Efficient Query Evaluation on Probabilistic Databases

Nilesh Dalvi Dan Suciu

{nilesh,suciu}@cs.washington.edu
University of Washington, Seattle, WA, USA

Abstract

We describe a system that supports arbitrarily complex SQL queries on probabilistic databases. The query semantics is based on a probabilistic model and the results are ranked, much like in Information Retrieval. Our main focus is efficient query evaluation, a problem that has not received attention in the past. We describe an optimization algorithm that can compute efficiently most queries. We show, however, that the data complexity of some queries is $\#P$-complete, which implies that these queries do not admit any efficient evaluation methods. For these queries we describe both an approximation algorithm and a Monte-Carlo simulation algorithm.

1 Introduction

Databases and Information Retrieval [5] have taken two philosophically different approaches to queries. In databases SQL queries have a rich structure and a precise semantics. This makes it possible for users to formulate complex queries and for systems to apply complex optimizations, but users need to have a pretty detailed knowledge of the database in order to formulate queries. For example, a single misspelling of a constant in the WHERE clause leads to an empty set of answers, frustrating casual users. By contrast, a query in Information Retrieval (IR) is just a set of keywords and is easy for casual users to formulate. IR queries offer two important features that are missing in databases: the results are *ranked* and the matches may be *uncertain*, i.e. the answer may include documents that do not match all the keywords in the query[1]. While several proposals exist for extending SQL with uncertain matches and ranked results [3, 19, 16], they are either restricted to a single table, or, when they handle join queries, adopt an ad-hoc semantics.

To illustrate the point consider the following structurally rich query, asking for an actor whose name is like 'Kevin' and whose first 'successful' movie appeared in 1995:

```
SELECT  *
FROM    Actor A
WHERE   A.name ≈ 'Kevin'
  and   1995 =
        SELECT  MIN(F.year)
        FROM    Film F, Casts C
        WHERE   C.filmid = F.filmid
                and C.actorid = A.actorid
                and F.rating ≈ "high"
```

The two $\approx$ operators indicate which predicates we intend as uncertain matches. Techniques like edit distances, ontology-based distances [15], IDF-similarity and QF-similarity [3] can be applied to a *single* table: to rank all `Actor` tuples (according to how well they match the first uncertain predicate), and to rank all `Film` tuples. But it is unclear how to rank the *entire* query. To date, no system combines structurally rich SQL queries with uncertain predicates and ranked results.

In this paper we propose such a system. We introduce a new semantics for database queries that supports uncertain matches and ranked results, by combining the probabilistic relational algebra [13] and models for belief [4]. Given a SQL query with uncertain predicates, we start by assigning a probability to each tuple in the input database according to how well it matches the uncertain predicates. Then we derive a probability for each tuple in the answer, and this determines the output ranking.

An important characteristic of our approach is that *any* query under set-semantics has a meaning, in-

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[1] Some IR systems only return documents that contain all keywords, but this is a feature specific to those systems, and not of the underlying vector model used in IR.

cluding queries with joins, nested sub-queries, aggregates, group-by, and existential/universal quantifiers[2]. Queries have now a probabilistic semantics, which is simple and easy to understand by both users and implementors.

The main problem is query evaluation, and this is the focus of our paper. Our approach is to represent SQL queries in an algebra, and modify the operators to compute the probabilities of each output tuple. This is called *extensional semantics* in [13], and is quite efficient. While this sounds simple, the problem is that it doesn't work: extensional evaluation ignores the complex correlations present in the probabilities of the intermediate results and the probabilities computed this way are wrong in most cases, and lead to incorrect ranking. In [13], the workaround is to use an *intensional semantics* [3], which is much more complex and, as we show here, impractical. Our approach is different: we rewrite the query plans, searching for one where the extensional evaluation is correct. We show however that certain queries have a #P-complete data complexity under probabilistic semantics, and hence do not admit a correct extensional plan. However, many queries that occur in practice do have a correct extensional plan (8 out of the 10 TPC/H queries fall in this category). For others, we describe two techniques for evaluation: a heuristics to choose a plan that avoids large errors, and a Monte-Carlo simulation algorithm, which is more expensive but can guarantee arbitrarily small errors.

Outline We give motivating examples in Sec. 2, define the problem in Sec. 3, and describe our techniques in Sec. 4-8. Sec. 9 reports experiments and Sec. 10 describes related work. We conclude in Sec. 11.

2 Examples

We illustrate the main concepts and techniques of this paper with two simple examples.

Probabilistic Database In a probabilistic database each tuple has a certain probability of belonging to the database. Figure 1 shows a probabilistic database D^p with two tables, S^p and T^p: the tuples in S^p have probabilities 0.8 and 0.5, and the unique tuple in T^p has probability 0.6. We use the superscript p to emphasize that a table or a database is probabilistic. We assume in this example that the tuples are independent probabilistic events, in which case the database is called *extensional* [13].

