Local Structure and Determinism in Probabilistic Databases

Theodoros Rekatsinas
University of Maryland
College Park, MD, USA
thodrek@cs.umd.edu

Amol Deshpande
University of Maryland
College Park, MD, USA
amol@cs.umd.edu

Lise Getoor
University of Maryland
College Park, MD, USA
getoor@cs.umd.edu

ABSTRACT

While extensive work has been done on evaluating queries over tuple-independent probabilistic databases, query evaluation over correlated data has received much less attention even though the support for correlations is essential for many natural applications of probabilistic databases, e.g., information extraction, data integration, computer vision, etc. In this paper, we develop a novel approach for efficiently evaluating probabilistic queries over correlated databases where correlations are represented using a *factor graph*, a class of graphical models widely used for capturing correlations and performing statistical inference. Our approach exploits the specific values of the factor parameters and the determinism in the correlations, collectively called *local structure*, to reduce the complexity of query evaluation. Our framework is based on *arithmetic circuits*, factorized representations of probability distributions that can exploit such local structure. Traditionally, arithmetic circuits are generated following a compilation process and can not be updated directly. We introduce a generalization of arithmetic circuits, called *annotated arithmetic circuits*, and a novel algorithm for updating them, which enables us to answer probabilistic queries efficiently. We present a comprehensive experimental analysis and show speed-ups of at least one order of magnitude in many cases.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; H.2.m [**Database Management**]: Miscellaneous; B.2.m [**Arithmetic and Logic Structures**]: Miscellaneous; G.3 [**Mathematics of Computing**]: [Probability and Statistics]

General Terms

Algorithms, Design, Management, Performance

Keywords

Probabilistic Databases, Arithmetic Circuits, Query processing

1. INTRODUCTION

An increasing number of applications are producing large volumes of uncertain data, fueling an interest in managing and querying such data using *probabilistic databases*. Examples of such applications include information extraction [20], data integration [17], sensor networks [16], object recognition [27], and OCR [26], to name a few. This has led to much work on extending relational databases to support uncertainty, and on efficiently evaluating different types of queries over such databases (see [40] for a recent survey).

Query evaluation over probabilistic databases is unfortunately known to be $\#P$-hard even under tuple-independence assumptions for perhaps the simplest class of queries, namely, conjunctive queries without self-joins [10]. To overcome this limitation, a number of different approaches have been explored that can be categorized, at a high level, into *extensional* approaches and *intensional* approaches. In an extensional approach, the query evaluation process is guided solely by the query expression, and query operators are extended to directly compute the corresponding probabilities. For example, the probability of a join result tuple $s \bowtie t$ is the product of the probabilities of the tuples s and t. When such extensional evaluation is possible, the query can be evaluated in polynomial time, hence, much research has focused on characterizing datasets, queries, and query plans for which extensional methods can be correctly applied [9, 30, 21, 31]. On the other hand, in an intensional approach, the intermediate tuples generated during query execution and the final result tuples are associated with propositional symbolic formulas (often called *lineage expressions*) over a subset of the random variables corresponding to the base input tuples. One of several general purpose inference algorithms can then be used to compute the result tuple probabilities, either exactly, e.g., using Shannon expansion [30], variable elimination [38], etc., or approximately [25, 32, 34], depending on the complexity of the lineage expression and the uncertainty model.

Extensional methods scale well but the class of queries for which they can be applied correctly is relatively small. Furthermore, if we relax the tuple-independence assumption and allow representation of correlated data, then extensional methods cannot be directly applied. Being able to handle correlations is critical for probabilistic databases because many of their natural applications require it. These include applications such as information extraction, data integration, computer vision applications, sensor networks, etc., where heavy use of machine learning techniques naturally results in complex correlations among the data. Hence, over the last few years, several probabilistic database systems have been proposed that can manage such correlated databases [37, 41, 42], with correlations typically captured using *graphical models* such as *factor graphs* or *Bayesian networks* [15]. However, intensional approaches that can process queries over such correlated databases and handle more general classes of queries, are typically much slower than extensional approaches and have poor scalability, leading to a significant efficiency gap between the two approaches [21].

In this work, we aim to increase the efficiency of intensional methods by developing a framework that represents correlations using factor graphs [37, 42], and can exploit *context-specific independence* and *determinism* in the correlations, collectively referred to as *local structure* [13]. Context-specific independence [3, 43], often observed in practice, refers to independences that hold given a specific assignment of values to certain variables. Determinism in the correlations, i.e., assigning zero probability to some joint variable assignments, typically manifests in uncertainties involving logical constraints, e.g., mutual exclusion, implications, etc. Exploiting such local structure enables probabilistic inference to run efficiently in many scenarios, where the standard inference techniques such as variable elimination are not feasible [13]. Our framework builds upon the notion of an *arithmetic circuit* (AC) [13], which is a compiled and compact representation of a factor graph that can effectively exploit local structure to drastically reduce online inference times [5]. To our knowledge, there is no prior work on either modifying ACs directly or on computing probabilities of Boolean formulas over them. The highly-compact compiled form of ACs makes it a non-trivial challenge to support either of these efficiently. Hence, we introduce *annotated arithmetic circuits* (AACs), an extension where we add variable annotations on the internal operation nodes of an AC, and develop a novel algorithm for *merging* two AACs to, in essence, combine the uncertainties captured by the AACs. For evaluating conjunctive queries over an AAC-representation of a probabilistic database, we represent the resulting lineage formulas using *ordered binary decision diagrams* (OBDDs), suggested in prior work [30]. However, the AAC-representation of the database imposes significant constraints on how OBDDs can be generated, requiring us to develop new algorithms for this task.

Our approach can be seen as generalizing the prior work on what we refer to as *instance optimal query evaluation*, i.e., optimizing the evaluation of a particular query on a given dataset (with or without correlations). Using AACs enables us to exploit not only the conditional independences in the uncertainty, but also the local structure widely observed in practice. On the other hand, representing the lineage expressions as OBDDs enables us to utilize the structure and the regularities within the lineage expressions themselves, e.g., read-once lineage expressions can be evaluated efficiently on tuple-independent databases [30, 38]. In fact, for tuple-independent databases our approach reduces to the OBDD-based approach of Olteanu et al. [30], whereas for correlated databases without any local structure it reduces to the *junction tree*-based approach of Kanagal et al. [24, 23].

The main contributions of this paper are as follows:

- We introduce a new exact probabilistic query evaluation framework for supporting arbitrary correlations among the stored tuples, that exploits local structure for efficiency and generalizes several prior techniques for instance-optimal query evaluation.
- We introduce an extension of arithmetic circuits called annotated arithmetic circuits (AACs), that enables us to answer probabilistic queries efficiently. To our knowledge this is the first work that addresses the problem of evaluating complex queries over arithmetic circuits.
- We introduce a novel algorithm for incrementally updating AACs by exploiting the annotations present in the internal operation nodes of the circuit. Our merging algorithm allows us to incorporate new correlations introduced over a subset of tuples into the correlations already present in the database, without recompiling the existing arithmetic circuits.
- We experimentally verify the performance of our algorithm for both tractable and hard queries over tuple-independent databases and correlated databases based on the TPC-H benchmark. We observe a speed-up of at least one order of magnitude over variable elimination. Moreover the performance of our algorithm is similar to that of other intensional methods based on Shannon decomposition.

2. PRELIMINARIES

In this section we present a short review of probabilistic databases and arithmetic circuits.

2.1 Probabilistic Databases

A probabilistic database can be defined using the *possible world semantics* [10]. Let $\mathcal{R}$ be a set of relations, $\mathcal{X} = \{X_1, \cdots, X_n\}$ be a set of random variables associated with the tuples or attributes stored in the database (these could either be binary random variables capturing tuple existence uncertainty, or discrete random variables capturing attribute value uncertainty), and Φ be a joint probability distribution over $\mathcal{X}$. A probabilistic database D is defined to be a probability distribution over a set of deterministic databases (or possible worlds) $\mathcal{W}$ each of which is obtained by assigning $\mathcal{X}$ a joint assignment $\mathbf{x} = \{X_1 = x_1, \ldots, X_n = x_n\}$ such that $x_i \in \text{dom}(X_i)$. The probability associated with a possible word obtained from the joint assignment $\mathbf{x}$ is given by Φ.

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. Furthermore, the marginal probability of each result t in the union is obtained by summing the probabilities of the possible worlds $W_t \subseteq W$ that return t: $\Pr(t) = \sum_{w \in W_t} \Pr(w)$.

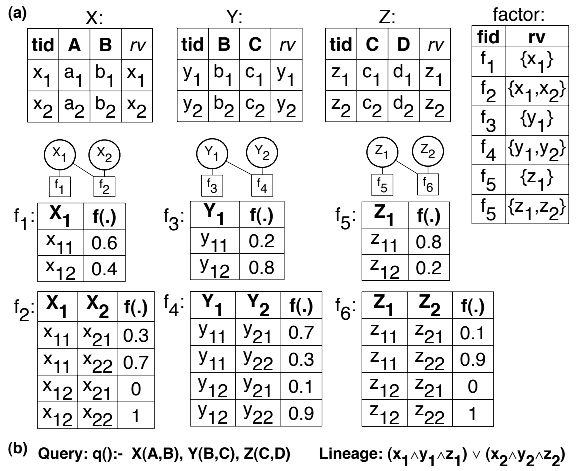
<u>**Representation:**</u> Typically, we are not able to represent the uncertainty in the dataset using an explicit listing of the joint probability distribution Φ. Instead more compact representations need to be used. The different representations differ in their expressibility and the complexity of query evaluation. The simplest representation associates *tuple existence* probabilities with individual tuples, and assumes that the tuple existences are independent of each other. However, most real-world datasets contain complex correlations, therefore, making an independence assumption can lead to over-simplification and large errors [36, 37].

