as our pivot. The following sequence of messages are passed to evaluate $p(a, k)$, also shown in Figure 2(v):

- Clique (gjk) eliminates j from its pdf $p(g, j, k)$ to compute $p(g, k)$. It eliminates j since it is not required by the query. However, it still needs to retain g for correctness since g is also present in the clique (cfg). (gjk) sends $m_1 = p(g, k)$ as a message to the clique (cfg).
- Clique (cfg) now multiples m_1 with its pdf $p(c, f, g)$ to get $p(c, f, g, k)$. It eliminates f, g to get $p(c, k)$ and sends $m_2 = p(c, k)$ to clique (ac).
- Clique (ac) multiplies m_2 with its pdf $p(a, c)$ to get $p(a, c, k)$, then eliminates c to get the result $p(a, k)$.

Note that for suitably chosen query variables, e.g., $\{a, n\}$, the size of the Steiner tree can be almost as large as the size of the junction tree itself. Thus, even for 2 or a 3 variable query, the query evaluation time can be very high. This limitation can be resolved by using the INDSEP data structure, which we discuss in Section 2.5.

Lineage queries: Lineage queries are important in the context of probabilistic databases. Lineage or Provenance [27] of a tuple in a database is a boolean formula, which represents all possible derivations of the tuple. Suppose that we want to execute query $R = \Pi_D(R_2 \bowtie_C R_3)$ over the database in Figure 2(i). Consider the tuple $(\delta) \in R$. It is generated by the projection of tuples $(\alpha_1, \gamma_1, \delta)$ and $(\alpha_2, \gamma_2, \delta)$ which are present in $(R_2 \bowtie_C R_3)$. If either of the tuples are present in the join, then the output will contain (δ). Hence, the lineage of (δ) is written as the boolean OR of the lineages of the tuples $(\alpha_1, \gamma_1, \delta)$ and $(\alpha_2, \gamma_2, \delta)$. $\lambda(\delta) = \lambda(\alpha_1, \gamma_1, \delta) \vee \lambda(\alpha_2, \gamma_2, \delta)$. The tuple $(\alpha_1, \gamma_1, \delta)$ itself is dependent on the presence of both tuples (α_1, γ_1) and (γ_1, δ) in relations R_2 and R_3 respectively. Hence, the lineage of $(\alpha_1, \gamma_1, \delta)$ is written as the boolean AND of the tuples (α_1, γ_1) and (γ_1, δ). $\lambda(\alpha_1, \gamma_1, \delta) = \lambda(\alpha_1, \gamma_1) \wedge \lambda(\gamma_1, \delta) = c \wedge g$. Similarly, $\lambda(\alpha_2, \gamma_2, \delta) = m \wedge n$. Hence, we can write the overall lineage of the output tuple (δ) as the following boolean formula: $((c \wedge g) \vee (m \wedge n))$. This formula is an example of a *read-once* boolean formula [12], i.e., each boolean variable appears exactly once in the formula. It can be represented as a parse tree, as illustrated in Figure 3(a). The root node of the tree corresponds to the entire boolean formula. Intermediate nodes correspond to subformulas, i.e., they represent the formula of the subtree below them, e.g., in Figure 3(a), the intermediate node 1 corresponds to $(c \wedge g)$ and the node 2 corresponds to $(m \wedge n)$.

A lineage query requires us to evaluate the probability of the lineage formula given the probability distribution of the input variables. A naive method is to use the joint probability distribution over the variables in the lineage. Suppose we need to compute the probability distribution of $(a \wedge b)$. We first compute the joint pdf over the variables a and b, i.e., $p(a, b)$. Then, we use the *conditional distribution*, $p(a \wedge b | a, b)$ which specifies how the random variable $a \wedge b$ depends on a and b. In this case, it is just the truth table of the boolean AND logic. We multiply $p(a, b)$ with $p(a \wedge b | a, b)$ and obtain $p(a, b, a \wedge b)$. Following this, we eliminate a and b from $p(a, b, a \wedge b)$ and determine $p(a \wedge b)$. Note that $a \wedge b$ is just another boolean variable with domain {0,1}.

Expressions: We collectively refer to lineages (e.g., $b \wedge c, d \wedge e$) and singleton random variables (e.g., a, f) as *expres-*

sions. As we describe later, we will often need to compute the joint probability distribution of a set of expressions, e.g., $\{a, b \wedge c, d \vee e\}$ – we call them *ExpressionSets*.

The algorithm described above does not scale to large expressions since the initial step computes a huge joint distribution. Even a simple formula of size 25 needs to compute a joint pdf of size 2^{25}, which is very inefficient. We develop better algorithms for processing lineages over junction trees, based on message passing in Section 3.

2.5 INDSEP

We proposed the INDSEP data structure in our prior work to scale inference and aggregation queries to very large disk resident junction trees. It is constructed from a hierarchical partitioning of the junction tree. The index corresponding to the junction tree shown in Figure 2(iii) is shown in Figure 2(iv). The hierarchical partitioning that generated the index is shown with dotted lines in Figure 2(iii). Initially, the junction tree is partitioned into I_1 and I_2 and I_3. Further, I_1 is partitioned into P_1 and P_2 and so on. Each node in the index represents a subtree of the junction tree. For instance, I_2 represents the subtree containing the cliques (gjk), (cfg), (fh), and (hi). Similarly, the leaf node P_1 corresponds to the partition P_1 in Figure 2(iii) composed of cliques (ab),(ac) and the separator (a). Each node in the index also stores a tree indicating how its children are connected. For instance, the child tree corresponding to the root node is shown in Figure 2(vi).

The key idea in INDSEP is the use of precomputed *shortcut* potentials to speed up the HUGIN algorithm by avoiding visiting every node in the junction tree. A shortcut potential is the joint distribution of all the separators adjacent to a subtree in the junction tree. For example, I_2 stores the shortcut potentials of the subtree P_3, which is $p(c, f, j)$ (since c, f and j are the separators adjacent to P_4) and that of P_4, which is $p(f)$. Query processing over INDSEP follows by recursion on the index tree. For instance, the inference query {a,n} is converted into the following recursive queries: {a,c} on I_1, {c,j} on I_2 and {j,n} on I_3. Variables c and j are added to the recursion in order to capture all possible correlations that exist between a and n. Note that $p(c, j)$ is directly obtained from the shortcut potential of P_3 and not by traversing its subtree - which leads to order of magnitude improvements in query processing time for large graphs. In addition, we also developed algorithms for processing aggregation queries and extraction queries [18].

