Approximate Confidence Computation in Probabilistic Databases

http://www.comlab.ox.ac.uk/projects/SPROUT/

Dan Olteanu (Oxford), Jiewen Huang (Oxford), Christoph Koch (Cornell)
Uncertain and Probabilistic Data

Uncertain and probabilistic data is commonplace:
- Information extraction
- Processing manually entered data (such as census forms)
- Data cleaning, data integration
- Risk management: Decision support queries, hypothetical queries
- Social network analysis
- ...

Recent years have seen advances in developing
- representation models for uncertain/probabilistic data,
- uncertainty-aware query languages, and
- query evaluation techniques for such data.
  ▶ scope of this work.
Contributions of this Work

- Efficient deterministic technique for confidence computation.
  - approximate confidences with error guarantees
    - positive relational algebra queries
    - U-relational databases
  - exact confidences for known tractable queries in polynomial time
    - hierarchical conjunctive queries without self-joins, max-one inequality queries
    - tuple-independent probabilistic databases

- Implementation of this technique in the SPROUT query engine.
  - extends PostgreSQL backend with confidence computation operators
  - used by MayBMS (maybms.sourceforge.net)

- Experimental comparison with existing techniques.
  - fastest technique so far for tractable queries (previous SPROUT)
  - Monte Carlo algorithm (MayBMS)
U-relational Probabilistic Databases

Syntax.
Probabilistic databases are relational databases where
- There is a finite set of independent random variables $X = \{x_1, \ldots, x_n\}$ with finite domains $\text{Dom}_{x_1}, \ldots, \text{Dom}_{x_n}$.
- Each tuple is associated with a conjunction of atomic events of the form $x_i = a$ or $x_i \neq a$ where $x_i \in X$ and $a \in \text{Dom}_{x_i}$.
- There is a probability distribution over the assignments of each variable.

Semantics.
- *Possible worlds* defined by total assignments $\theta$ over $X$.
- The world defined by assignment $\theta$
  - consists of all tuples with condition $\phi$ such that $\theta(\phi) = \text{true}$.
  - has probability defined by the product of probabilities of each assignment in $\theta$.

This formalism can represent any discrete probability distribution over relational databases.
Example: Probabilistic Databases

Consider a simplified TPC-H scenario with customers (Cust) and orders (Ord):

<table>
<thead>
<tr>
<th>Cust</th>
<th>name</th>
<th>V_1</th>
<th>P_1</th>
<th>V_2</th>
<th>P_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>x_1</td>
<td>0.1</td>
<td>x_3</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>Dan</td>
<td>x̄_1</td>
<td>0.9</td>
<td>x_4</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>Li</td>
<td>x_2</td>
<td>0.3</td>
<td>x̄_4</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>Mo</td>
<td>x̄_2</td>
<td>0.7</td>
<td>x̄_5</td>
<td>0.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ord</th>
<th>okey</th>
<th>ckey</th>
<th>date</th>
<th>V_3</th>
<th>P_3</th>
<th>V_4</th>
<th>P_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1995-01-10</td>
<td>y_1</td>
<td>0.1</td>
<td>x̄_5</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1996-01-09</td>
<td>y_2</td>
<td>0.2</td>
<td>x̄_4</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1994-11-11</td>
<td>y_3</td>
<td>0.3</td>
<td>x_3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

- Variables are Boolean (wlog); write x instead of x = 1, x̄ instead of x = 0.
- A pair (V_i, P_i) states that the variable assignment given by V_i has the probability given by P_i.
- Conditions can represent arbitrary correlations between tuples, eg,
  - (1,Joe) and (3,Li) are independent: They use disjoint sets of variables.
  - (1,Joe) and (2,Dan) are mutually exclusive: x_1 is either true or false.
Query Evaluation in Probabilistic Databases

Semantically, the query is evaluated in each world.

- Too expensive for any practical purpose!