Instead, we use a general and flexible representation of a probabilistic database, proposed by Sen et al. [36] and also used by Wick et al. [42], that can capture complex correlations among the tuples or the attributes in the database through use of *factor graphs*, a class of graphical models that generalizes both directed Bayesian networks and undirected Markov networks. More formally we have:

***Definition** 1.* A *factor* $f : \text{dom}(X_1) \times \text{dom}(X_2) \times \cdots \times \text{dom}(X_m) \to R^+$ is a function over a set of random variables $\mathbf{X} = \{X_1, X_2, \ldots, X_m\}$ such that $f(\mathbf{x}) \geq 0, \forall \mathbf{x} \in \text{dom}(X_1) \times \cdots \times \text{dom}(X_m)$. The set of variables $\mathbf{X}$ is called the scope of the factor and denoted $\text{Scope}[f]$.

***Definition** 2.* A factor graph $\mathcal{P} = (\mathcal{F}, \mathcal{X})$ defines a joint distribution Φ over the set of random variables $\mathcal{X}$ via a set of factors $\mathcal{F}$, where $\forall f(\cdot) \in \mathcal{F}$, $\text{Scope}[f] \subseteq \mathcal{X}$. Given a complete joint assignment $\mathbf{x} = \{X_1 = x_1, \ldots, X_n = x_n\}$ such that $x_i \in \text{dom}(X_i)$, the joint distribution is defined by $\Phi(\mathbf{x}) = \Pr(\mathbf{x}) = \frac{1}{\mathcal{Z}} \prod_{f \in \mathcal{F}} f(\mathbf{x}_f)$ where $\mathbf{x}_f$ denotes the assignments restricted to $\text{Scope}[f]$ and $\mathcal{Z} = \sum_{\mathbf{x}' \in \mathcal{X}} \prod_{f \in \mathcal{F}} f(\mathbf{x}'_f)$.

This leads us to a formal definition of a probabilistic database:


(a)
X:
Y:
Z:
factor:
x1
x2
f1
f2
y1
y2
f3
f4
z1
z2
f5
f6
(b) Query: q():- X(A,B), Y(B,C), Z(C,D)
Lineage: $(x_1 \wedge y_1 \wedge z_1) \vee (x_2 \wedge y_2 \wedge z_2)$

X:

tid	A	B	*rv*
x_1	a_1	b_1	x_1
x_2	a_2	b_2	x_2

Y:

tid	B	C	*rv*
y_1	b_1	c_1	y_1
y_2	b_2	c_2	y_2

Z:

tid	C	D	*rv*
z_1	c_1	d_1	z_1
z_2	c_2	d_2	z_2

factor:

fid	rv
f_1	$\{x_1\}$
f_2	$\{x_1,x_2\}$
f_3	$\{y_1\}$
f_4	$\{y_1,y_2\}$
f_5	$\{z_1\}$
f_5	$\{z_1,z_2\}$

f_1:

X_1	f(.)
x_{11}	0.6
x_{12}	0.4

f_3:

Y_1	f(.)
y_{11}	0.2
y_{12}	0.8

f_5:

Z_1	f(.)
z_{11}	0.8
z_{12}	0.2

f_2:

X_1	X_2	f(.)
x_{11}	x_{21}	0.3
x_{11}	x_{22}	0.7
x_{12}	x_{21}	0
x_{12}	x_{22}	1

f_4:

Y_1	Y_2	f(.)
y_{11}	y_{21}	0.7
y_{11}	y_{22}	0.3
y_{12}	y_{21}	0.1
y_{12}	y_{22}	0.9

f_6:

Z_1	Z_2	f(.)
z_{11}	z_{21}	0.1
z_{11}	z_{22}	0.9
z_{12}	z_{21}	0
z_{12}	z_{22}	1

Figure 1: (a) A probabilistic database where the uncertainty is represented with factors. (b) The lineage corresponding to a conjunctive query.

***Definition** 3.* A probabilistic database D is a pair $(\mathcal{R}, \mathcal{P})$ where $\mathcal{R}$ is a set of relations and $\mathcal{P}$ denotes a factor graph defined over the set of random variables associated with the tuples in $\mathcal{R}$.

We require that the joint distribution defined by the factor graph satisfy certain normalization constraints, i.e., the partition function $\mathcal{Z} = 1$. This does not imply a limitation on the applicability of the proposed framework but is only used for ease of representation. Figure 1(a) shows an example probabilistic database represented using factors, along with the factor graphs themselves (which contain nodes for each factor and each random variable, and a factor is connected to all variables that it is defined over). We assume we only have tuple existence uncertainties, and the Boolean random variables corresponding to the tuple existences are associated with the tuples $(x_1, x_2, \cdots)$. In the remainder of the paper, for any Boolean random variable x we will denote its **true** and **false** assignment with x_1 and x_2 respectively. The factors are stored separately. In this example, we have six factors, $f_1, \cdots, f_6$. The random variables over which they are defined are stored in a separate table (called *factor*). As an example, we have two factors containing variable x_1, namely, f_1 and f_2, the latter of which is a joint factor over x_1 and x_2. The joint probability distribution over all variables is defined as:
$\Phi = f_1(X_1)f_2(X_1, X_2)f_3(Y_1)f_4(Y_1, Y_2)f_6(Z_1)f_6(Z_1, Z_2)$.

Querying a probabilistic database: Executing an SQL query over a probabilistic database efficiently has been a subject of much research over the last decade in the database community, and as we discussed earlier, the approaches can roughly be divided into *extensional approaches* and *intensional approaches*. With our focus on correlated databases, we are restricted to using an intensional approach. In intensional methods the relational operators are extended to build a Boolean formula (called *lineage*) for each intermediate tuple and each result tuple generated during query evaluation (Figure 1(b)). The marginal probability of a result tuple can now be obtained by computing the probability of the corresponding Boolean formula evaluating to **true**, which is #P-hard.

Next, we review some of the intensional query evaluation techniques that have been proposed in the literature.

Variable Elimination (VE)-based Approach: In the VE-based approach [36], instead of constructing a lineage formula for each result tuple, we construct an equivalent representation as a factor graph where each intermediate tuple is explicitly represented (see Figure 2(a)). For each intermediate tuple, we add an appropriate factor containing the tuple and the tuples that generated it. For instance, for tuple i_1 generated by joining tuples x_1 and y_1, we introduce an AND factor that captures the logical constraint that i_1 is **true** iff x_1 and y_1 are both **true**. Similarly, for tuple i_5, we add an OR factor capturing the logical constraint that i_5 is **true** if either i_2 or i_4 is **true** (corresponding to a *project* operation). Query evaluation is now equivalent to performing inference on this factor graph to compute the marginal probability distribution of i_5.

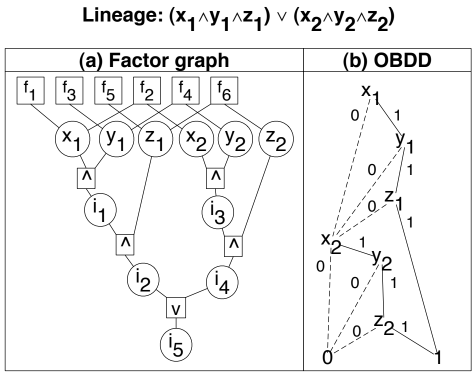


Figure 2: A query over the probabilistic database in Figure 1. (a) The factor graph for the query with AND and OR factors. (b) The OBDD for the lineage of the result tuple when the random variables are independent.

Variable elimination [14] is a simple and widely-used technique for performing inference over factor graphs. In essence, VE operates by eliminating one variable at a time from the factor graph until we are only left with the variable of interest (in this case, i_5). For this purpose, a *variable ordering* needs to be chosen *a priori* to specify the order in which to eliminate the variables. At each iteration two operations are performed: Let X be the variable under consideration. All factors that refer to X are *multiplied* to get a new factor f' and then X is *summed out* of f' to get a factor f'' with no reference to X. As an example, if we choose to eliminate x_1 in the first step, we would multiply the factors f_1, f_2, and the AND factor on x_1, x_2, i_1 to get a factor on x_1, y_1, i_1, x_2, and sum-out x_1 to get a new factor on y_1, i_1, x_2. The complexity of the inference procedure is exponential in the size of the largest factor (measured as the number of variables) created during the process, which is at least the *treewidth* of the factor graph. However, finding the optimal variable ordering is NP-hard and heuristics are typically used.

Sen et al. [37] introduced a *lifted* inference technique that exploits the symmetry in the probabilistic database to reduce the complexity of query evaluation. Our work is orthogonal to their proposal of exploiting symmetry, and it is an interesting future direction to see how these two can be combined.

OBDD-based approach: In a different approach, Olteanu et al. [30] focus on tuple-independent databases and explore the connection between *ordered binary decision diagrams* (OBDDs) [4] and query evaluation for a large class of queries ranging from conjunctive queries with safe plans, to hard queries on restricted databases. OBDDs are rooted, directed acyclic graphs that compactly represent Boolean formulas. They consist of decision nodes and two terminal nodes, called 0-terminal and 1-terminal. Each decision node is labeled with a Boolean variable and has two children, one for each instantiation of the corresponding Boolean variable. Dashed edges represent the assignment of the variable to **false**, while solid edges represent the assignment to **true**. Finally, the two terminal nodes represent the value of the Boolean formula for a particular variable assignment defined by a path from the root node to that terminal node. In the worst case, an OBDD may be a complete binary tree with exponential size, but since it can exploit the structure of the Boolean formula, it is typically much more compact. Figure 2(b) shows the OBDD corresponding to the lineage formula in Figure 1(b).