3. LINEAGE PROCESSING ALGORITHMS OVER JUNCTION TREES

In this section, we develop algorithms for processing lineage formulas over junction trees. Although our focus is on read-once lineages, our algorithms for lineage processing can be applied even to non-tree structured lineages. While it is known that read-once lineages can be processed in polynomial time for tuple independent probabilistic databases [8, 28], we can also show the following result:

THEOREM 1. *Processing read-once lineages over lightly correlated probabilistic databases, even restricted to the class of Markov sequences, is #P-complete.*

We omit the proof due to space constraints. However, most real world datasets do not exhibit this worst case behavior.

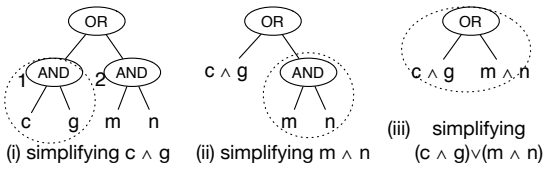


Figure 3: Different stages in the simplification process

The junction trees are often disconnected, which allows us to perform exact computation.

3.1 Message Passing for Lineage Processing

Before discussing the message passing algorithms, we introduce a parameter called *lwidth* that captures the complexity of processing a lineage formula. We illustrate lwidth using the naive method we described in Section 2. Suppose that we want to compute the probability of the lineage formula $(c \wedge g) \vee (m \wedge n)$. In the first step, we evaluate the inference query $\{c, g, m, n\}$ and determine $p(c, g, m, n)$. In the next step, called *simplification*, we evaluate the probability of the lineage. As described earlier, we first multiply the pdf with $p(c \wedge g|c, g)$ to get $p(c, g, m, n, c \wedge g)$ and then eliminate c and g to get $p(c \wedge g, m, n)$. We proceed in similar fashion to compute $p(c \wedge g, m \wedge n)$, and finally the probability $p((c \wedge g) \vee (m \wedge n))$. We pictorially illustrate the simplification process in Figure 3. The above algorithm creates several intermediate pdfs, the size of the largest intermediate pdf being $2^5 = 32$. The time taken by the algorithm is influenced by the size of the maximum pdf created. We formally capture the complexity of the algorithm by introducing the parameter called *lwidth*.

DEFINITION 1. Lwidth *is the* treewidth *[26] of the graphical model obtained by combining the junction tree and a graphical model corresponding to the lineage formula.*

Intuitively, lwidth corresponds to the size of largest intermediate pdf created if the problem is solved "optimally", i.e., with an optimal junction tree, optimal *rewriting* of the boolean formula and the optimal algorithm. Like treewidth, lwidth is in general NP-hard to compute exactly, and we can only estimate it, i.e., compute an upper bound for it in most cases. In the example above, lwidth was estimated to be 5.

Next, we develop an improved recursive algorithm for lineage processing based on message passing. The trick is to create smaller intermediate pdfs by performing the simplifications eagerly, whenever we detect a message that can be simplified. We illustrate our Eager strategy next.

Improvement 1 (Eager): As with the HUGIN algorithm (Section 2), we first select a pivot and construct a Steiner tree over the variables referred in the lineage formula. At each clique, we multiply all the incoming messages and the clique's pdf to get an intermediate pdf, and then eliminate the non-query variables. However, we try to simplify the pdf as much as possible based on the given lineage before sending the resulting pdf as a message. Let us see how we would compute $p((c \wedge g) \vee (m \wedge n))$ using the Eager strategy. Suppose that we select the clique (cfg) as the pivot. The Steiner tree for this lineage is the path connecting the clique (ln) to the clique (cfg) (Figure 2(iii)). The algorithm proceeds as follows:

- The clique (ln) sends message $p(l, n)$ to the clique (jlm).
- Now, clique (jlm) multiplies the incoming message $p(l, n)$ with its pdf $p(j, l, m)$ to get $p(j, l, m, n)$ (it also divides by $p(l)$). It eliminates l to get $p(j, m, n)$. Since m and n are together (and do not appear elsewhere in the junction tree), it simplifies $p(j, m, n)$ here itself, to get $p(j, m \wedge n)$ and sends it to clique (gjk).
- Clique (gjk) multiplies $p(j, m \wedge n)$ with its pdf $p(g, j, k)$, eliminates k & j, sends $p(g, m \wedge n)$ to clique (cfg).
- Clique (cfg) eliminates f from its pdf $p(c, f, g)$ to get $p(c, g)$ and multiplies it with $p(g, m \wedge n)$ to get $p(c, g, m \wedge n)$. After relevant simplifications, the clique (cfg) computes the final result $p((c \wedge g) \vee (m \wedge n))$.

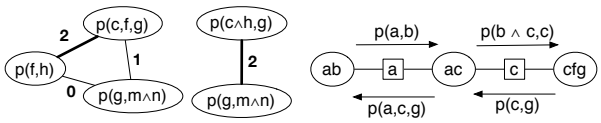
In this approach, the maximum intermediate pdf size generated is 2^4. This reduction is small for the above toy example, but it can be very large for larger lineages, since the computational complexity is exponential in the lwidth.

We can reduce the intermdiate pdf sizes induced by the lineage even further by performing the simplification even before we multiply all of the incoming messages at a clique node. Suppose that we want to compute the probability of another boolean formula $(c \wedge h) \vee (m \wedge n)$ and we pick the clique (cfg) as the pivot using the Eager strategy. In the last step, the Eager strategy would require us to multiply $p(c, f, g)$, $p(f, h)$ and $p(g, m \wedge n)$ to get $p(c, f, g, h, m \wedge n)$. This would result in a pdf of size 2^5. Alternatively, we can multiply the pdfs $p(c, f, g)$ and $p(f, h)$ to get $p(c, f, g, h)$, eliminate f and simplify the resulting pdf to $p(c \wedge h, g)$. We can now multiply this with $p(g, m \wedge n)$ and eliminate g to get the output. In this case, we would only create a pdf of size 2^4. This observation brings us to our next improvement.