The meaning of a probabilistic database is a probability distribution on all database instances, which we call *possible worlds*, and denote $pwd(D^p)$. Fig. 2 (a) shows the eight possible instances with non-zero probabilities, which are computed by simply multiplying

$S^p =$

	A	B	
s_1	'm'	1	0.8
s_2	'n'	1	0.5

$T^p =$

	C	D	
t_1	1	'p'	0.6

Figure 1: A probabilistic database D^p

$pwd(D^p) =$

database instance	probability
$D_1 = \{s_1, s_2, t_1\}$	0.24
$D_2 = \{s_1, t_1\}$	0.24
$D_3 = \{s_2, t_1\}$	0.06
$D_4 = \{t_1\}$	0.06
$D_5 = \{s_1, s_2\}$	0.16
$D_6 = \{s_1\}$	0.16
$D_7 = \{s_2\}$	0.04
$D_8 = \phi$	0.04

(a)

$$q(u) : -S^p(x, y), T^p(z, u), y = z$$

(b)

$q^{pwd}(D^p) =$

answer	probability
$\{'p'\}$	0.54
$\emptyset$	0.46

(c)

Figure 2: (a) The possible worlds for D^p in Figure 1, (b) a query q, and (c) its possible answers.

the tuple probabilities, as we have assumed them to be independent. For example, the probability of D_2 is $0.8 * (1 - 0.5) * 0.6 = 0.24$, since the instance contains the tuples s_1 and t_1 and does not contain s_2.

We now illustrate query evaluation on probabilistic databases. Consider the conjunctive query q in Fig. 2 (b). Its meaning on D^p is a set of possible answers, shown in Fig. 2 (c). It is obtained by applying q to each deterministic database in $pwd(D^p)$, and adding the probabilities of all instances that return the same answer. In our example we have $q(D_1) = q(D_2) = q(D_3) = \{'p'\}$, and $q(D_4) = \ldots = q(D_8) = \emptyset$. Thus, the probability of the answer being $\{'p'\}$ is $0.24+0.24+0.06 = 0.54$, while that of the answer $\emptyset$ is 0.46. This defines the set of possible answers, denoted $q^{pwd}(D^p)$. Notice that we have never used the structure of the query explicitly, but only applied it to deterministic databases taken from $pwd(D^p)$. Thus, one can give a similar semantics to *any* query q, no matter how complex, because we only need to know its meaning on deterministic databases.

The set of possible answers $q^{pwd}(D^p)$ may be very large, and it is impractical to return it to the user. Instead, we compute for each possible tuple t a *probability rank* that t belongs to any answer, and return tuples sorted by this rank. We denote this $q^{rank}(D^p)$. In our example this is:

$q^{rank}(D^p) =$

D	Rank
'p'	0.54

[2]In this paper we restrict our discussion to SQL queries whose normal semantics is a set, not a bag or an ordered list.

[3]We define extensional and intensional semantics formally in Sec. 4.

A	B	C	D	*prob*
'm'	1	1	'p'	0.8*0.6 = 0.48
'n'	1	1	'p'	0.5*0.6 = 0.30

(a) $S^p \bowtie_{\mathbf{B=C}} T^p$

D	*prob*
'p'	(1 - (1 - 0.48)(1 - 0.3)) = 0.636

(b) $\Pi_{\mathbf{D}}(S^p \bowtie_{\mathbf{B=C}} T^p)$

Figure 3: Evaluation of $\Pi_{\mathbf{D}}(S^p \bowtie_{\mathbf{B=C}} T^p)$

In this simple example $q^{rank}(D^p)$ contains a single tuple and the distinction between q^{pwd} and q^{rank} is blurred. To see this distinction clearer, consider another query, $q_1(x) : -S^p(x, y), T^p(z, y), y = z$. Here q_1^{pwd} and q_1^{rank} are given by:

$q_1^{pwd}(D^p) =$

answer	probability
$\{'m', 'n'\}$	0.24
$\{'m'\}$	0.24
$\{'n'\}$	0.06
$\emptyset$	0.46

$q_1^{rank}(D^p) =$

D	Rank
'm'	0.48
'n'	0.30

For example, the rank probability of $'m'$ is obtained as $Pr(\{'m', 'n'\}) + Pr(\{'m'\})$. In general, $q^{pwd}(D^p)$ may be exponentially large, while $q^{rank}(D^p)$ is simply a set of tuples, which are sorted by **Rank**. The problem in this paper is now to compute $q^{rank}(D^p)$ efficiently.

Extensional Query Semantics A natural attempt to compute $q^{rank}(D^p)$ is to represent q as a query plan then compute the probabilities of all tuples in all intermediate results. For the query q in Fig. 2 (b), such a plan is $p = \Pi_{\mathbf{D}}(S^p \bowtie_{\mathbf{B=C}} T^p)$, and the corresponding probabilities are shown in Fig. 3. The formulas for the probabilities assume tuple independence, are taken from [13] and are rather straightforward (we review them in Sec. 4). For example the probability of a joined tuple $s \bowtie t$ is the product of the probabilities of s and t. Clearly, this approach is much more efficient than computing the possible worlds $q^{pwd}(D^p)$ and then computing $q^{rank}(D^p)$, but it is wrong ! It's answer is 0.636, while it should be 0.54. The reason is that the two tuples in $S^p \bowtie_{\mathbf{B=C}} T^p$ are not independent events, hence the formula used in $\Pi_{\mathbf{D}}$ is wrong.