Under the tuple-independence assumption, given the OBDD of a lineage formula, each edge can be annotated with the probability of its source decision node taking the corresponding value. The probability of any non-terminal node is computed as the sum over the probabilities of its children, weighted by their corresponding edge probabilities. One can, therefore, compute the probability that the lineage formula evaluates to true by traversing all bottom-up paths from the 1-terminal node to the root, multiplying the probabilities along the way, and then summing the products. This can be done in time linear in the size of the OBDD, hence, when the lineage formula results in an OBDD of polynomial size, the query can be evaluated efficiently.

However not all Boolean formulas admit an OBDD of polynomial size. In fact, OBDD construction is also driven by a variable ordering which dictates the order in which the variables are evaluated and corresponds to the top-down order of decision nodes in the final OBDD. Choosing the optimal variable ordering is NP-hard. A comprehensive review of different construction techniques is presented by Mantadelis et al. [29].

Discussion: All the approaches mentioned above present significant limitations in presence of correlations and local structure. Factor graphs do not exploit the local structure of the factors to reduce the complexity of inference. Moreover, OBDDs are applicable only under the tuple-independence assumption. In the next section we present an approach that combines the representational power of factor graphs with the compactness of decomposition methods leading to more efficient query evaluation over correlated databases.

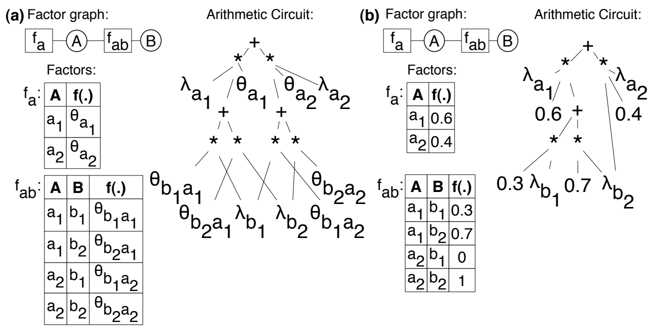
2.2 Arithmetic Circuits

In this section we briefly review how context-specific independence and determinism can be exploited to enable efficient exact inference even in factor graphs with high treewidth, through use of arithmetic circuits [5, 6, 7, 8]. Context-specific independence is prevalent in relational domains, since the underlying structure introduces regularities and conditional independencies that are true only under specific contexts. Furthermore, determinism appears during query evaluation, where every relational operator introduces deterministic constraints over its input tuples, e.g., both input tuples of a join must exist in order for the intermediate tuple to exist. One can also consider the constraints introduced by foreign keys as another source of deterministic correlations. Exploiting such determinism is important for improving the efficiency of probabilistic query processing.

Let $\Phi(\cdot)$ be the joint distribution over a set of random variables $\mathcal{X}$ defined by a factor-graph. We associate Φ with a unique multilinear function (MLF) [12] over two types of variables:

- *Evidence indicators:* For each random variable $Y \in \mathcal{X}$ with $\mathrm{dom}(Y) = \{y_1, \cdots, y_n\}$, we have a set of evidence indicators: $\{\lambda_{y_1}, \lambda_{y_2}, \ldots, \lambda_{y_n}\}$, i.e., one evidence indicator for each y_i.
- *Factor parameters:* For each factor f over a set of random variables $\mathbf{X}$, we have a set of parameters $\theta_{\mathbf{X}=\mathbf{x}}$.

For any unobserved random variable, i.e., a random variable whose value is not fixed, all the evidence indicators are set to 1. When a particular value is assigned to a random variable, the indicator corresponding to that value is set to 1 and all other indicators is set to 0. The factor parameters $\theta_{\mathbf{X}=\mathbf{x}}$ correspond to the actual values in the factors of the factor graph. The MLF for a factor graph has an


(a) Factor graph:
f_a — A — f_{ab} — B
Factors:
Arithmetic Circuit:
λ_{a_1} θ_{a_1} θ_{a_2} λ_{a_2}
$\theta_{b_1a_1}$ $\theta_{b_2a_2}$ $\theta_{b_2a_1}$ λ_{b_1} λ_{b_2} $\theta_{b_1a_2}$
(b) Factor graph:
Arithmetic Circuit:
λ_{a_1} 0.6 0.4 λ_{a_2}
0.3 λ_{b_1} 0.7 λ_{b_2}

f_a:

A	f(.)
a_1	θ_{a_1}
a_2	θ_{a_2}

f_{ab}:

A	B	f(.)
a_1	b_1	$\theta_{b_1a_1}$
a_1	b_2	$\theta_{b_2a_1}$
a_2	b_1	$\theta_{b_1a_2}$
a_2	b_2	$\theta_{b_2a_2}$

f_a:

A	f(.)
a_1	0.6
a_2	0.4

f_{ab}:

A	B	f(.)
a_1	b_1	0.3
a_1	b_2	0.7
a_2	b_1	0
a_2	b_2	1

Figure 3: ACs and their factor graphs. Although ACs are DAGs, the directions on the edges are not explicitly drawn. (a) Assuming no local structure, the size of the AC is exponential in the number of variables. (b) Exploiting determinism (i.e., $Pr(A = a_2, B = b_1) = 0$) leads to an AC of smaller size.

exponential number of terms, i.e., one term for each joint assignment of the random variables in $\mathcal{X}$. For example, the factor graph in Figure 3(a), in which all random variables are binary, induces the following MLF:

$$\lambda_{a_1}\lambda_{b_1}\theta_{a_1}\theta_{b_1a_1} + \lambda_{a_1}\lambda_{b_2}\theta_{a_1}\theta_{b_2a_1} + \\ \lambda_{a_2}\lambda_{b_2}\theta_{a_2}\theta_{b_2a_2} + \lambda_{a_2}\lambda_{b_2}\theta_{a_2}\theta_{b_2a_2}$$

Given the MLF for a factor graph, we can compute the probability of evidence, denoted by $\Pr(e)$, i.e., the probability of a specific joint assignment of all (or of a subset of) the random variables, by setting the appropriate evidence indicators to 0 instead of 1 and evaluating the MLF. While the MLF has exponential size, if we can factor it into something small enough to fit within memory, then we can compute $\Pr(e)$ in time linear in the size of the factorization. The factorization will take the form of an arithmetic circuit [13]. More rigorously we have the following definition.

Definition 4. An *arithmetic circuit* (AC) over variables Σ is a rooted, directed acyclic graph whose leaf nodes are either numeric constants or evidence indicators, internal nodes correspond to product and sum operations, and the root node corresponds to the circuit's output. The size of the arithmetic circuit is defined to be the number of its edges.

We elaborate more on the connection between ACs and variable elimination. As mentioned earlier, VE is an algorithm that acts on a set of factors and, driven by a variable ordering, performs two operations at each iteration: First, factors that contain a particular variable are multiplied to create a new factor and, then, that variable is summed out of that factor. An arithmetic circuit can be viewed as the trace of the VE process for a particular factor graph [13].

We refer to the process of producing an AC from a factor graph as *compilation*. One way to do this is to represent the joint distribution by a propositional logical formula in tractable logical form, known as *deterministic, decomposable negation normal form* (d-DNNF), which is then mapped to an AC [11]. Other approaches are based on decision diagrams, and we discuss them in Section 4.2. Jha and Suciu [22] show that d-DNNF is a tractable logical form which subsumes decision diagrams. Therefore, arithmetic circuits can be viewed as a generalization of the decision diagrams used in the intensional probabilistic inference methods presented earlier.

A probability of evidence query is computed by assigning appropriate values to the evidence-indicator nodes and evaluating the circuit in a bottom-up fashion to compute the value of the root. For example, using the AC in Figure 3(b) we can compute the probability of evidence $\Pr(A = a_1, B = b_1)$ by first setting $\lambda_{a_1} = 1, \lambda_{a_2} = 0, \lambda_{b_1} = 1, \lambda_{b_2} = 0$, and then traversing and evaluating the circuit. This process may be repeated for as many probability of evidence queries as desired and it is only linear in the size of the AC. The size of an AC is in the worst case exponential in the treewidth of the factor graph. However, if local structure is present, the size of an AC is often significantly smaller. Figure 3(b) shows one example where the factor value for the joint assignment $A = a_2, B = b_1$ is set to 0, hence, the corresponding sub-circuit is pruned, resulting in a much smaller AC.

3. ARITHMETIC CIRCUITS IN PROBABILISTIC DATABASES

In this section we discuss how arithmetic circuits can be used in correlated probabilistic databases. We begin by discussing a naive approach that uses ACs for inference alone, discuss its limitations, and then present an overview of our proposed approach.

3.1 Naive Approach

Let D denote a probabilistic database and $\mathcal{P}$ the factor graph representing the correlations among the stored data. Consider a query Q against D. Following the factor graph approach (Figure 2(a)), we can construct a new (augmented) factor graph $\mathcal{P}'$ for Q on which inference needs to be performed. Compiling $\mathcal{P}'$ into an arithmetic circuit results in a compact representation of the VE process due to the deterministic intermediate factors introduced. Inference can be performed by parsing the circuit to compute the result probabilities. However, although the inference time in ACs is low, compilation time can be quite expensive. Furthermore, for each different query, the corresponding augmented factor graph needs to be compiled into a new AC. Therefore, such an approach is not a viable means for evaluating queries against probabilistic databases.