Common approach:

1. Evaluate the query directly on the representation.
   - Done with relational query plans for our probabilistic data formalism.
   - In addition to standard evaluation, copy the input conditions to the output.

2. Compute the confidence of each answer tuple.
   - Reducible to probability computation of Boolean formulas over random variables
   - Known to be \#P-hard already for positive bipartite DNF formulas!
Example: Query Evaluation

Query asking for the probability that customer 'Joe' has placed orders:

\[
Q = \pi_\emptyset(\sigma_{\text{name}='}Joe'(\text{Cust}) \land_{\text{key}} \text{Ord})
\]

<table>
<thead>
<tr>
<th></th>
<th>V_1</th>
<th>P_1</th>
<th>V_2</th>
<th>P_2</th>
<th>V_3</th>
<th>P_3</th>
<th>V_4</th>
<th>P_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1</td>
<td>0.1</td>
<td></td>
<td>x_3</td>
<td>0.1</td>
<td>y_1</td>
<td>0.1</td>
<td></td>
<td>(\overline{x_5})</td>
</tr>
<tr>
<td>x_1</td>
<td>0.1</td>
<td></td>
<td>x_3</td>
<td>0.1</td>
<td>y_2</td>
<td>0.2</td>
<td></td>
<td>(\overline{x_4})</td>
</tr>
</tbody>
</table>

- Probability of the answer tuple is the probability of the associated DNF \(x_1 x_3 y_1 \overline{x_5} + x_1 x_3 y_2 \overline{x_4}\).

Difficulty:
- The sets of satisfying assignments of any two clauses in the DNF may overlap.
- It may require to iterate over its (exponentially many) satisfying assignments.

Approximate computation, if done quickly enough, may suffice in most applications.
Approximate Confidence Computation in SPROUT

Basic algorithm:

- decompose the DNF into an equivalent form that allows for efficient probability computation.

- after each decomposition step, compute lower and upper bounds on the probabilities of the DNFs obtained by decomposition and of the initial DNF.

- stop when the desired approximation is obtained or on timeout.

- otherwise, continue with a new decomposition step.

In practice, good approximations can be obtained after a few *well-chosen* decomposition steps.
Types of Decompositions

Given DNF formula $\Phi$. Apply the following steps in the given order.

1. **Independent-or $\otimes$:** Partition $\Phi$ into independent DNFs $\Phi_1, \Phi_2 \subset \Phi$ such that $\Phi \equiv \Phi_1 \lor \Phi_2$.

2. **Independent-and $\odot$:** Partition $\Phi$ into independent DNFs $\Phi_1$ and $\Phi_2$ such that $\Phi \equiv \Phi_1 \land \Phi_2$.

3. **Exclusive-or $\oplus$:** Choose a variable $x$ in $\Phi$. Then,

$$\Phi \equiv \bigoplus_{a \in \text{Dom}_x, \Phi|_{x=a} \neq \emptyset} \left( (x = a) \odot \Phi |_{x=a} \right).$$

For DNFs of query answers, the decompositions

- preserve equivalence,
- are efficiently computable, and
- allow for efficient probability computation.
D-trees: Decomposition Trees

A d-tree is a formula constructed from $\otimes$, $\oplus$, $\odot$ and nonempty DNFs (as “leaves”). If each leaf holds one clause, the d-tree is complete.

Example: Complete d-tree for

\begin{align*}
x &= 1 \lor \\
x &= 2 \land y = 1 \lor \\
x &= 2 \land z = 1 \lor \\
u &= 1 \land v = 1 \lor \\
u &= 2.
\end{align*}
Lower and Upper Bounds for D-trees

Bounds $[L, U]$ on the probability of a d-tree can be computed efficiently if each leaf of a d-tree has lower $L_i$ and upper $U_i$ bounds of its probability.

Example: D-tree for $\Phi = \Phi_1 \otimes \{[(x = 1) \odot \Phi_2] \oplus \Phi_3\}$:

Then,

\[
L(\Phi) = L_1 \otimes [Pr(x = 1) \odot L_2 \oplus L_3] \\
U(\Phi) = U_1 \otimes [Pr(x = 1) \odot U_2 \oplus U_3]
\]
How to Efficiently Compute Probability Bounds for Leaves?