Improvement 2 (Eager+Order): We construct a complete edge weighted graph in which each node corresponds to the probability distribution which is to be multiplied. The weight of an edge is set to be equal to the *amount of simplification* that is possible if we multiply the pdfs corresponding to its adjacent nodes. The amount of simplification while multiplying two probability distributions f_1 and f_2 is given by $|f_1 \cup f_2| - |f|$, where f is the simplified output after multiplying f_1 and f_2. For instance, in the example above, when we multiply $p(c, f, g)$ and $p(f, h)$ the final output is $p(g, c \wedge h)$, hence the simplification is given by $4 - 2 = 2$. We greedily pick the edge with the largest weight and multiply the probability distributions together. We then perform simplification and update the graph, by clustering the 2 nodes together and recomputing the weights of all edges incident on the newly created node. We continue this process until all the probabilities have been multiplied, i.e., when there are no more edges in the graph. The order of multiplication for the above example is illustrated in Figure 4(a). The edges that are selected by the heuristic are darkened. In the first step, we multiply $p(c, f, g)$ and $p(f, h)$ to obtain $p(g, c \wedge h)$. In the second step, there is only one edge left, hence we multiply this with $p(g, m \wedge n)$.

3.2 Pivot Selection

Another factor influencing the intermediate pdf sizes induced by the lineage formula is the pivot selected by the algorithm. Suppose we want the probability of $(b \wedge c) \vee g$. The Steiner tree corresponding to this query is shown in Figure 4(b). As shown in the figure, there are 3 choices of pivot selection, i.e., one of (ab), (ac) or (cfg). We will evaluate the lwidth estimate for two different pivot locations - clique (ab) and clique (cfg).



(a)Eager+Order heuristic (b)Pivot selection

Figure 4: (a)Illustrating the order of multiplication and simplification in Eager+Order heuristic. Initially, we multiply pdfs $p(f, h)$ and $p(c, f, g)$ since that edge has the maximum weight. (b) When pivot = (cfg), the sequence of messages passed is indicated above the graph (right arrows). When pivot = (ab), the sequence of messages is indicated below the graph (left arrows).

Case 1: Pivot = (ab): The sequence of messages passed in this case are indicated in Figure 4(b). Clique (cfg) sends message $p(c, g)$ to clique (ac). Now, (ac) multiplies it with its pdf $p(a, c)$ to get $p(a, c, g)$, which is sent to clique (ab). (ab) multiplies $p(a, c, g)$ with $p(a, b)$ to get $p(a, b, c, g)$. Then it eliminates a to get $p(b, c, g)$ from which we get $p((b \wedge c) \vee g)$. The maximum intermediate pdf for this pivot location is 2^4.

Case 2: Pivot = (cfg): In this case, the clique (ab) sends the message $p(a, b)$ to clique (ac). Now, (ac) multiplies it with its pdf $p(a, c)$ to obtain $p(a, b, c)$. It also eliminates a since it is not required and simplifies $p(b, c)$ to $p(b \wedge c, c)$, which is then sent to clique (cfg). (cfg) first computes $p(c, g)$ by eliminating f from its joint pdf and then multiplies with $p(b \wedge c, c)$ to get $p(b \wedge c, c, g)$ which is then simplified to the result $p((b \wedge c) \vee g)$. Note that in this case, the maximum pdf size generated in this case is just 2^3. Since there are only n choices, (n is the number of clique nodes in the Steiner tree) for the pivot position, we use the naive approach in which we estimate the *lwidth* for each pivot location. We then select the node which induces the smallest lwidth as the pivot. In future, we plan to develop more efficient algorithms for pivot selection.

3.3 Dealing with Disconnections

Until now, we have assumed that the junction tree is a single tree that connects all the variables. However, the random variables may be correlated as a forest of junction trees. Here, we adapt our lineage processing algorithms to deal with these disconnections. In the first step, we split the query into subqueries over each of the components in the junction forest. For instance, consider the query $Q = (d \vee e) \wedge (b \vee c \vee q)$ over the junction tree in Figure 2(iii). We see that the variables in the lineage formula belong to three different connected partitions $\{d, e\}$ in one partition P_2, $\{b, c\}$ in the second partition, P_1 and q in the third partition P_6. Hence, we split Q into three subqueries, one for each connected partition. However, we see that instead of posing an inference query $\{d, e\}$ on P_2, we can actually pose a lineage query $d \vee e$ on P_2. Similarly, we can pose query $b \vee c$ on P_1 and query q on P_6. After executing each query independently on each of the components in the junction forest, we get 3 pdfs, namely $p(d \vee e)$, $p(b \vee c)$ and $p(q)$. We combine the result pdfs together using the Eager+Order heuristic as described before.

4. LINEAGE PROCESSING USING INDSEP

In the previous section, we described efficient techniques for processing lineage formulas on junction trees. However, they do not scale to large scale junction trees, since performing a lineage query over few variables may require the algorithm to access the entire junction tree. Hence, we use INDSEP, the recursive query processing framework that we developed in our prior work [18], to scale our lineage processing algorithms.

4.1 Recursive Approach

Recall that INDSEP is a hierarchical, tree-like data structure built on top of a junction tree (or a forest of junction trees). Lineage processing on INDSEP, analogous to any hierarchical index proceeds by recursion. Now, we describe the key recursion step for processing lineage formulas. During lineage processing, each index node involved is given as input, a set of expressions *ExpressionSet* which has two types of expressions contained in it: (1) Lineage formulas, which we denote by $\vec{\lambda}$, (2) Singleton random variables, which we denote by $\vec{V}$. The Index node is required to compute as output, the joint probability distribution between the expressions in the set, i.e., $p(\vec{\lambda} \cup \vec{V})$. The complete lineage processing algorithm is shown in Algorithm 1. We explain the algorithm using a simple example.

Algorithm 1 process_lineage(inode, $\vec{\lambda}$, $\vec{V}$)

1: $qvars = \vec{V} \cup variables(\vec{\lambda})$
2: **for** all $v \in qvars$ **do**
3: $found[v] = search(v, inode.vars)$
4: Graph $tree = inode.childTree.Steiner\_tree(found)$
5: {$found$ = set of child inodes that contain qVars}
6: JunctionTree jtree = null
7: **for** each $node \in tree$ **do**
8: $lvars = node.vars \cap vars(\vec{\lambda})$
9: $ivars = node.vars \cap \vec{V}$
10: nrs = neighboring separator variables of $node$
11: **if** $lvars = \phi$ **then**
12: **if** $ivars = \phi$ **then**
13: jtree.add(inode.shortcutPotential(nrs))
14: **else**
15: jtree.add(inference($nrs \cup ivars$))
16: **else**
17: $\vec{\lambda'}$ = **getSubExp**($\vec{\lambda}, lvars$)
18: $\vec{V'} = nrs \cup ivars \cup (lvars - vars(\vec{\lambda'}))$
19: **if** $node$ is a leaf **then**
20: jtree.add(node.junctionTree.process_lineage($\vec{\lambda'}, \vec{V'}$))
21: **else**
22: jtree.add(process_lineage(node,$\vec{\lambda'}, \vec{V'}$))
23: **return** jtree.process_lineage($\vec{\lambda}, \vec{V}$)