However, let us consider an alternative plan, $p' = \Pi_{\mathbf{D}}((\Pi_{\mathbf{B}}(S^p)) \bowtie_{\mathbf{B=D}} T^p)$. The extensional evaluation of this expression is shown in Figure 4, and this time we do get the correct answer. As we will show later, this plan will always compute the correct answer to q, on any probabilistic tables S^p, T^p. In this paper we show how to find automatically a plan whose extensional evaluation returns the correct answer to a query q. Finding such a plan requires pushing projections early (as shown in this example), join reordering, and other kinds of rewritings.

B	*prob*
1	(1 - (1 - 0.8)(1 - 0.5)) = 0.9

(a) $\Pi_{\mathbf{B}}(S^p)$

B	C	D	*prob*
1	1	'p'	0.9 * 0.6 = 0.54

(b) $\Pi_{\mathbf{B}}(S^p) \bowtie_{\mathbf{B=C}} T^p$

D	*prob*
'p'	0.54

(c) $\Pi_{\mathbf{D}}(\Pi_{\mathbf{B}}(S^p) \bowtie_{\mathbf{B=C}} T^p)$

Figure 4: Evaluation of $\Pi_{\mathbf{D}}(\Pi_{\mathbf{B}}(S^p) \bowtie_{\mathbf{B=C}} T^p)$

Queries with uncertain matches While query evaluation on probabilistic databases is an important problem in itself, our motivation comes from answering SQL queries with uncertain matches, and ranking their results. We illustrate here with a simple example on the Stanford movie database[1].

```
SELECT   DISTINCT F.title, F.year
FROM     Director D, Films F
WHERE    D.did = F.did
  and    D.name ≈ 'Copolla'
  and    F.title ≈ 'rain man'
  and    F.year ≈ 1995
```

The predicates on the director name and the movie title and year are here *uncertain*.

Our approach is to translate the query into a regular query over a probabilistic databases. Each tuple in the table `Films` is assigned a probability based on how well it matches the predicates `title ≈ 'rain man'` and `year ≈ 1995`. Several techniques for doing this exist already, and in this paper we will adopt existing ones: see Sec. 8. In all cases, the result is a probabilistic table, denoted $\texttt{Films}^p$. Similarly, the uncertain predicate on `Director` generates a probabilistic table $\texttt{Director}^p$. Then, we evaluate the following query:

```
SELECT   DISTINCT F.title, F.year
FROM     Director^p D, Films^p F
WHERE    D.did = F.did
```

This is similar to the query q considered earlier (Figure 2 (b)), and the same extensional plan can be used to evaluate it. Our system returns:

title	year	rank
The Rainmaker (by Coppola)	1997	0.110
The Rain People (by Coppola)	1969	0.089
Rain Man (by Levinson)	1988	0.077
Finian's Rainbow (by Coppola)	1968	0.069
Tucker, Man and Dream (Coppola)	1989	0.061
Rain or Shine (by Capra)	1931	0.059
...	...	...

3 Problem Definition

We review here the basic definitions in probabilistic databases, based on [13, 4], and state our problem.

Basic Notations We write R for a relation name, $Attr(R)$ for its attributes, and $r \subseteq U^k$ for a relation instance where k is $arity(R)$ and U is a fixed, finite universe. $\bar{R} = R_1, \ldots, R_n$ is a database schema and D denotes a database instance. We write $\Gamma \models D$ when D satisfies the functional dependencies in Γ.

Probabilistic Events Let AE be a set of symbols and $Pr : AE \to [0, 1]$ a probability function. Each element of AE is called a *basic event*, and we assume that all basic events are independent. The event $\bot \in AE$ denotes the impossible event and $Pr(\bot) = 0$. A *complex event* is an expression constructed from atomic events using the operators $\wedge$, $\vee$, $\neg$. E denotes the set of all complex events. For each complex event e, let $Pr(e)$ be its probability.

Example 3.1 Consider $e = (s_1 \wedge t_1) \vee (s_2 \wedge t_1)$, and assume $Pr(s_1) = 0.8$, $Pr(s_2) = 0.5$, $Pr(t_1) = 0.6$. To compute $Pr(e)$ we construct the truth table for $e(s_1, s_2, t_1)$ and identify the entries where e is true, namely $(1, 0, 1), (0, 1, 1), (1, 1, 1)$. The three entries have probabilities $Pr(s_1)(1 - Pr(s_2))Pr(t_1) = 0.8 \times 0.5 \times 0.6 = 0.24$, $(1 - Pr(s_1))Pr(s_2)Pr(t_1) = 0.06$ and $Pr(s_1)Pr(s_2)Pr(t_1) = 0.24$ respectively. Then $Pr(e)$ is their sum, 0.54.