Ideally, we would like to avoid repeatedly compiling the base arithmetic circuit, AC_P, from the database. Instead, it would be desirable that we construct AC_P offline once, and save it in the database. Then, for a given query Q, we can construct a new arithmetic circuit, AC_Q, and somehow "merge" the two ACs to get a single AC for computing the query result. However, arithmetic circuits do not support online updates. This significant limitation arises because only the leaf nodes of the circuit provide information about the random variables present in the factor graph through the corresponding evidence indicators. The internal nodes do not have enough information to determine which variables participate in a particular operation. Therefore, it is impossible to merge arithmetic circuits into a unified variable elimination trace, which takes into account all the corresponding correlations.

3.2 Overview of the Proposed Framework

We begin with a brief overview of our proposed query evaluation framework. We elaborate on the steps in the next two sections. We also introduce the running example that we will be using.

Phase 1 - Preprocessing: We assume that a probabilistic database D and the factor graph $\mathcal{P}$ representing the correlations among the stored tuples are given as input to the proposed framework. The factors in $\mathcal{P}$ are represented using ADDs to capture the local structure. Offline, we compile the given probabilistic database into an *annotated arithmetic circuit* (AAC), an extended version of an AC where sum nodes are annotated with the variable on which the summation is performed. The AACs corresponding to the probabilistic database shown in Figure 1 are shown in Figure 4. As depicted, we do not have a single AAC for the entire network. Instead, we require that disconnected parts of the factor graph referring to independent sets of variables correspond to separate AACs. An immediate consequence of this is that a

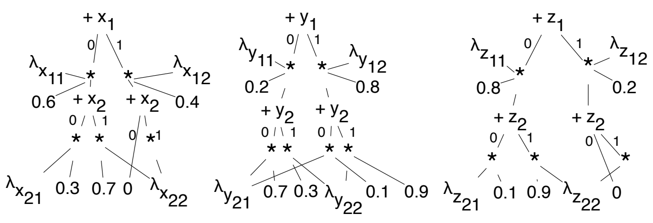


Figure 4: The (complete) AACs corresponding to the probabilistic database shown in Figure 1.

random variable can be present only in one AAC. Maintaining a collection of AACs, denoted by A_{Col}, instead of a single AAC, allows for indexing the AACs, thereby, offering more flexibility while merging them.

Phase 2 - Lineage Processing: Given a query Q, we compute a lineage formula for each result tuple using standard techniques.

Phase 3 - Query Evaluation: During this phase we iterate through the result tuples of the given query Q. For each tuple we perform the following steps:

- **(a)** The lineage formula introduces a set of new deterministic correlations among the random variables present in it. Specifically, the new correlations describe the logical constraints under which the lineage formula evaluates to 1. Then lineage is compiled into a new AAC (called a *lineage-AAC*) that captures the constraints and represents them in a compact way.
- **(b)** The lineage-AAC is merged with the collection of all the AACs that refer to variables present in the lineage-AAC.
- **(c)** The result tuple probability is computed by traversing the resulting merged AAC.

We define and describe AACs in Section 4.1, and describe algorithms to compile the database factor graph and the lineage formula into AACs in Sections 4.2 and 4.3. We then present our merging algorithm for combining multiple AACs in Section 5.

4. ANNOTATED ARITHMETIC CIRCUITS

In this section we introduce annotated arithmetic circuits and show how a correlated probabilistic database can be compiled into a collection of AACs. We also introduce a novel algorithm for representing lineage formulas as AACs.

4.1 Definitions

An annotated arithmetic circuit is a generalization of an arithmetic circuit that includes full information on the sequence of arithmetic operations performed during inference and the variables that participate in them. We maintain this information by adding variable annotations to the internal operation nodes. More formally, we have the following definition:

***Definition** 5.* An *annotated arithmetic circuit* (AAC) over variables Σ is a rooted, directed acyclic graph whose leaf nodes are either numeric constants or evidence indicators, internal nodes correspond to product and sum operations, and the root node corresponds to the circuit's output. Each sum node is *annotated* with the corresponding variable s_i that is being summed out and its outgoing edges are annotated with the values of the different instantiations of variable s_i. The size of the annotated arithmetic circuit is defined to be the number of its edges.

AACs inherit their representational power from regular ACs, and therefore, can capture the local structure and the conditional independences present in a network. Moreover, variable annotations provide the necessary information to detect the exact order of operations performed during variable elimination, as we discuss in the next section. This enables us to detect and directly update the corresponding parts of the circuit when new correlations are introduced and thus plays a key role when merging different arithmetic circuits. To guarantee the correctness of the merging algorithm presented in Section 5, we require that the circuit be a *complete trace* of the variable elimination algorithm.

***Definition** 6.* An AAC over variables Σ is a *complete* trace of variable elimination over variables in Σ, if each evidence-indicator $\lambda_{X=x}$ for a random variable X is preceded by a sum node annotated with variable X and none of its sibling nodes is an evidence-indicator.

When an AAC is a complete trace of VE, sum and product nodes appear in an interleaving manner. Every sum node will be followed by a product node and every operation child of a product node will be a sum node. From now on, we will assume that all AACs correspond to complete traces, and we will drop the qualification.

Examples of AACs corresponding to complete traces of VE are shown in Figure 4. As we can see the annotations in the sum nodes allow us to trace the exact variable that is being summed out. Furthermore if we compare the first AAC presented in Figure 4 with the arithmetic circuit shown in Figure 3, we have that both factor graphs present the same pattern of determinism, however the AAC has an extra sum node in the sub-circuit that expresses the deterministic conditional probabilities. This is necessary because the presented AAC keeps full trace of the VE process. Moreover, when a product node has constant 1 as child, we drop it but the product node itself is kept since it is needed during merging. We note that a complete AAC exploits local structure as a regular AC does, and presents only a small increase in size compared to a regular AC.

4.2 Compiling Factor Graphs into AACs

We compile a factor graph into the corresponding collection of AACs by extending a compilation algorithm introduced by Chavira [6], which is based on variable elimination and *algebraic decision diagrams* (ADDs) [1]. Analogous to how a BDD is a representation of a Boolean function, an ADD is a graph representation of a function that maps instantiations of Boolean variables to real numbers. Our algorithm to compile factor graphs into AACs extends the algorithm described in [6] by adding variable annotations and merging contiguous product nodes, outputting a complete AAC. For brevity, we omit a detailed description here.

However, one important issue that warrants discussion here is the variable ordering used to generate the AACs. Similar to variable elimination and the OBDD construction algorithm, we need to choose an ordering of the variables to compile the database into AACs using the above procedure. Let Π denote the total ordering over all variables that is used to generate the AACs, and let Π_{Col} denote a collection of partial orderings over the disjoint sets of variables corresponding to the different AACs in A_{Col}.

As we will see in the next section, the AAC corresponding to the lineage of a query result tuple must respect all of these partial orderings. This crucial constraint imposed by the AAC merging algorithm means that we cannot use standard algorithms for constructing an AAC from a lineage expression.

4.3 Compiling Lineage Formulas into AACs

Lineage can be represented as a factor graph (Figure 2(a)), and we can use the above algorithm to construct an AAC for it. However, the lineage corresponds to a factor graph limited to consist of only two deterministic factors (AND and OR). Hence, we can employ more efficient techniques based on OBDD construction.

Recall that an OBDD corresponding to a lineage formula is a compact decision diagram representing the set of constraints over the instantiations of the random variables under which the lineage formula evaluates to true (Section 2), and can be constructed by choosing a ordering of the variables in which the variables are evaluated. As discussed above, the variable ordering we choose must respect all the partial orderings in Π_{Col}. However, none of the standard order selection algorithms can be used for our purpose, as they do not take into account the ordering constraints.

Constructing an AAC from an OBDD: The easiest way to construct an AAC for a given lineage formula is to first construct an OBDD, and then modify it by adding the necessary annotated operation nodes with the appropriate indicator constants. Figure 5(b) depicts the OBDD and the AAC (Figure 5(c)) corresponding to a conjunctive query, executed against the probabilistic database shown in Figure 1. The query generates a single Boolean formula. There is a one-to-one mapping between the OBDD and the corresponding AAC; in particular, each decision node is converted into a sum node, and expanded to add a product node and an appropriate evidence indicator. We see that the AAC represents the deterministic correlations introduced by the query. During the lineage-AAC construction we ignore the correlations among the random variables.