Many possible approaches. We used the following simple approach:

- Given a leaf (that is, a DNF) $\Psi$.
- Choose a maximal subset $S$ of pairwise independent clauses in $\Psi$.
- Then, $P(S)$ is a lower bound for $P(\Psi)$, and

$$\min(1, P(S) + \sum_{c \in (\Psi - S)} (P(c)))$$

is an upper bound for $P(\Psi)$.

Rationale: We want a quick solution for computing the bounds, since this operation needs to be done for each node of the d-tree.

- We compute in one scan over $\Psi$ the lower and upper bounds for $P(\Psi)$. 

Absolute and Relative Approximation Errors

- \( \hat{p} \) is an **absolute** \( \epsilon \)-approximation of \( p \) if \( p - \epsilon \leq \hat{p} \leq p + \epsilon \).

- \( \hat{p} \) is a **relative** \( \epsilon \)-approximation of \( p \) if \( (1 - \epsilon) \cdot p \leq \hat{p} \leq (1 + \epsilon) \cdot p \).

Given a DNF \( \Phi \), a fixed error \( \epsilon \), and a d-tree for \( \Phi \) with bounds \([L, U] \).

- If \( U - \epsilon \leq L + \epsilon \), then any value in \([U - \epsilon, L + \epsilon]\) is an absolute \( \epsilon \)-approximation of \( P(\Phi) \).

- If \( (1 - \epsilon) \cdot U \leq (1 + \epsilon) \cdot L \), then any value in \([(1 - \epsilon) \cdot U, (1 + \epsilon) \cdot L]\) is a relative \( \epsilon \)-approximation of \( P(\Phi) \).
Memory-Efficient Version of the Algorithm

The previous algorithm keeps the entire d-tree in main memory.

Improvement idea:

- Construct the d-tree in depth-first traversal.
- When at a leaf, decide locally whether we should further decompose it or close it, that is, move up the tree to the following open leaf.

When can we close a leaf?

- compute the bounds of the d-tree with largest difference for any possible probability each open leaf may take.
- these bounds must satisfy the condition for an $\epsilon$-approximation.
- efficient way: bounds computed by choosing for each open leaf the bounds $[L_i, L_i]$, where $L_i$ is a lower bound for that leaf.
Example: Memory-efficient Algorithm

Assume $\Phi_1$ is closed, $\Phi_2$ is current, $\Phi_3$ is open. Let absolute error $\epsilon = 0.012$.

Test at $\Phi_2$ whether

- we can stop with an absolute $\epsilon$-approximation.
  - NO! Check by considering all leaves closed and compute the bounds.
    - $U - L = 0.644 - 0.595 = 0.049 \leq 2 \cdot 0.012 = 0.024$ does not hold.

- we can close $\Phi_2$.
  - YES! Check by considering all preceding leaves closed and all following leaves open, then compute the bounds.
    - $U' - L = 0.6173 - 0.595 = 0.0223 \leq 0.024$ holds.
Tractable Queries on Tuple-Independent Databases

Our d-trees naturally capture DNFs for tractable queries:

- DNFs for any tractable conjunctive query without self-joins can be compiled in polynomial time into complete d-trees with nodes $\otimes$ and $\odot$.
  - In this case, the d-trees correspond to read-once functions.

- DNFs for existing (max-one) tractable inequality queries can be compiled in polynomial time into complete d-trees with nodes $\oplus$.

In both cases, the d-trees have sizes linear in the number of literals in the DNF.
Experiments

Scale factor 1, probabilities of input tuples in (0,0.01)

Tractable TPC-H queries (aggregations/ineq-joins dropped) on tuple-independent tables

Dolphin social network

Time in sec (ln scale)
Relative error (ln scale)
Thanks!