This method generalizes to any complex event $e(s_1, \ldots, s_k)$, but it is important to notice that this algorithm is exponential in k. This cannot be avoided: it is known that computing $Pr(e)$ is #P-complete [26] even for complex events without negation.

Probabilistic Databases A *probabilistic relation* is a relation with a distinguished *event attribute* E, whose value is a complex event. We add the superscript p to mean "probabilistic", i.e. write $R^p, r^p, \bar{R}^p, \Gamma^p$. Given R^p, we write R for its "deterministic" part, obtained by removing the event attribute: $Attr(R) = Attr(R^p) - \{E\}$. Users "see" only R, but the system needs to access the event attribute $R^p.E$. The set of functional dependencies Γ^p always contains

$$Attr(R) \to R^p.E$$

for every relation R^p, i.e. $Attr(R)$ functionally determines $R^p \cdot E$. This ensures that we don't associate two different events e_1 and e_2 to the same tuple t (instead, we may want to associate $e_1 \vee e_2$ to t).

In addition to this *tabular* representation of a probabilistic relation, we consider a *functional* representation, where a probabilistic instance r^p, of type R^p, is described by the following function $e_R : U^k \to E$, where $k = arity(R)$. When t occurs in r^p together with some event e, then $e_R(t) = e$, otherwise $e_R(t) = \bot$. Conversely, one can recover r^p from the function e_R by collecting all tuples for which $e_R(t) \neq \bot$.

The input probabilistic databases we consider have only atomic events: complex events are introduced only by query evaluation. A probabilistic relation with atomic events which satisfies the FD $R^p.E \to Attr(R)$ is called *extensional*. Otherwise, it is called *intensional*. For example, the database in Fig. 1 is an extensional probabilistic database, where the atomic events are s_1, s_2, t_1 respectively.

Semantics of a probabilistic database We give a simple and intuitive meaning to a probabilistic relation based on possible worlds. The meaning of a probabilistic relation r^p of type R^p is a probability distribution on deterministic relations r of type R, which we call the *possible worlds*, and denote $pwd(r^p)$. Let $e_R : U^k \to E$ be the functional representation of r^p. Given $r \subseteq U^k$, $Pr(r)$ is defined to be $Pr(\bigwedge_{t \in r} e_R(t)) \wedge (\bigwedge_{t \notin r} \neg e_R(t))$. Intuitively, this is the probability that exactly the tuples in r are "in" and all the others are "out". One can check that $\sum_{r \subseteq U^k} Pr(r) = 1$.

Similarly, the meaning of a probabilistic database D^p is a probability distribution on all deterministic databases D, denoted $pwd(D^p)$.

Query semantics Let q be a query of arity k over a deterministic schema $\bar{R}$. We define a very simple and intuitive semantics for the query. Users think of q as normal query on a deterministic database, but the database is given by a probability distribution rather than being fixed. As a result, the query's answer is also a probability distribution. Formally, given a query q and a probabilistic database D^p: $q^{pwd}(D^p)$ is the following probability distribution on all possible answers, $Pr_q : \mathcal{P}(U^k) \to [0, 1]$:

$$\forall S \subseteq U^k, Pr_q(S) = \sum_{D | q(D) = S} Pr(D)$$

We call this the *possible worlds semantics*. This definition makes sense for every query q that has a well defined semantics on all deterministic databases.

It is impossible to return $q^{pwd}(D^p)$ to the user. Instead, we compute a *probabilistic ranking* on all tuples $t \in U^k$, defined by the function: $rank_q(t) = \sum_S \{Pr_q(S) \mid S \subseteq U^k, t \in S\}$, for every tuple $t \in U^k$. We denote with $q^{rank}(D^p)$ a tabular representation of the function $rank_q$: this is a table with $k+1$ attributes, where the first k represent a tuple in the query's answer while the last attribute, called **Rank** is a real

number in $[0, 1]$ representing its probability.

The Query Evaluation Problem This paper addresses the following problem: given schema $\bar{R}^p, \Gamma^p$, a probabilistic database D^p and a query q over schema $\bar{R}$, compute the probabilistic rankings $q^{rank}(D^p)$.

Application to queries with uncertain predicates Consider now a deterministic database D and a query $q^{\approx}$ that explicitly mentions some uncertain predicates over base tables. We convert this problem into evaluating a deterministic query q, obtained by removing all uncertain predicates from $q^{\approx}$, on a probabilistic database, obtained by associating a probability $Pr(t)$ to each tuple t based on how well t satisfies the uncertain predicates in the query.

4 Query Evaluation

We turn now to the central problem, evaluating $q^{rank}(D^p)$ for a query q, and a probabilistic database D^p. Applying the definition directly is infeasible, since it involves iterating over a large set of database instances. Instead, we will first review the intensional evaluation of [13] then describe our approach.

We restrict our discussion first to conjunctive queries, which alternatively can be expressed as select(distinct)-project-join queries. This helps us better understand the query evaluation problem and its complexity, and will consider more complex query expressions in Sec. 7. We use either datalog notation for our queries q, or plans p in the select/project/product algebra[4]: $\sigma, \Pi, \times$.