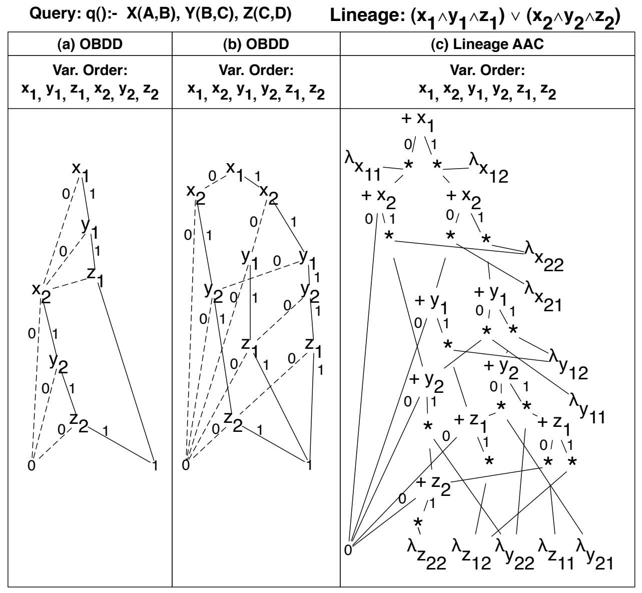

Query: q():- X(A,B), Y(B,C), Z(C,D)
Lineage: $(x_1 \wedge y_1 \wedge z_1) \vee (x_2 \wedge y_2 \wedge z_2)$
(a) OBDD
(b) OBDD
(c) Lineage AAC
Var. Order: $x_1, y_1, z_1, x_2, y_2, z_2$
Var. Order: $x_1, x_2, y_1, y_2, z_1, z_2$
Var. Order: $x_1, x_2, y_1, y_2, z_1, z_2$

Figure 5: (a) The OBDD for the given lineage formula when the variable ordering is generated by a constraint oblivious heuristic. (b,c) An example of the lineage AAC corresponding to the given query when executed against the database shown in Figure 1. The new constraint aware heuristic was used to generate the variable ordering.

Choosing a variable order for the lineage OBDD: The order in which the variables are evaluated in an OBDD plays a crucial role in determining its size. Given a good variable ordering, the size of the OBDD can be polynomial in the size of the corresponding Boolean expression. In the general case, finding the optimal variable ordering for a given Boolean formula is NP-hard [2]. OBDDs have been extensively used in VLSI design and many heuristics that give good variable orderings have been proposed in the corresponding literature [18]. In particular, there are two main guidelines that the proposed heuristics satisfy:

(a) Input variables of a Boolean formula that are connected should appear together in the ordering. For example, consider a

CNF formula where we fix the values of a set of variables that belong to the same clause so that the clause evaluates to zero. The entire formula will then evaluate to zero.

(b) Input variables with higher fan-out should appear sooner in the variable ordering since they have greater influence in the output of the formula. Fixing the values of influential variables first may lead to fixing the values of larger parts of the Boolean formula. By having those variables together in the OBDD, we can significantly reduce the size of the OBDD.

Another advantage of these requirements is that they facilitate *caching.* During OBDD construction the results of intermediate operations are stored in a cache, following a similar rationale as dynamic programming algorithms where the subproblems of the initial problem are cached. This ensures that all intermediate OBDDs are of polynomial size if the final OBDD is of polynomial size.

Partial order constraints: This brings us to the main challenge in constructing an AAC for the lineage formula. Recall that Π_{Col} can be seen as a list of disjoint orderings, each specifying an order over a disjoint subset of the variables. Let Π_{final} denote the variable ordering used to construct the lineage-AAC. We require that Π_{final} satisfy two constraints, one motivated by the merging algorithm (Section 5), and the second motivated by the desire to enable caching in that phase (cf. Section 5). Specifically we require that:

(c) Π_{final} must respect the partial orderings in Π_{Col}. This is necessary since it enables merging multiple AACs that refer to the same set of variables.

(d) Variables that are present in a constraint must be kept together in Π_{final} to enable caching in the merging phase.

Requirement (c) is mandatory for the correctness of the merging algorithm, therefore, we assume that it is always satisfied. We elaborate more on requirement (d). Heuristics that only take into account requirements (a) and (b) may generate a variable ordering that minimizes the size of the lineage-AAC but will not always give a good variable ordering for the final AAC. In particular, disregarding the partial ordering constraints may lead to a variable ordering that does not enable efficient caching during the merging phase, leading to a final AAC of exponential size. As we show in Section 5, caching plays an important role in the performance of our proposed AAC merging algorithm.

Keeping the variables that are present in a constraint together in Π_{final}, enables the detection of parts of the final AAC that refer to disjoint sets of variables, allowing caching of those parts. Interleaving variables present in different partial ordering constraints makes it harder to detect isomorphic sub-graphs of the AAC.

Consider the left part of the OBDD shown in Figure5(a). As mentioned earlier, we require that the final (after merging) AAC respect a global variable ordering, specifically, the one shown in the figure. Observe that during merging, variables y_1 and z_1 will be inserted before x_2, y_2, and z_2, as the latter depend on the former. Since x_2 is independent of z_1, the parts of the left sub-AAC that refer to x_2 and appear in the two sub-AACs corresponding to $z_1 = 0$ and $z_1 = 1$, denoted by $A_{z_1=0}$ and $A_{z_1=1}$ respectively, will be the same. However, caching can not be used since $A_{z_1=0}$ and $A_{z_1=1}$ are not isomorphic. The reason is that the sub-circuits corresponding to z_2 which appear at the end of $A_{z_1=0}$ and $A_{z_1=1}$ are different. Finally, we note that the problem of minimizing the size of an OBDD given order constraints over the input variables is NP-hard because the OBDD construction without any such constraints is NP-hard. Next, we develop a heuristic for this problem.

Variable ordering heuristic: Ideally, we would like to find a variable order that satisfies all four requirements listed above. However, the requirements will usually conflict with each other. We introduce a new heuristic algorithm to generate a good variable ordering Π_{final} that will be used during the construction of both the lineage AAC and the final AAC. The new heuristic is shown in Algorithm 1. It takes as input the ordering constraints defined in Π_{Col} and the lineage formula L and it returns the variable ordering Π_{final}. In Algorithm 1, we represent orderings as vectors.

Let us elaborate on the algorithm. First, the Boolean formula L is converted to its corresponding Boolean circuit C. Assuming that connections between the input variables are only introduced because of the lineage formula, the algorithm starts by detecting groups of connected input variables. This directly addresses requirement (a). Let S_g denote the set of groups. Variables contained in a single group should appear together.

Variables are assigned to groups in S_g by traversing circuit C recursively: for each variable node v in C we examine its siblings, i.e., other nodes that are connected with v via some Boolean operation. If any of these siblings is assigned to a group g, we assign v to g. Otherwise, we create a new group and assign v to it. For each internal operation node o we examine its children nodes. If all of them belong to the same group then we assign o to it, otherwise we assign o to a new one.

The algorithm proceeds by generating an ordering O_g^v among the variables contained in a single group. Variables are ordered in descending order of their fan-out in C. This step complies with requirement (b), stating that more influential variables appear sooner. So far the algorithm is oblivious to the ordering constraints in Π_{Col}. The following steps are introduced to account for them.

Ordering variables within groups only generates partial orderings over the variables. To get a total ordering, it is necessary to impose an ordering O_{Cl} on the groups according to their *position score* (PScore). Motivated by requirement (c), we define the following process: Each input variable is associated with a position score, which is defined to be its position in the corresponding ordering in Π_{Col}. The position score of each group in S_g, is defined to be the average position score of the variables contained in it. The intuition behind this is that variables that appear early in an ordering in Π_{Col} should also appear early in Π_{final}.

Until now the proposed algorithm does not explicitly satisfy requirements (c) and (d), presented above. To address them, the algorithm iterates over the groups and the variables in them. Let v denote the variable under consideration at each step of the iteration. The algorithm finds the ordering constraint Constr_v corresponding to v and appends in the final ordering Π_{final} the set of variables S that contains v and all variables $u \in \mathsf{Constr}_v$ that precede v and are not present in Π_{final}. It is easy to see that both requirements (c) and (d) are satisfied.

In Algorithm 1 we show the append operation with the concatenation symbol $\|$. Notice that some of the variables may not be present in the lineage formula but are necessary in the merging phase as they appear before v in the corresponding ACs.

5. MERGING AACs

In this section, we introduce a new algorithm for merging a lineage-AAC with the corresponding complete AACs in the database. We begin with formally defining the problem.

The merging problem: As before, let A_{Col} denote the collection of AACs produced after the compilation of the database, Π_{Col} the collection of partial orderings over the random variables in the database, and Π_{final} the variable ordering generated by the heuristic algorithm presented in the previous section. The merging algorithm takes as input A_{Col} and the lineage-AAC and generates a new AAC, which is used to evaluate the probability of the lineage.

Algorithm 1 variableOrdering(Π_{Col}: a collection of ordering constraints, L: a lineage formula): returns Variable Ordering

```
1: $C \leftarrow$ transform $L$ into its corresponding Boolean circuit.
2: $Var_L \leftarrow$ get the set of variables present in $L$.
3: $S_g \leftarrow$ Get the set of groups of connected variables for circuit $C$.
4: for $c \in S_g$ do
5:     $O_g^v \leftarrow$ order the variables in $c$ in a decreasing order with respect to their fan-out in $C$.
6:     Assign a position score PScore$_g$ to group $g$.
7: end for
8: $O_{\text{Cl}} \leftarrow$ order the groups in an increasing order by their PScore.
9: $\Pi_{\text{final}} \leftarrow \{\}$
10: for $c \in O_{\text{Cl}}$ do
11:     for $v \in O_g^v$ do
12:         if $v \notin \Pi_{\text{final}}$ then
13:             $\text{Constr}_v \leftarrow$ get the constraint for $v$ from $\Pi_{\text{Col}}$
14:             $u \leftarrow arg \max_{w \in \text{Constr}_v \cap L}$ (Position of $w$ in $\text{Constr}_v$)
15:             $S \leftarrow \bigcup_{w \in \text{Constr}_v, w \preceq u} w$
16:             $\Pi_{\text{final}} \leftarrow \Pi_{\text{final}} \| S$
17:         end if
18:     end for
19: end for
20: return $\Pi_{\text{final}}$
```

The core idea of the algorithm can be simply stated: we traverse the lineage-AAC and all the appropriate database-AACs, i.e., AACs that refer to the variables present in the lineage-AAC, simultaneously by keeping one or more cursors over each of them. At any point, the algorithm considers exactly two AAC nodes, a node from the lineage-AAC and a corresponding node from a database-AAC, and tries to merge them. Since the database-AACs refer to disjoint sets of random variables, a node in the lineage-AAC can only correspond to one database-AAC node. We check whether we can compute the result of the merge operation for the two nodes immediately, otherwise we choose a variable to branch on and recursively perform the merge operation for each instantiation of the variable. The variable is chosen according to Π_{final}. In the remainder of the section, we elaborate on these steps.