Suppose we need to compute the probability of lineage formula $\lambda = ((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$ (shown in Figure 5(i)) over the INDSEP data structure shown in Figure 2(iv). In the first step, we determine the random variables contained in λ, in our case this is $\{b, c, d, e, n, o\}$. For each variable here, we search for the child index node to which it belongs (pick arbitrarily if a variable belongs to multiple child nodes) (Steps 1-3). We then collect the variables present in each of the child nodes (Steps 8-9). In our example, the child node I_1 contains the set of random variables $\{b, c, d, e\}$ and I_2 contains the set $\{n, o\}$. We now construct a Steiner tree over the childTree of the node (Section 2.5), joining all the child nodes which contain query variables (Step 4). In our example, we construct the Steiner tree connecting I_1 and I_3, over the childTree of the root (Figure 2(vi)). Now, we need to determine the recursive calls to be made over the nodes in the Steiner tree, i.e., the ExpressionSet ($\vec{\lambda'} \cup V'$) that needs to be posed to continue the recursion.

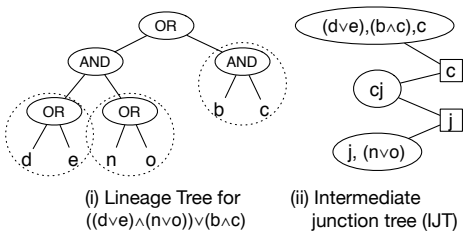


Figure 5: (i) illustrates the computation of subexpressions (ii) shows an intermediate junction tree generated in the root node while processing $((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$

Determining Recursive Calls: We scan the set of random variables allotted to a child node and check if we can group the variables present in the child to form subformulas of the input lineage. These are added to λ' (getSubExp, Step 17). The remaining variables, which could not be grouped are collected in V'. In our example, in child I_1, the random variables b,c,d,e can be grouped into $\{d \vee e, b \wedge c\}$ (Figure 5(i)) and is therefore added to $\vec{\lambda'}$, and there are no more variables, hence V' is empty. Note that we still need to capture the correlations among the variables $b \wedge c$, $d \vee e$ and the rest of the variables in the query. Hence, we add the list of variables in the relevant separators of the child node to V' to capture all correlations (Step 15, 18). In our example, we add random variable c to V' since it is the separator of I_1 (Figure 2(vi)). Hence, the complete ExpressionSet for child I_1 is given by $\{b \wedge c, d \vee e, c\}$.

Recursion: Now, we proceed with recursive calls on the child nodes using the ExpressionSet assigned to them. In the special case when the ExpressionSet contains only separator variables, we can obtain the probability distribution directly from the shortcut potential (Step 13). In our example, since I_2 does not contain any query variables, the only variables added to its ExpressionSet are its separator variables (Figure 2(vi)), given by $\{c, j\}$. This can be answered directly using the shortcut potential of I_2. To bottom out the recursion at the leaf nodes of the index, we use the algorithm of Section 3 to process the issued ExpressionSet over the junction trees contained in the leaf nodes (Step 20). Although the algorithms in Section 3 were designed for a single lineage formula, we can adapt them to process ExpressionSets easily, by ensuring that we do not eliminate the random variables that belong to other terms in the set.

Assembling Child Results: After obtaining the results from the child nodes, they are assembled as a junction tree - we call this the *intermediate* junction tree (IJT). We now evaluate the remaining portion of the lineage over the IJT and return the result to the parent node. In our example, the child node I_1 returns $p(b \wedge c, d \vee e, c)$, I_2 returns $p(c, j)$, I_3 returns $p(j, n \vee o)$, which is assembled as the junction tree shown in Figure 5(ii). Note that the cliques in the IJT contain newly created boolean variables $d \vee e$, $n \vee o$ and $b \wedge c$.

We now compute the probability of $((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$ over this junction tree using the algorithms of Section 3.

4.2 Shortcomings

Although the above algorithm works correctly, it has a few shortcomings which we describe here.

Feasibility: The complexity of the above algorithm is not entirely evident from the algorithm itself and is highly dependent on the nature of the underlying junction tree, and the structure of the lineage formula. If the random variables in the junction tree are independent, the algorithm runs quickly even for large lineage formulas. If the variables are correlated, then the complexity depends on the placement of the variables of the lineage in the junction tree i.e., if the lineage formula can be decomposed over the junction tree such that the subexpressions are present locally, the algorithm is efficient. In the worst case, when the variables are spread out arbitrarily, the algorithm can take time exponential in the size of the formula. This high variance in the processing time is troubling and must be mitigated.

Redundant Variables: Since our underlying data model is a *forest* of junction trees, there are a number of independence relationships that are present among the random variables. However, Algorithm 1 is currently unaware of these independence relationships and might perform unnecessary computation. We illustrate this with an example. Consider the index shown in Figure 2(iv). Suppose we are interested in computing the probability of $a \wedge o$. The efficient way to process this lineage is to compute $p(a)$ and $p(o)$ separately since they are independent, and use them to determine the probability of $a \wedge o$. However, Algorithm 1 proceeds by making the following recursive calls on the child nodes I_1: $\{a, c\}$, I_2: $\{c, j\}$ and I_3: $\{j, o\}$, which is significantly more computation since we have to maintain joint probability distributions $p(a, c)$, $p(c, j)$ and $p(j, o)$. The reason behind this is that the knowledge of the disconnection is "hidden" in the leaf of the INDSEP and can only be discovered when the recursion reaches the leaf. Clearly, this computation is redundant since a and o are actually independent.

Multiple Lineage formulas: Many output tuples of a conjunctive query share common subexpressions in their lineage. Instead of computing the probability of the same expressions repeatedly, we can exploit this commonality by reusing the previously computed results. This could bring down computation time by a large fraction. We note here that such sharing is possible not only when the lineages share terms, it is quite useful even otherwise when the Steiner trees corresponding to the lineages share large paths. For instance, consider the lineages $c \wedge n$ and $b \wedge m$. In this case, the lineage $c \wedge n$ recursively generates ExpressionSets $\{c, j\}$ on I_2 and $\{j, n\}$ on I_3. Similarly, the lineage $b \wedge m$ generates ExpressionSets $\{b, c\}$ on I_1, $\{c, j\}$ on I_2 and $\{j, n\}$ on I_3. The ExpressionSet $\{c, j\}$ is common to both lineages and it needs to be computed only once.