4.1 Intensional Query Evaluation

One method for evaluating queries on probabilistic databases is to use complex events, and was introduced in [13]. We review it here and discuss its limitations. Start by expressing q as a query plan, using the operators $\sigma, \Pi, \times$. Then modify each operator to compute the event attribute E in each intermediate result: denote $\sigma^i, \Pi^i, \times^i$ the modified operators. It is more convenient to introduce them in the functional representation, by defining the complex event $e_p(t)$ for each tuple t, inductively on the query plan p:

$$\begin{aligned} e_{\sigma^i_c(p)}(t) &= \begin{cases} e_p(t) & \text{if } c(t) \text{ is true} \\ \bot & \text{if } c(t) \text{ is false} \end{cases} \\ e_{\Pi^i_{\bar{A}}(p)}(t) &= \bigvee_{t':\Pi_{\bar{A}}(t')=t} e_p(t') \qquad (1) \\ e_{p\times^i p'}(t,t') &= e_p(t) \wedge e_{p'}(t') \end{aligned}$$

The tabular definitions for $\sigma^i, \Pi^i, \times^i$ follow easily: σ^i acts like σ then copies the complex events from the input tuples to the output tuples; Π^i associates to a tuple t the complex event $e_1 \vee \ldots \vee e_n$ obtained from the complex events of all input tuples $t_1, \ldots, t_n$ that project into t; and $\times^i$ simply associates to a product tuple (t, t') the complex event $e \wedge e'$.

[4]Notice that Π also does duplicate elimination

A	B	C	D	E
'm'	1	1	'p'	$s_1 \wedge t_1$
'n'	1	1	'p'	$s_2 \wedge t_1$

(a) $S^p \bowtie^i_{B=C} T^p$

D	E
'p'	$(s_1 \wedge t_1) \vee (s_2 \wedge t_1)$

(b) $\Pi^i_D(S^p \bowtie^i_{B=C} T^p)$

D	Rank
'p'	$Pr((s_1 \wedge t_1) \vee (s_2 \wedge t_1)) = 0.54$

(c) $q^{rank}(D^p) = Pr(\Pi^i_D(S^p \bowtie^i_{B=C} T^p))$

Figure 5: Intensional Evaluation of $\Pi_D(S^p \bowtie_{B=C} T^p)$

Example 4.1 Let us consider the database D^p described in Figure 1. Consider the query plan, $p = \Pi_D(S^p \bowtie_{B=C} T^p)$. Figure 5 shows the intensional evaluation of the query (we used the tuple names as atomic events). $p^i(D^p)$ contains a single tuple $'p'$ with the event $(s_1 \wedge t_1) \vee (s_2 \wedge t_1)$.

Thus, $p^i(D^p)$ denotes an intensional probabilistic relation. It can be shown that this is independent on the particular choice of plan p, and we denote $q^i(D^p)$ the value $p^i(D^p)$ for any plan p for q, and call it the *intensional semantics*[5] of q on the probabilistic database D^p. We prove now that it is equivalent to the possible worlds semantics, $q^{pwd}(D^p)$.

Theorem 4.2. *The intensional semantics and the possible worlds semantics on probabilistic databases coincide for conjunctive queries. More precisely, $pwd(q^i(D^p)) = q^{pwd}(D^p)$ for every intensional probabilistic database D^p and conjunctive query q.*

(All proofs in this paper are available in our technical report [10].) Theorem 4.2 allows us to compute $q^{rank}(D^p)$, as follows. First compute $q^i(D^p)$, then compute the probability $Pr(e)$ for each complex event. Then $q^{rank}(D^p) = Pr(q^i(D^p))$.

Example 4.3 Fig. 5(c) shows $p^{rank}(D^p)$ for Ex. 4.1. $Pr((s_1 \wedge t_1) \vee (s_2 \wedge t_1))$ was shown in Ex. 3.1.

It is very impractical to use the intensional semantics to compute the rank probabilities, for two reasons. First, the event expressions in $q^i(D^p)$ can become very large. In the worst case the size of such an expression can become of the same order of magnitude as the database. For instance, if a projection on a table

[5]In [13] this is the only query semantics considered.

produces a single output tuple, its event expression is the disjunction of all the events in the table. This increases the complexity of the query operators significantly, and makes the task of an optimizer much harder, because now the cost per tuple is no longer a constant. Second, for each tuple t one has to compute $Pr(e)$ for its event e, which is a #P-complete problem.