Path annotations: When traversing the input AACs, it is important to be able to identify in which path of an AAC a variable appears, since traversing redundant paths will significantly deteriorate performance. In general, a product node can have multiple sum nodes as children. This can happen when two or more variables are conditionally independent given the value of a particular variable. To address this issue, we introduce a new annotation for each variable, which we call the *path annotation* of the variable.

Path annotations are set according to the following process: we start by assigning a path annotation of 0 to the root of each AAC in A_{Col} and then we traverse each AAC in a depth-first manner. If a product node has multiple sum children we extend the path annotation with the count information of each child. Consider for example a product node with two sum nodes as children and a path annotation 0. The annotations of its children will be 0::1 and 0::2.

Path annotations are created offline during the compilation phase. The order in which sum nodes appear in the set of children of a product node is determined by the corresponding ordering Π_{Col}. Thus, for different instantiations of the predecessor variables the children of a product node appear in the same order.

Traversing multiple AACs simultaneously: The merging algorithm traverses all the AACs in a breadth-first or a depth-first manner, by keeping multiple cursors at different sum nodes, and recursively traversing down the children of an appropriate product node. If it is traversing down a product node that has two or more sum children, then multiple cursors are generated pointing to those different sum nodes. At any point, we may have at most as many cursors as the number of variables in the AAC. A key requirement here is to be able to identify first which database-AAC contains a particular variable, and then, along which path the variable may be found in that AAC. We use a simple index for the first purpose, whereas the path annotations are used for the second purpose. For example, let the considered variable have a path annotation of 0::1::1, and let the cursors in the corresponding database-AAC point to 0::1 and 0::2. By comparing the prefixes, we can deduce that the variable will be found under the former cursor.

Merging AACs: We continue our discussion by introducing the merging algorithm (shown in Algorithm 2). The multi-merge operation is implemented recursively, reducing the operation of merging the lineage-AAC with the appropriate AACs, into operations over smaller AACs, until we reach boundary conditions: AACs that correspond to constant nodes.

Let a_1 and v_1 denote the root node of the input lineage-AAC and the variable that is associated with it. The algorithm starts by finding the database-AAC A_d that needs to be merged and selects the appropriate cursor of A_d (using the procedure described above, and encapsulated in the function $\text{getAACCursor}()$). Let a_2 denote the sum node that the selected cursor points to.

If the algorithm is given a trivial input, i.e., a lineage-AAC equal to 0 or 1, or if the result of the multi-merge between a_1 and a_2 is present in the cache, the multi-merge operation terminates immediately. Otherwise we must recursively compute the result of the operation for a_1 and a_2. Recall that v_1 is the variable that corresponds to a_1 and let v_2 be the variable that corresponds to a_2. Because all variables present in the lineage-AAC are already present in an AAC in A_{Col} and all AACs respect Π_{final}, there are only two cases that the algorithm needs to consider: (a) $v_2 \prec v_1$ and (b) $v_2 = v_1$. We also note that in each turn the merging algorithm expands two contiguous levels of the AACs under consideration, exploiting that in a complete AAC sum and product nodes appear in turns. The algorithm proceeds as described below.

Case - Same variables: When both sum nodes refer to the same variable v_1, the merge operation outputs a sum node a annotated with v_1. To construct the children of a the algorithm iterates through all values v in the domain of v_1 and performs the following process: let c and c' denote the child node of a_1 and a_2 respectively that correspond to $v_1 = v$. The algorithm checks for the following terminal cases: (1) if either c or c' are 0 it outputs 0 and (2) if c and c' are indicator constants it outputs c'. If none of the terminal cases are met, then it outputs a new product node c_1 with children as the constant children of c. Subsequently the merge operation is propagated by: (1) traversing both input AACs and updating the cursors of the database-AAC appropriately, (2) considering the descendant sum nodes of a_1 and a_2 and (3) recursively merging them. Finally, the result of the merge operation between a_1 and a_2 is cached.

Figure 6 depicts the merging operation as applied to the database-AACs in Figure 4 and the lineage-AAC in Figure 5(c) using the variable order X_1, X_2, Y_1, Y_2, Z_1, Z_2. The first step is to merge the sum nodes that are annotated with X_1 and create a new sum node with the same annotation that contains both 0.6 and $\lambda_{x_{11}}$ in its children nodes. As shown the algorithm continues in a depth-first manner and considers the sum node present in the left sub-AAC.

Case - Different variables: When the merge operation is applied to sum nodes with variables $v_2 \prec v_1$ in Π_{final} the output node is a sum node a annotated with v_2. To construct the children of node a the algorithm iterates through *each* child c of node a_2 corresponding to a particular instantiation of v_2 and performs the following process: if c is a constant it outputs c, otherwise it outputs a new

Algorithm 2 multiMerge(a_1: lineage AAC, a_{col}: AAC Collection): returns AAC

```
$a_2 \leftarrow$ getAACCursor(Var($a_1$))
**if** cache($a_1, a_2$) $\neq$ *null* **then**
    return cache($a_1, a_2$)
**else if** ($a_1 == 0$) or ($a_1 == 1$) **then**
    return $a_1$
**end if**
**if** (Pos($a_2$) $<$ Pos($a_1$)) **then**
    $a \leftarrow$ new + node; Var($a$) $\leftarrow$ Var($a_2$)
    **for** $v \in$ Values(Var($a_2$)) **do**
        $c \leftarrow$ getChild($a_2, v$)
        //c is either a $*$ node or a constant
        **if** ($c == 0$) or ($c$ is indicator constant) **then**
            $c_1 \leftarrow c$
        **else**
            $c_1 \leftarrow$ new $*$ node
            Const($c_1$) $\leftarrow$ Const($c$)
            moveCursorToChild($a_2, v$)
            $c_2 \leftarrow$ multiMerge($a_1, a_{col}$); addChild($c_1, c_2$)
            moveCursorToParent($a_2, v$)
        **end if**
        addChild($a, c_1, v$)
    **end for**
**else if** (Pos($a_2$) $==$ Pos($a_1$)) **then**
    $a \leftarrow$ new + node; Var($a$) $\leftarrow$ Var($a_1$)
    **for** $v \in$ Values(Var($a_1$)) **do**
        $c \leftarrow$ getChild($a_1, v$); $c' \leftarrow$ getChild($a_2, v$)
        //c and c' are either $*$ nodes or constants
        **if** ($c == 0$) or ($c' == 0$) **then**
            $c_1 \leftarrow 0$
        **else if** ($c$ and $c'$ are indicator constants) **then**
            $c_1 \leftarrow c'$
        **else**
            $c_1 \leftarrow$ new $*$ node
            Const($c_1$) $\leftarrow$ Const($c'$)
            moveCursorToChild($a_2, v$)
            **if** ($c$ has a + node in its children) **then**
                $c \leftarrow$ getChild($c$)
                $c_2 \leftarrow$ multiMerge($c, a_{col}$); addChild($c_1, c_2$)
            **end if**
            moveCursorToParent($a_2, v$)
        **end if**
        addChild($a, c_1$)
    **end for**
**end if**
cacheInsert($a, a_1, a_2$)
return $a$
```

product node c_1 with children the constant children of c. Subsequently, the merge operation is propagated by: (1) traversing the database-AAC and updating its cursors appropriately, (2) considering the descendant sum nodes of a_2, and (3) recursively merging them with a_1. Finally, the result of the merge operation between a_1 and a_2 is stored in the cache.

Figure 7 depicts a subsequent step of the merging process shown in Figure 6. The merging operation is performed between two different variables, namely Y_2 and Y_1. Since $Y_1 \prec Y_2$ in the variable order, the output is a copy of the sum node, annotated with Y_1, in the database-AAC. The algorithm proceeds recursively to merge the sum nodes that are annotated with variable Y_2.

In the algorithm, method getAACCursor(x) returns the appropriate running cursor of the AAC in A_{Col} in which variable x appears. When applied to a sum node, method Var() returns the variable annotation of that node. Furthermore method Values(v) returns a set of all possible values in the domain of variable v. For a sum node a, method addChild(a, c, v) adds a new child c in a for Var(a) $= v$. For product nodes, method Const() returns the children of the node corresponding to constant nodes. For a sum node

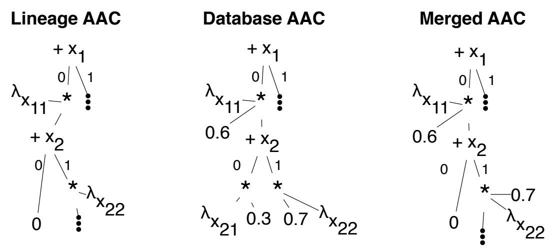


Figure 6: Partial AAC produced after merging the lineage-AAC with the corresponding database-AAC.