To effectively handle all the three issues discussed above, we introduce a lineage planning phase to our algorithm, which we describe in the next Section.

5. LINEAGE PLANNING & EVALUATION

We develop a two-pass approach to lineage processing - in the first step, we work through the index and formulate a plan for the lineage and in the next step, we execute the formulated lineage plan. We illustrate the complete sequence of query processing operations in Figure 6. As shown in the figure, a conjunctive query is first executed by a relational query processor and lineages of the output tuples are computed (Section 6). Following this, the lineage planning and the lineage processing phases occur; we describe these here.

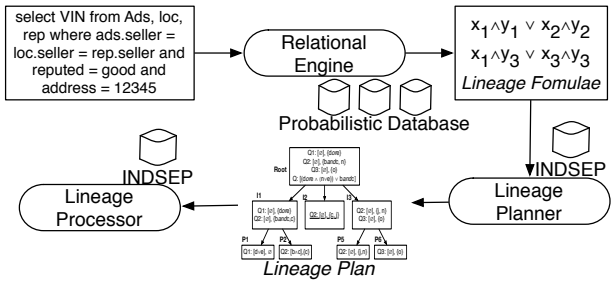


Figure 6: System Overview: Input conjunctive query is first executed by the relational engine which computes lineages of output tuples. Lineage Planner then computes an optimal plan for processing these lineages, which is executed by the Lineage Processor.

5.1 Lineage Planning

In this phase, we determine the lineage plan, i.e., the set of recursive calls to be made in each index node in the INDSEP data structure. In addition, we optimize the lineage plan by (a) identifying common subexpressions across a batch of lineages and sharing such computation. (b) identifying redundancies (c) estimating the lwidth induced by lineage formula at the intermediate nodes in INDSEP.

Naive Plan: We first describe how to compute a naive lineage plan. Note that we are given a batch of lineages as input to the system. Just as in Algorithm 1, we determine the ExpressionSet corresponding to each child node. However, in this case, we have a list of ExpressionSets corresponding to each index node. We denote this list by the notation E^{node}. The expression set corresponding to the ith lineage is given by E_i^{node}. The lineage plan is a hierarchical data structure (corresponding to INDSEP) that essentially stores the E^{node} list against each node. Now we discuss how to optimize the naive lineage plan.

Batch/Multiple Lineage Processing: The INDSEP data structure naturally allows the sharing of computation between lineage formulas that share subformulas. This results not only in reduced number of disk accesses but also cpu processing time. Here, we look for duplicate ExpressionSets in each node in the lineage plan and remove them. After computing the naive lineage plan, each node N in the lineage plan stores the list E^N as described above. Now, we modify this list by removing the duplicate entries of ExpressionSets. This ensures that we will only execute distinct ExpressionSets. However, we need to maintain additional bookkeeping information corresponding to the duplication to execute the plan correctly. Specifically, we need a mapping from the list E^P to the list E^N (P is the parent of N). This mapping helps the lineage processor to correctly identify the parent recursive calls generating the ExpressionSets.

However, even more aggressive sharing can be performed. Suppose that processing lineage λ_1 generates ExpressionSet $\{j, m, n\}$ on child I_3 and processing λ_2 generates ExpressionSet $\{m, n\}$ on child I_3. The above technique would treat the two ExpressionSets separately since they are different.

However, a more useful technique here is to first compute $p(j, m, n)$ by evaluating ExpressionSet $\{j, m, n\}$ and then using this result to compute $p(m, n)$ (by eliminating j) which is the result of evaluating ExpressionSet $\{m, n\}$. We plan to consider such aggressive sharing in our future work.

Redundancy Detection: For simplicity, we discuss the case of removing redundancies for a single lineage λ (The discussion extends to batch of lineages as well). We take care of detecting redundancies at the root level of the index itself. Given a lineage formula λ as input, we split it into multiple ExpressionSets, where each ExpressionSet corresponds to a connected component, just as we described in Section 3.3. We modify INDSEP to additionally store the knowledge of the components in the junction tree. For each random variable, INDSEP stores the id of the component to which it belongs in a hash table. The hash table is constructed while building the index and we use a Union-Find data structure [5] to maintain this data structure up-to-date in response to updates (inserting new tuple involves adding a new random variable). By splitting the input lineage in this manner, we guarantee that none of the nodes in the lineage plan contain an ExpressionSet with two independent variables. Hence, we never compute a joint pdf between a pair of independent variables. After splitting the lineage as described above, we use the previously described multiple lineage planning algorithm to determine the lineage plan. In addition, we mark the root so as to *combine* the results of each of the lineages to produce the final result.

Lwidth Estimation: After modifying the lineage plan as discussed above, we evaluate the feasibility of executing each step of the plan. The feasibility is determined by estimating the lwidth value at each node in INDSEP, since we process ExpressionSets at each node. At the leaf nodes of the tree, which correspond to the disk partitions, we process ExpressionSets on the junction tree corresponding to the disk partition (Step 20 in Algorithm 1). At each internal node of INDSEP, after building the IJT, we process ExpressionSets on it (Step 23 in Algorithm 1). To estimate the lwidth values, we use the eager+order heuristic described in Section 3. In addition, we also compute the optimal pivot locations. Note that we need not know the actual pdfs, but only the sets of variables over which they are defined. Hence, the time for estimating lwidths and pivots is quite small, compared to the lineage processing times. When the junction tree is disconnected, we estimate lwidth and determine the pivot for each partition separately along with the lwidth involved in combining the results from the different partitions together. We enforce an *lwidth threshold* on the computation in order to bound the lineage processing time. If the lwidth estimate at a given node exceeds the threshold, then we mark the relevant node in the lineage plan to indicate that we need to perform approximations (Section 5.3). Our method ensures that we use approximations only for the portions of the lineage formula that have large lwidths and not for the complete formula as a whole. As we show in our experiments, this significantly improves the quality of our approximations.