4.2 Extensional Query Evaluation

We now modify the query operators to compute probabilities rather than complex events: we denote $\sigma^e, \Pi^e, \times^e$ the modified operators. This is much more efficient, since it involves manipulating real numbers rather than event expressions. We define a number $Pr_p(t) \in [0,1]$ for each tuple t, by induction on the structure of the query plan p. The inductive definitions below should be compared with those in Equations (1).

$$\begin{aligned} Pr_{\sigma^e_c(p)}(t) &= \begin{cases} Pr_p(t) & \text{if } c(t) \text{ is true} \\ 0 & \text{if } c(t) \text{ is false} \end{cases} \\ Pr_{\Pi^e_{\bar{A}}(p)}(t) &= 1 - \prod_{t':\Pi_{\bar{A}}(t')=t} (1 - Pr_p(t')) \\ Pr_{p \times^e p'}(t,t') &= Pr_p(t) \times Pr_{p'}(t') \end{aligned}$$

Again, the tabular definitions of $\sigma^e, \Pi^e, \times^e$ follow easily: σ^e acts like σ then propagates the tuples' probabilities from the input to the output, Π^e computes the probability of a tuples t as $1-(1-p_1)(1-p_2)\dots(1-p_n)$ where $p_1, \dots, p_n$ are the probabilities of all input tuples that project to t, while $\times$ computes the probability of each tuple (t,t') as $p \times p'$.

Thus, $p^e(D^p)$ is an extensional probabilistic relation, which we call the *extensional semantics* of the plan p. If we know $p^e(D^p) = q^{rank}(D^p)$, then we simply execute the plan under the extensional semantics. But, unfortunately, this is not always the case, as we saw in Sec. 2. Moreover, $p^e(D^p)$ depends on the particular plan p chosen for q. Our goal is to find a plan for which the extensional semantics is correct.

Definition 4.4. *Given a schema $\bar{R}^p, \Gamma^p$, a plan p for a query q is* safe *if $p^e(D^p) = q^{rank}(D^p)$ for all D^p of that schema.*

We show next how to find a safe plan.

4.3 The Safe-Plan Optimization Algorithm

We use the following notations for conjunctive queries:

- $Rels(q) = \{R_1, \dots, R_k\}$ all relation names occurring in q. We assume that each relation name occurs at most once in the query (more on this in Sec. 7).
- $PRels(q) =$ the probabilistic relation names in q, $PRels(q) \subseteq Rels(q)$.
- $Attr(q) =$ all attributes in all relations in q. To disambiguate, we denote attributes as $R_i.A$.
- $Head(q) =$ attributes in the result of q, $Head(q) \subseteq Attr(q)$.

Let q be a conjunctive query. We define the *induced* functional dependencies $\Gamma^p(q)$ on $Attr(q)$:

- Every FD in Γ^p is also in $\Gamma^p(q)$.
- For every join predicate $R_i.A = R_j.B$, both $R_i.A \to R_j.B$ and $R_j.B \to R_i.A$ are in $\Gamma^p(q)$.
- For every selection predicate $R_i.A = c$, $\emptyset \to R_i.A$ is in $\Gamma^p(q)$.

We seek a safe plan p, i.e. one that computes the probabilities correctly. For that each operator in p must be safe, i.e. compute correct probabilities: we define this formally next.

Let q_1, q_2 be two queries, and let $op \in \{\sigma, \Pi, \times\}$ be a relational operator. Consider the new query $op(q_1, q_2)$ (or just $op(q_1)$ when op is unary). We say that op^e is *safe* if $op^e(Pr(q_1^i(D^p)), Pr(q_2^i(D^p))) = Pr(op^i(q_1^i(D^p)), q_2^i(D^p))$ (and similarly for unary operators), $\forall D^p$ s.t. $\Gamma^p \models D^p$. In other words, op is safe if, when given correct probabilities for its inputs op^e computes correct probabilities for the output tuples.

Theorem 4.5. *Let q, q' be conjunctive queries.*

1. *σ^e_c is always safe in $\sigma_c(q)$.*
2. *$\times^e$ is always safe in $q \times q'$.*
3. *$\Pi^e_{A_1,\dots,A_k}$ is safe in $\Pi_{A_1,\dots,A_k}(q)$ iff for every $R^p \in PRels(q)$ the following can be inferred from $\Gamma^p(q)$:*

$$A_1, \dots, A_k, R^p.E \to Head(q) \qquad (2)$$

A plan p is safe iff all operators are safe.

We explain the Theorem with an example below. A formal proof can be found in our technical report [10].

Example 4.6 Continuing the example in Sec. 2, assume that both S^p and T^p are extensional probabilistic relations, hence Γ^p is:

$$\begin{aligned} S^p.A, S^p.B &\to S^p.E \\ T^p.C, T^p.D &\to T^p.E \\ S^p.E &\to S^p.A, S^p.B \\ T^p.E &\to T^p.C, T^p.D \end{aligned}$$

The last two dependencies hold because the relations are extensional. Consider the plan $\Pi_D(S^p \bowtie_{B=C} T^p)$. We have shown in Fig. 3 that, when evaluated extensionally, this plan is incorrect. We explain here the reason: the operator Π^e_D is not safe. An intuitive justification can be seen immediately by inspecting the

intensional relation $S^p \bowtie^i_{B=C} T^p$ in Fig. 5 (a). The two complex events share the common atomic event t_1, hence they are correlated probabilistic events. But the formula for Π^e_D only works when these events are independent. We show how to detect formally that Π^e_D is unsafe. We need to check:

$$\begin{aligned} T^p.D, S^p.E &\rightarrow S^p.A, S^p.B, T^p.C, T^p.D \\ T^p.D, T^p.E &\rightarrow S^p.A, S^p.B, T^p.C, T^p.D \end{aligned}$$

The first follows from Γ^p and from the join condition $B = C$, which adds $S^p.B \rightarrow T^p.C$ and $T^p.C \rightarrow S^p.B$. But the second fails: $T^p.D, T^p.E \not\rightarrow S^p.A$.