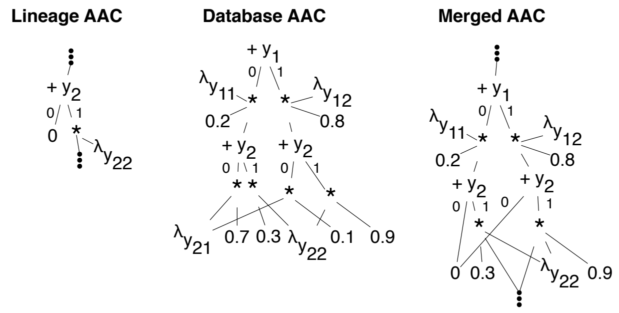


Figure 7: A subsequent step of the merging process shown in Figure 6. Note that the sum nodes to be merged in this step refer to different variables, namely Y_2 and Y_1.

with a variable annotation V, methods getChild(v) and Pos() return the child of the node corresponding to $V = v$ and the position of the variable labeling of the node in the global variable order Π_{final} respectively. Method moveCursorToChild(a, v) sets the AAC cursor to point to the descendant sum node of node a in the path for which Var(a) $= v$. Finally, moveCursorToParent(a, v) returns the AAC cursor to the preceding sum node of a.

As mentioned in the previous section, minimizing memory usage and the number of operations performed during merging is important for the performance of the algorithm. A variable in the lineage-AAC may appear in different paths, therefore, after the merge, parts of the database-AACs will be repeated. In order to leverage the detection and caching of those sub-circuits we require that variables which appear together in an AAC from A_{Col} also appear together in the final AAC (Section 4.3, Requirement (d)).

Finally, we analyze the complexity of the merging algorithm after caching is introduced. Let m be the size of the query AAC and s_i be the size of the i^{th} AAC from the set $A \subseteq A_{\text{Col}}$ of AACs used during the merge phase. In the worst case the algorithm parses each entire annotated arithmetic circuit at most once. Therefore, its complexity is $O(m * \sum_{i \in A} s_i)$. To compute the result probability, we set all indicator variables in the final AAC to 1 and parse the circuit. This operation takes time linear in the size of the final circuit.

6. EXPERIMENTS

In this section we present an experimental evaluation of our framework. The evaluation was performed on an Intel(R) Core(TM) i5 2.3 GHz/64bit/8GB machine running Mac OS X/g++ 4.6.1. Our framework is implemented in C++ for query extraction, lineage processing and probability computation. We used PostgreSQL 9.0 for storing the probabilistic database and the factors. For BDD construction we use the publicly available CUDD package [39] released by the VLSI group at the University of Colorado. We compare our approach to variable elimination, a generic approach which can support both tuple-independent and correlated tuples [36]. We examine two versions of VE. The first is regular VE using a tabular representation of the factors, and the second is VE where factors are represented using ADDs. VE with ADDs can capture the local structure in the factors of the network. For our results we report wall-clock times of queries averaged over 5 runs.

6.1 Datasets and Queries

We study both tuple-independent and correlated cases. The data used for the experiments was generated by a modified version of the TPC-H data generator. In the first case the generator was modified to create tuple-independent probabilistic databases. We assume that all tables apart from *nation* and *region* are probabilistic and associate each tuple with some existence probability uniformly sampled between $(0, 1]$.

In the second case, we focus on probabilistic databases with arbitrary correlations. We extend the TPC-H data generator to generate correlated probabilistic data according to the following model. We assume that all the tables apart from *nation* and *region* contain uncertain data. Furthermore, tables *customer*, *supplier*, and *partsupp* contain independent tuples. Following the foreign key constraint, each tuple in the *lineitem* table depends on the corresponding tuple from the *orders* table. We note that many entries of the *orders* table are associated with multiple entries from the *lineitem* table. This introduces many conditionally independent random variables associated with tuples from the *lineitem* table.

For the *part* table, we assume that there is uncertainty over the part price, and that there is a mutual exclusion constraint over price. Finally, for the table *orders* we assume that the orders of a particular customer for a given month are correlated. In particular, this type of correlation can be represented as a chain where the orders are sorted chronologically and then the existence of an order depends on the preceding order. This scenario is realistic as the orders within a particular month may be connected with the same project and they may depend on each other for the fulfillment of that project. The length of the chain varies for databases of different sizes. For a scale factor of 0.001 the maximum length is restricted to 2 while for a scale factor of 0.1 it increases to 10.

We evaluate our framework for both tractable and hard queries for five different scale factors, namely 0.001, 0.005, 0.01, 0.05 and 0.1. We consider queries Q2, Q3, Q5, Q6, Q8, and Q16. Queries Q6 and Q16 do not contain complicated joins and are easy. However, they are challenging because they generate an increased number of result tuples. For every query we remove the top-level aggregate functions and consider its Boolean version.

6.2 Experimental Results

We begin by evaluating the scalability of the database compilation technique based on arithmetic circuits. Figure 8 illustrates the compilation time as a function of the size of the underlying database and, in particular, the scale factor used to generate it. As illustrated the time required for compiling the database introduces an sizeable overhead to our framework. However, since compilation is performed offline this overhead does not affect the performance of the system during query evaluation.

We examine the efficiency of our framework during query evaluation and, in particular, the total execution time for the Boolean versions of the TPC-H queries described above. The results are shown in Figure 9. Each graph presents the total evaluation time for a single query for both independent and correlated databases of different sizes. Note that in all graphs the y-axis is in logarithmic scale. Finally, missing values for a particular scale factor correspond to cases where the total evaluation time exceeds the time threshold of 100 seconds.

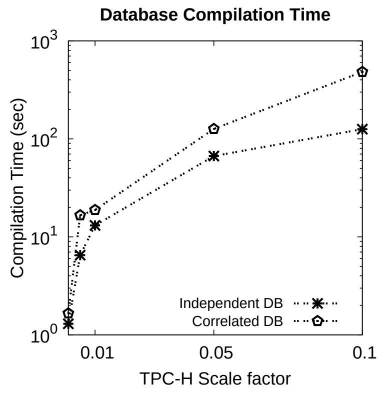


Figure 8: The time required to compile the database as a function of its size.

We focus on hard queries. As shown, for all hard queries, the evaluation based on AACs is at least one order of magnitude faster compared to regular VE but it is also significantly faster than VE with ADDs . As expected, VE with ADDs is faster than tabular VE, since ADDs can capture the local structure in the factors of the network. However, even when using VE with ADDs we still have to pay the cost of multiplying the different ADDs and summing-out variables during online query evaluation. To the contrary, our approach has a significant advantage compared to both baselines as this cost is paid only once, during the offline preprocessing phase. Of particular interest are queries Q3 and Q8 where both versions of VE exceeds the threshold of 100 seconds for both dependency assumptions for scale factors larger than 0.01. This is because of the increasing number of distinct variables in the lineage formulas for larger scale factors. For example the lineage formula for Q3 contains 6276 distinct random variables for a scale factor of 0.1.

Since we are considering the Boolean versions of the queries, the size of the lineage formula is directly associated with the treewidth of the final augmented factor graph. To the contrary, AACs are more scalable since they can fully exploit determinism across the entire network rather than only at a factor level. For example, for Q3 and a scale factor 0.1, the resulting AACs have 42486 and 53803 edges for the independent and the correlated case respectively. Observe that the difference in the size is not that significant despite the presence of correlations as determinism is present in the network. In general, the sizes of the final AACs were sensitive to the queries and the size of the database. For the correlated database experiments the size of the AACs ranged from 19 to 2570 edges for a scale factor of 0.001 and from 1359 to 353214 edges for a scale factor of 0.1.

We continue our discussion and focus on queries 6 and 16. Recall that query 6 contains a projection over table *lineitem* and no joins, while query 16 contains a join between tables *partsupp* and *part*. For both, we observe that the performance gain of using AACs is decreasing as the size of the database increases.

To understand this behavior better, we ran micro-benchmarking experiments to investigate the performance of the different components in our framework. We evaluated all queries against correlated databases and we measured the time spent at the different steps of the final AAC creation process. We measure the time for: (a) generating the lineage for the result tuples, (b) generating the final variable ordering using the new algorithm presented in Section 4.3, (c) creating the lineage OBDD and converting it to an AAC using the previous variable ordering, and (d) merging all the AACs together.