5.2 Lineage Plan and Execution

Lineage Plan: As specified earlier, the lineage plan data structure specifies the list of ExpressionSets to be executed at each index node in INDSEP. The lineage plan is a tree based data structure, where each node in the tree corresponds to one of the index nodes in INDSEP. Each lineage plan node N (with parent P) in the lineage plan contains the following which were computed in the previous section:

- **(C1)** Id of the index node to which it corresponds
- **(C2)** List of ExpressionSets: $E^N = \{E_1^N, E_2^N, \dots\}$
- **(C3)** Optimal pivot(s), lwidth(s) - do we approximate ?
- **(C4)** Can we get results from shortcut potentials ?
- **(C5)** Pointers to children, Hashtable ($E^P \rightarrow E^N$)
- **(C6)** Whether to combine multiple lineage results ? (e.g., due to disconnections)

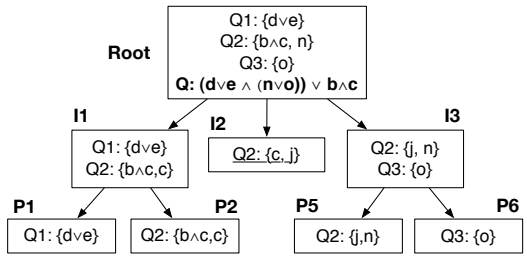


Figure 7: Lineage Plan for lineage $\lambda = (d \vee e)(n \vee o) \vee (b \wedge c)$

An example of a simple lineage plan, for the lineage $\lambda = (d \vee e)(n \vee o) \vee (b \wedge c)$ is shown in Figure 7. Notice that we have indicated the INDSEP node to which each plan node corresponds. Owing to disconnections, λ is initially split into three ExpressionSets $\{d \vee e\}$, $\{b \wedge c, n\}$ and $\{o\}$. Hence, the root has 3 ExpressionSets, and an additional ExpressionSet while tells the lineage processor to combine the results together. Q_1 introduces recursive calls $\{d \vee e\}$ over I_1 as shown in the figure. Also Q_2 introduces recursive calls $\{b \wedge c, c\}$, on I_1, $\{c, j\}$ on I_2 and $\{j, n\}$ on I_3 as shown. Note that the ExpressionSet corresponding to I_2 is marked, since it can be directly obtained from the shortcut potential of I_2. Q_3 introduces recursion over I_3 as shown in the figure.

Executing the Plan: We execute the lineage plan via recursion over the lineage plan structure. We explain the key recursive step here. Given a lineage plan P on an index node I, we recursively assign children of P to the respective child nodes of I. We collect the results of the executions of each of the child nodes and construct the IJTs (Section 4) using the hashtable mappings. Now, we execute the ExpressionSets contained in I over the IJTs using the optimal pivot locations, and return the result to $I's$ parent. We have 2 special cases to take care of: (1) when the lineage plan is marked, we directly obtain the result from the shortcut potential (2) Whenever the lwidth exceeds the threshold, we perform approximations while processing lineage (Section 5.3). Whenever the index node corresponds to a leaf, we execute the ExpressionSets on the junction trees corresponding to the leaf and return the results to the parent node.

5.3 Approximation Technique

In this section, we describe how we deal with lineage formulas that induce large lwidths on the underlying junction tree. Currently, we use a simple Monte Carlo technique which is based on Gibbs sampling [11]. The accuracy of the estimates can be improved by using more samples. In our ongoing work we are developing new techniques based on

modifying lineage formula to allow efficient processing, similar to the approximate lineage computation of Re et al. [25].

The central idea behind this technique is to use samples of the probability distributions and pass them around as messages, instead of the complete pdfs. We modify the Eager and the Eager+Order message passing algorithms of Section 3 in order to support sampling techniques.

The algorithm we present is a recursive algorithm on the junction tree. We illustrate it with a simple example. Suppose that we want to compute the probability of $(a \wedge k) \vee h$ in the Junction tree of Figure 2(iii). We construct the Steiner tree corresponding to a, k and h as before and also select a pivot node. Suppose we select clique (ac) as pivot. In the first step, the clique (ac) constructs N samples from the pdf $p(a, c)$ and sends it to clique (cfg). The clique (cfg) now computes $p(g, f|c)$, (dividing $p(g, f, c)$ by $p(c)$) and uses the samples (c_i) from (ac) to generate samples (g_i, f_i) from $p(g, f|c)$. Now clique (cfg) recursively sends samples g_i to clique (gjk) and samples f_i to clique (fh). These cliques, return samples corresponding to k and h respectively to clique (cfg). Now (cfg) combines these samples and returns them (c_i, h_i, k_i) to clique (ac), which now evaluates the probability of $(a \wedge k) \vee h$.

6. IMPLEMENTATION DETAILS

Our implementation is part of an ongoing effort to build a scalable, general-purpose probabilistic database system called *PrDB* that allows manipulating correlations and uncertainty flexibly. The lineage processing component is implemented in Java using JDK 1.5. We use a MySQL database to store the data and the correlations. In addition to the indexing and query processing components, our system contains an input parser for users to declaratively insert data and the correlations in the database. The parser allows users to define correlations as factor objects and also insert them against corresponding tuples in the database. For instance, the statement: `define factor` mutex (0 0 0.4, 0 1 0.3, 1 0 0.3, 1 1 0) defines a mutual exclusion factor with the given probabilities. We can then insert this factor on tuples x_1 and x_2 (Section 1) using the statement `insert factor` mutex `in` CarAds `on` $x_1.E$, $x_2.E$. In fact, we can also perform the above statement on multiple tuples by including a `<where>` predicate: `insert factor` (1 0.8, 0 0.2) `in` CarAds `on` E `where` VIN = 2B1.

We use a simple query rewrite to track the lineages of the result tuples of a conjunctive query by exploiting the `concat` and the `group_concat` constructs of SQL. Given an SPJ query such as, `SELECT <S> FROM <F> WHERE <W>`. We rewrite this query as:

```
SELECT <S>, GROUP_CONCAT (SEPARATOR '+') AS E
FROM { SELECT <S>, CONCAT(t1.E,'*',t2.E,'*',.. AS E
       FROM <F> WHERE <W> }
GROUP BY <S>
```

Here, t_i's are the tables that are contained in the `FROM` clause of the input query. We also assume that each of the input tables has an attribute named "E" that represents the random variable corresponding to its tuple uncertainty. The boolean expression output by the above rewriting is not in a *read-once* format. Hence, we use the co-graph recognition algorithm of Golumbic et al [12] to rewrite a boolean formula as a read-once function. This algorithm is known to be complete, i.e., if the boolean formula has a read-once form, then it will find one such representation.