Example 4.7 Continuing the example, consider now the plan $\Pi_D(\Pi_B(S^p) \bowtie_{B=C} T^p)$. We will prove that Π^e_D is safe. For that we have to check:

$$\begin{aligned} T^p.D, S^p.E &\rightarrow S^p.B, T^p.C, T^p.D \\ T^p.D, T^p.E &\rightarrow S^p.B, T^p.C, T^p.D \end{aligned}$$

Both hold, hence Π^e_D is safe. Similarly, Π^e_B is safe in $\Pi_B(S^p)$, which means that the entire plan is safe.

Algorithm 1 is our optimization algorithm for finding a safe plan. It proceeds top-down, as follows. First, it tries to do all safe projections late in the query plan. When no more late safe projections are possible for a query q, then it tries to perform a join $\bowtie_c$ instead, by splitting q into $q_1 \bowtie_c q_2$. Since $\bowtie_c$ is the last operation in the query plan, all attributes in c must be in $Head(q)$.

Splitting q into $q_1 \bowtie_c q_2$ is done as follows. Construct a graph G whose nodes are $Rels(q)$ and whose edges are all pairs (R_i, R_j) s.t. q contains some join condition $R_i.A = R_j.B$ with both[6] $R_i.A$ and $R_j.B$ in $Head(q)$. Find the connected components of G, and choose q_1 and q_2 to be any partition of these connected components: this defines $Rels(q_i)$ and $Attr(q_i)$ for $i = 1, 2$. Define $Head(q_i) = Head(q) \cap Attr(q_i)$, for $i = 1, 2$. If G is a connected graph, then the query has no safe plans (more on this below). If G has multiple connected components, then we have several choices for splitting q, and we can deploy any standard cost based optimizations algorithm that works in top-down fashion[7].

Finally, the algorithm terminates when no more projections are needed. The remaining join and/or selection operators can be done in any order.

[6]One can show that, if $R_i.A$ is in $Head(q)$, then so is $R_j.B$. Indeed, assume $R_j.B \notin Head(q)$. Then $\Pi_{Head(q)}(q_{R_j.B})$ is safe, so we should have performed it first. Then, both $R_i.A$ and $R_j.B$ are in $Head(q_{R_j.B})$.

[7]It is also possible to adapt our algorithm to work with a bottom-up optimizer.

Algorithm 1 SAFE-PLAN(q)

```
if Head(q) = Attr(q) then
  return any plan p for q
    (p is projection-free, hence safe)
end if
for A ∈ (Attr(q) − Head(q)) do
  let q_A be the query obtained from q
    by adding A to the head variables
  if Π_{Head(q)}(q_A) is a safe operator then
    return Π_{Head(q)}(SAFE-PLAN(q_A))
  end if
end for
Split q into q_1 ⋈_c q_2 (see text)
if no such split exists then
  return error("No safe plans exist")
end if
return SAFE-PLAN(q_1) ⋈_c SAFE-PLAN(q_2)
```

Example 4.8 Continuing the example in Sec. 2, consider the original query in Fig. 2 (b), which we rewrite now as:

$$q(D) :- S^p(A, B), T^p(C, D), B = C$$

Here $Attr(q) = \{A, B, C, D\}$ and $Head(q) = \{D\}$ (we write D instead of $T^p.D$, etc, since all attributes are distinct). The algorithm first considers the three attributes A, B, C in $Attr(q) - Head(q)$, trying to see if they can be projected out late in the plan. A cannot be projected out. Indeed, the corresponding q_A is:

$$q_A(A, D) :- S^p(A, B), T^p(C, D), B = C$$

and Π^e_D is unsafe in $\Pi_D(q_A)$ because $T^p.D, T^p.E \not\rightarrow S^p.A$, as we saw in Example 4.6. However, the other two attributes can be projected out, hence the plan for q is $\Pi_D(q_{BC})$, where:

$$q_{BC}(B, C, D) :- S^p(A, B), T^p(C, D), B = C$$

Now we optimize q_{BC}, where $Attr(q_{BC}) = \{A, B, C, D\}$, $Head(q_{BC}) = \{B, C, D\}$. No projection is possible, but we can split the query into $q_1 \bowtie_{B=C} q_2$ where q_1, q_2 are:

$$\begin{aligned} q_1(B) &:- S^p(A, B) \\ q_2(C, D) &:- T^p(C, D) \end{aligned}$$

The split $q_{BC} = q_1 \bowtie_{B=C} q_2$ is indeed possible since both B and C belong to $Head(q_{BC})$. Continuing with q_1, q_2, we are done in q_2, while in q_1 we still need to project out A, $q_1 = \Pi_B(S^p)$, which is safe since $B, S^p.E \rightarrow A$. Putting everything together gives us the following safe plan: $p' = \Pi_D(\Pi_B(S^p) \bowtie_{B=C} T^p)$.