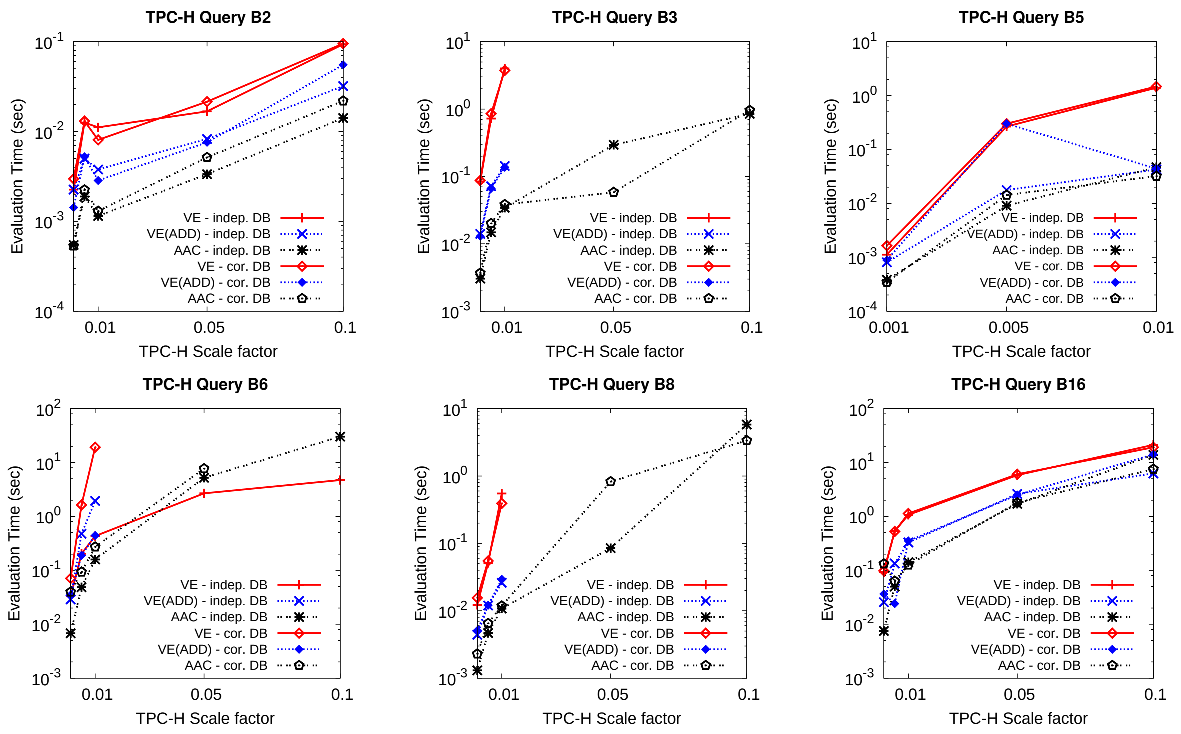


Figure 9: Query evaluation times for the Boolean versions of TPC-H queries for both independent and correlated databases of different sizes. Figures (a), (b), (e) refer to hard queries, while the rest to easy queries. Missing values for a scale factor correspond to queries that exceeded the time threshold of 100 seconds.

We omit the actual evaluation of the final AAC since it is linear in the size of the final structure , therefore, extremely efficient. Due to space constraints we present the results for two representative queries in Figure 10. Similar patterns were observed for the rest of the queries.

As shown in the figure, most of the time is spent in creating the lineage OBDD. We demonstrate that this time increases significantly as the size of the database (and consequently the size of the lineage formula) increases. In particular, the size of the linage formula ranges from 59 to 6276 distinct random variables for Q3 and from 142 to 15010 distinct random variables for Q16. Moreover we observed that for all cases where query evaluation with AACs exceeded the threshold of 100 seconds, the actual bottleneck was creating the OBDD for the lineage formula. We would like to point out that an external package was used for creating OBDDs. Improving the performance and optimizing this process is left as future work. Nevertheless, we see that only a small portion of the total running time is spent in the new merging algorithm proposed. Finally, this analysis also explains why for Q6 and Q16 we see a decreasing performance gain when using AACs.

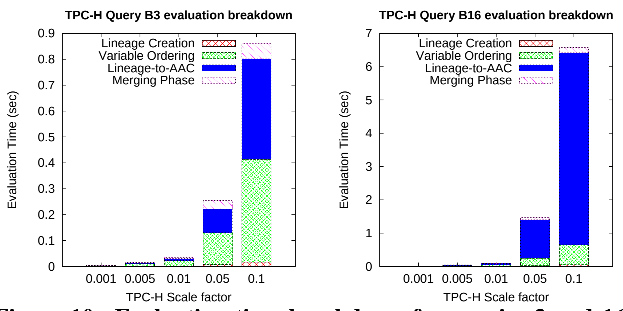


Figure 10: Evaluation time breakdown for queries 3 and 16, against correlated databases of multiple sizes.

7. RELATED WORK

Much of the work in probabilistic database literature has focused on query evaluation under tuple-independence assumption, with some of that work also allowing deterministic correlations like mutual exclusion. Several recent works have attempted to support more complex correlations, typically represented using graphical models; these include BayesStore [41], PrDB [37], and the work by Wick et al. [42] which uses an MCMC-based technique. However, none of that work exploits local structure for efficient query evaluation. In a followup work to the OBDD-based approach that is limited to tuple-independent databases, Olteanu et al. [32] proposed *decomposition trees* (d-trees), that can support simple correlations expressed via Boolean formulas, but they cannot handle arbitrary correlations in a natural way. While obeying similar structural properties as AACs, d-trees can decompose the lineage formula only partially and can exploit sub-formulas that can be evaluated efficiently. Moreover d-trees can be used to compute approximate confidence values. It would be an interesting future research direction to combine our approach with d-trees. In a recent work, Jha et al. [21] proposed a framework to combine the intensional and extensional approaches, where they try to use an extensional method as much as possible, falling back to using an intensional approach only when necessary. However, their approach cannot be applied directly to correlated databases represented using factor graphs. Aside from factor graphs, other representations like *pc-tables* [19] can be used to represent correlations. We note that our framework is still applicable in that case, however the preprocessing compilation algorithm (Section 4.2) should be replaced with a logical knowledge base compilation algorithm [5] for compiling the database-AACs. Finally, Sanner et al. [35] propose an extension of ADDs, called Affine ADDs, that is capable of compactly representing context-specific, additive, and multiplicative structure.

While sharing similarities with AACs, affine ADDs cannot represent conditional independences present in the correlations.

8. CONCLUSIONS AND FUTURE WORK

Probabilistic databases are becoming an increasingly appealing option to store and query uncertain data generated in many application domains. In this paper, we focused on efficiently supporting query evaluation over probabilistic databases that contain *correlated* data, naturally generated in many applications of probabilistic databases. We introduced a novel framework that exploits the prevalent determinism in the correlations, and other types of local structure such as context-specific independence. Our framework builds upon arithmetic circuits, an enhanced exact inference technique that exploits such local structure for compact representation and efficient inference. We introduced an extension of arithmetic circuits, called annotated arithmetic circuits, and showed how to execute probabilistic queries over them. Our experimental evaluation shows that our approach can result in orders-of-magnitude speedups in query execution times over prior approaches. Our techniques are also of independent interest to the machine learning community where arithmetic circuits have re-emerged as an appealing alternative to model uncertainties in several domains with large volumes of uncertain data [28, 33]. Our research so far has raised several interesting challenges, including developing *lifted inference*-based approaches that can exploit not only local structure but also *symmetry* and other regularities in the uncertainties, and developing approximate query evaluation techniques.

Acknowledgments: We would like to thank Angelika Kimmig and Vassilis Zikas for their valuable comments. This work was supported by Air Force Research Lab (AFRL) under contract FA8750-10-C-0191, and by NSF under grant IIS-0916736.

9. REFERENCES

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD*, 1993.

[2] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.*, 45, 1996.

[3] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *UAI*, 1996.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 1992.

[5] M. Chavira and A. Darwiche. Compiling Bayesian networks with local structure. In *IJCAI*, 2005.

[6] M. Chavira and A. Darwiche. Compiling Bayesian networks using variable elimination. In *IJCAI*, 2007.

[7] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7), 2008.

[8] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational Bayesian networks for exact inference. *Int. J. Approx. Reasoning*, 42(1–2), 2006.

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

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

[11] A. Darwiche. A logical approach for factoring belief networks. In *Proceedings of KR*, 2001.

[12] A. Darwiche. A differential approach to inference in Bayesian networks. *JACM*, 50, 2003.

[13] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[14] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *UAI*, 1996.

[15] A. Deshpande, L. Getoor, and P. Sen. *Graphical Models for Uncertain Data*. Managing and Mining Uncertain Data. Charu Aggarwal ed., Springer, 2008.

[16] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

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

[18] R. Ebendt, G. Fey, and R. Drechsler. *Advanced BDD Optimization*. Springer, 2005.

[19] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *EDBT Workshops*, 2006.

[20] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. In *IEEE Data Engineering Bulletin*, 2006.

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

[22] A. Jha and D. Suciu. Knowledge compilation meets database theory: compiling queries to decision diagrams. In *ICDT*, 2011.

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

[24] B. Kanagal and A. Deshpande. Lineage processing over correlated probabilistic databases. In *SIGMOD*, 2010.

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

[26] A. Kumar and C. Re. Probabilistic management of OCR data using an RDBMS. In *PVLDB*, 2012.

[27] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *TODS*, 1997.

[28] D. Lowd and P. Domingos. Learning arithmetic circuits. In *UAI*, 2008.

[29] T. Mantadelis, R. Rocha, A. Kimmig, and G. Janssens. Preprocessing boolean formulae for bdds in a probabilistic context. In *JELIA*, 2010.

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

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

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

[33] H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In *UAI*, 2011.

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

[35] S. Sanner and D. McAllester. Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference. In *IJCAI*, 2005.

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

[37] P. Sen, A. Deshpande, and L. Getoor. PrDB: managing and exploiting rich correlations in probabilistic databases. *The VLDB Journal*, 18, 2009.

[38] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 2010.

[39] F. Somenzi. Cudd: Cu decision diagram package. `http://vlsi.colorado.edu/fabio/CUDD/`.

[40] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool, 2011.

[41] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. BayesStore: Managing large, uncertain data repositories with probabilistic graphical models. In *VLDB*, 2008.

[42] M. L. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and MCMC. *PVLDB*, 3(1), 2010.

[43] N. L. Zhang and D. Poole. On the role of context-specific independence in probabilistic inference. In *IJCAI*, 1999.