7. EXPERIMENTAL EVALUATION

The main objectives of our experimental analysis are to show the benefits of (1) our heuristics for processing lineage formulas over junction trees, (2) our lineage planning algorithms to improve lineage processing times, (3) our approximation algorithms to generate accurate results. We begin with a discussion of the experimental setup.

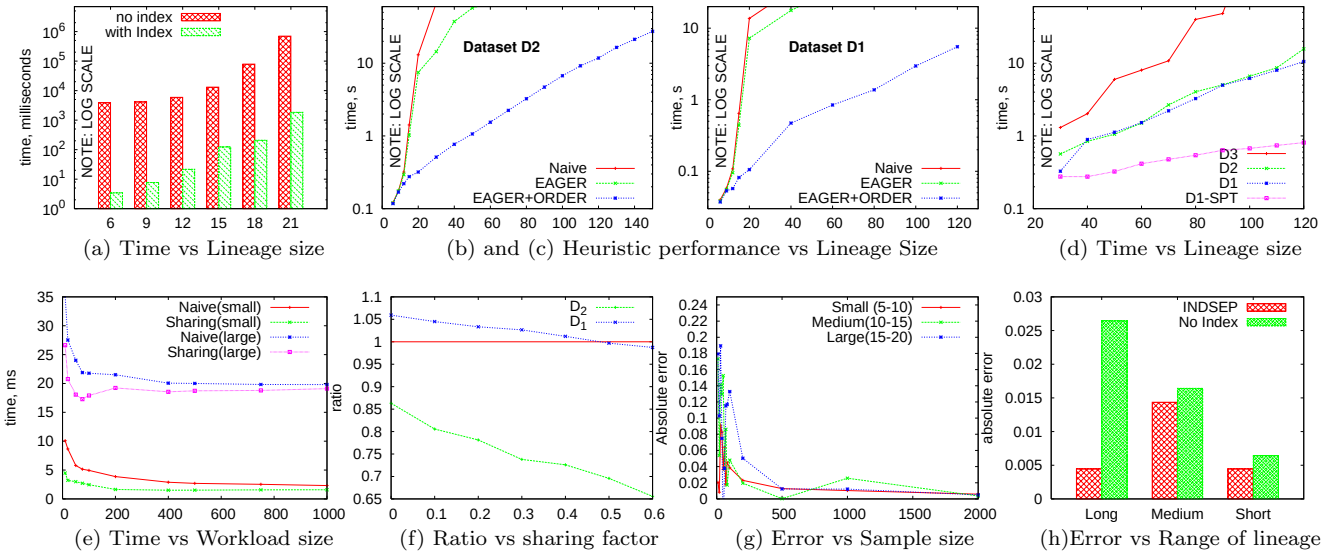
Dataset: We generated a synthetic dataset with 3 relations R_1, R_2, R_3, each of size 100,000 tuples that correspond to the Car Ads application (Section 1). All tuples are uncertain. In addition, each tuple in the relation was correlated with a random number (between 2-10) of other tuples in the database. The correlations are randomly generated factors that correspond to the conditional probability distributions. After populating the database, we build the junction forest (set of junction trees) and use the algorithms from [18] to construct the INDSEP data structure. To generate databases with varying amounts of correlations, we vary the sizes of the connected partitions in the junction tree. We generate 3 different datasets, each with different partitioning sizes. Database D_1 has single node partitions, i.e., it is a tuple independent database. D_2 has partitions of size 100, i.e., it is lightly correlated. D_3 is a *moderately correlated* database with partitions of size 1000. In addition, we generate a Markov sequence denoted by M [17], which is a single junction tree.

Query Workload: We used the query Q from the introduction as part of our experiments. We carefully vary the data in the join columns in order to create lineages of different sizes. We also generate artificial lineages of the form $A_1 \wedge A_2 \wedge A_3$, where each A_i is a disjunction of input random variables, to illustrate the results.

Our most important experimental findings are as follows:

Suitability of INDSEP for Lineage queries: In this experiment, we evaluate the benefits of INDSEP for processing lineage queries. As noted before, INDSEP is extremely useful for *inference* and *aggregation queries*. We run the lineages of various sizes on the Markov sequence database M alternatively, using: (a) INDSEP (b) directly on the underlying junction tree. We measured the wall-clock times for processing the lineage formulas for both cases. The bar graph containing these results is shown in Figure 8(a). The results are plotted as a function of the size of the formula. We observe exponential decrease in the lineage processing time. Note that the y-axis is in *log* scale, so the benefits of INDSEP are more substantial than apparent.

Performance of the heuristics: We now evaluate the performance of our heuristics Eager and Eager+Order for evaluating boolean formulas over a junction tree as opposed to the Naive approach. We used both the heuristics alternatively to bottom out the recursion (Section 4). We used both D_1 and D_2 for this experiment. We compare the time taken to evaluate a lineage formula for both the heuristics and the naive approach as a function of the size of the formula. We used lineages of different sizes varying from [10,150] for the experiment. The time taken for lineage processing is plotted in Figures 8(b) & (c). Note that the y-axis is in log-scale. As shown in the figures, the Naive and the Eager approaches perform very poorly as compared to our Eager+Order heuristic. Even for small lineages below 30, the



(a) Time vs Lineage size (b) and (c) Heuristic performance vs Lineage Size (d) Time vs Lineage size

(e) Time vs Workload size (f) Ratio vs sharing factor (g) Error vs Sample size (h)Error vs Range of lineage

Figure 8: Results: (a) Processing lineages using INDSEP is more scalable. (b) and (c) Illustrating benefits of EAGER+ORDER heuristic over the EAGER and naive approaches for datasets D_2 and D_1. (d) As the correlations increase, lineage processing times increase. Special purpose technique (SPT) performs better for the independent dataset D_1. (e) and (f) Illustrating benefit of batch lineage processing. (g) Sampling errors can be reduced by increasing number of samples. (h) INDSEP improves the quality of our approximations significantly.

amount of processing time is very large, since very large intermediate pdfs are created by the heuristic. In contrast, even for large lineage formula, the heuristic Eager+Order performs very efficiently as shown in the figure (taking less than 0.4 seconds for lineage of size 30 and about 7 seconds for lineage of size 100).

Study of Lineage Processing Performance: Now, we illustrate the performance of our lineage processing algorithm as a function of the size of lineage. For each of the databases D_1, D_2 and D_3, we evaluate the time taken for exact lineage processing (we set the lwidth threshold to be ∞) for different lineage sizes. The times are plotted in Figure 8(c). As shown in the figure, the lineage processing time increases as the database becomes more correlated owing to the larger intermediate pdfs generated during lineage processing. As we can see from the figure, once the lwidths generated exceed 20, (the intermediate pdfs = $8MB$) the algorithm is largely unusable. The exponential blow-up due to the large intermediate pdfs is evident from the figure. In addition, the figure also shows the time taken by a special purpose technique such as Mystiq [8] and Sen et al. [28] to process lineages on the dataset D_1. Our algorithm is quadratic in the size of the formula, while special purpose techniques are linear. Not surprisingly, our system performs poorly for tuple independent probabilistic databases.

Lineage Planning (Batch Lineages): With this experiment, we show the benefits of processing multiple lineages together using our batch processing algorithm (Section 5.1). We use correlated database D_2 for our experiments and we measure the time taken to process a workload using the batch lineage processing algorithm and for comparison, we process each lineage in the workload individually. We compare the performance for random chosen small lineages (< 20) and large lineages (> 50). The results are plotted in Figure 8(e). As shown in the figure, the time taken by the batch lineage processing algorithm is less than the time taken for processing each lineage separately. Notice that even for randomly generated lineages without any explicit sharing, we obtain a significant reduction in the lineage processing time. As the workload sizes get very large > 1000, we gradually lose the benefits of the batch lineage processing. This is because of the overheads that arise in removing duplicates. Next, we evaluate the performance of the batch lineage processing algorithm as a function of the amount of sharing that is present between the lineages. We create workloads with a sharing coefficient ranging from 0.0 (no overlapping variables) to 0.6 (60% repetitions of variables) and evaluate them on datasets D_1 and D_2. We plot the ratio of the time taken for batch lineage processing to the time taken for processing each lineage separately (Smaller ratios are better). As shown in Figure 8(f), for both D_1 and D_2, the lineage processing time reduces as the sharing coefficient increases. Also, we find that the batch lineage processing is more beneficial for correlated datasets than for completely independent datasets.In fact, it induces an overhead for D_1 at low sharing due to the blowup of the workload size at the start of the algorithm. We also found that the batch processing algorithms is quite beneficial for sharing across inference queries.

Approximate Lineage Processing: Now, we evaluate the accuracy of our approximate lineage processing algorithms. We used the Markov sequence dataset M. We first compute the error in the output probability as a function of the number of samples used in our Monte Carlo algorithm. We compute the exact probabilities by setting the lwidth threshold to ∞. Since we were using the fully connected junction tree, we had to limit the size of the formula to less than 20. We used three workloads depending on the size of formula - (5-10), (10-15) and (15-20). The results are shown in Figure 8(h). As shown in the figure, for each of the three workloads, we observe that the accuracy rate improves with the number of samples. In fact, when we use more than 500 samples, we only notice a difference in the second decimal position. With 2000 samples, we notice an error in the third decimal position. Next, we illustrate the benefit of using INDSEP for our approximation. We first run the approximation algorithm directly on the underlying junction tree and then using INDSEP(1000 samples). We

used three kinds of lineage formula, short range, medium range and long range lineages - which have varying spans on the underlying junction tree. Span of a lineage is the size of the Steiner tree induced by the lineage formula. As shown in Figure 8(i), the amount of error using INDSEP is smaller than the error without it. This is due to 2 reasons: First, this is because of selectively approximating only portions of the formula that lead to large lwidth. Second, due to the shortcut potentials in INDSEP, the size of the effective junction tree is small. Also, the difference in the errors is more pronounced for long range lineages. This is because the errors continuously add up sequentially for large span lineages when we process the lineage without INDSEP.

8. CONCLUSIONS

In recent years, there has been a significant increase in the amount of large scale correlated probabilistic data being generated by two sources: Increasing use of machine learning and data mining techniques to understand data on the web, and the expanding use of low cost sensing devices and other measurement infrastructure. Thereby, efficiently processing queries over such data is becoming a significant research task. In this paper, we develop techniques for evaluating conjunctive queries over lightly correlated probabilistic databases. While the general problem is #P-complete, we develop algorithms to characterize the complexity of evaluating a query. For queries with low complexity, we develop techniques based on message passing algorithms over the junction forest corresponding to the probabilistic database. We develop Gibbs sampling based algorithms for evaluating queries that have very large complexity. Further, we scale our algorithms to very large scale probabilistic data using the previously proposed *INDSEP* index data structure. Our experimental results demonstrate the benefits of our approaches for conjunctive query evaluation over correlated probabilistic databases.

Acknowledgement

This work was supported by NSF Grants IIS-0546136 and IIS-0916736. We also thank Prithviraj Sen and Lise Getoor for valuable feedback.

9. REFERENCES

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

[2] A. Chechetka and C. Guestrin. Efficient principled learning of thin junction trees. In *NIPS*, 2007.

[3] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.

[4] A. Choi and A. Darwiche. An edge deletion semantics for belief propagation and its practical impact on approximation quality. In *AAAI*, 2006.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2001.

[6] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhater. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[7] N. Dalvi et al. A web of concepts. In *PODS*, 2009.

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

[9] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, 2007.

[10] J. Finn and J. Frank. Optimal junction trees. In *UAI*, 1994.

[11] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE PAMI*, 1984.

[12] M. C. Golumbic, A. Mintz, and U. Rotics. Factoring and recognition of read-once functions using cographs and normality. In *DAC*, 2001.

[13] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, 2006.

[14] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Int. J. Approx. Reasoning*, 1996.

[15] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.

[16] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *ICDE*, 2008.

[17] B. Kanagal and A. Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *ICDE*, 2009.

[18] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *SIGMOD*, 2009.

[19] U. Kjærulff. Hugs: Combining exact inference and gibbs sampling in junction trees. In *UAI*, 1995.

[20] J. Letchner, C. Re, M. Balazinska, and M. Philipose. Access methods for markovian streams. In *ICDE*, 2009.

[21] E. Michelakis, R. Krishnamurthy, P. J. Haas, and S. Vaithyanathan. Uncertainty management in rule-based information extraction systems. In *SIGMOD*, 2009.

[22] M. A. Paskin. Thin junction tree filters for simultaneous localization and mapping. In *IJCAI*, 2003.

[23] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[24] C. Re, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, 2008.

[25] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *PVLDB*, 2008.

[26] N. Robertson and P. D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1), 1984.

[27] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, 2008.

[28] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[29] P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. In *VLDB*, 2008.

[30] The RFID Ecosystem, University of Washington. http://rfid.cs.washington.edu/.