We state now the soundness of our algorithm: the proof follows easily from the fact that all projection operators are safe. We prove in the next section that the algorithm is also complete.

Proposition 4.9. *The* SAFE-PLAN *optimization algorithm is sound, i.e. any plan it returns is safe.*

5 Theoretical Analysis

We show here a fundamental result on the complexity of query evaluation on probabilistic databases. It forms a sharp separation of conjunctive queries into queries with low and high data complexity, and shows that our optimization algorithm is complete.

The data complexity of a query q is the complexity of evaluating $q^{rank}(D^p)$ as a function of the size of D^p. If q has a safe plan p, then its data complexity is in PTIME, because all extensional operators are in PTIME. We start by showing that, for certain queries, the data complexity is $\#P$-complete. $\#P$ is the complexity class of some hard counting problems. Given a boolean formula φ, counting the number of satisfying assignments, denote it $\#\varphi$, is $\#P$-complete [26]. (Checking satisfiability, $\#\varphi > 0$, is NP-complete.) The data complexity of any conjunctive query is $\#P$, since $q^{rank}(D^p) = Pr(q^i(D^p))$. The following is a variant of a result on query reliability by Gradel et al. [14]. The proof is novel and is of independent interest in our setting.

Theorem 5.1. *Consider the following conjunctive query on three probabilistic tables:*

$$q() := L^p(x), J(x,y), R^p(y)$$

Here L^p, R^p are extensional probabilistic tables and J is deterministic[8]*. The data complexity for q is $\#P$-hard.*

Proof. (Sketch) Provan and Ball [22] showed that computing $\#\varphi$ is $\#P$-complete even for *bipartite monotone 2-DNF* boolean formulas φ, i.e. when the propositional variables can be partitioned into $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$ s.t. $\varphi = C_1 \vee \ldots \vee C_k$ where each clause C_i has the form $x_j \wedge y_k$, $x_j \in X, y_k \in Y$. (The satisfiability problem, $\#\varphi > 0$, is trivially true.). Given φ, construct the instance D^p where L^p is X, R^p is Y and J is the set of pairs (x_j, y_k) that occur in some clause C_i. Assign independent probability events to tuples in L^p, R^p, with probabilities 1/2. Then $q^{rank}(D^p)$ returns a single tuple, with probability $\#\varphi/2^{m+n}$. Thus, computing $q^{rank}(D^p)$ is at least as hard as computing $\#\varphi$. □

We state now the main theoretical result in this paper. We consider it to be a fundamental property of query evaluation on probabilistic databases.

Theorem 5.2 (Fundamental Theorem of Queries on Probabilistic DBs). *Consider a schema $\bar{R}^p, \Gamma^p$ where all relations are probabilistic and Γ^p has only the trivial FDs*[9] *$Attrs(R) \rightarrow R^p.E$, $R^p.E \rightarrow Attrs(R)$, for every R^p. Let q be a conjunctive query s.t. each relation occurs at most once. Assuming $\#P \neq PTIME$ the following statements are equivalent:*

1. *The query q contains three subgoals of the form:*
$$L^p(x,\ldots), J^P(x,y,\ldots), R^p(y,\ldots)$$
where $x, y \notin Head(q)$.
2. *The data complexity of q is $\#P$-complete.*
3. *The* Safe-Plan *optimization algorithm fails to return a plan.*

Proof. (Sketch) (1) ⇒ (2) is a simple extension of Th. 5.1. (2) ⇒ (3) is obvious, since any safe plan has data complexity in PTIME. The proof of (3) ⇒ (1) is based on a detailed analysis of what happens when Safe-Plan fails: the details are in [10]. □

Theorem 5.2 provides a sharp separation of feasible and infeasible queries on probabilistic databases. It can be extended to mixed probabilistic/deterministic databases and richer functional dependencies [10].

6 Unsafe Plans

When a query's data complexity is $\#P$-complete, then Safe-Plan fails to return a plan. Since this can indeed happen in practice, we address it and propose two solutions.

6.1 Least Unsafe Plans

Here we attempt to pick a plan that is less unsafe than others, i.e. minimizes the error in computing the probabilities. Recall from Eq.(2) that $\Pi^e_{A_1,\ldots,A_k}$ is safe in $\Pi^e_{A_1,\ldots,A_k}(q)$ iff $A_1, \ldots, A_k, R^p.E \rightarrow Head(q)$ for every R^p. Let $\bar{B} = \{A_1, \ldots, A_k, R^p.E\} \cap Attr(R^p)$ (hence $R^p.E \in \bar{B}$) and $\bar{C} = Head(q) \cap Attr(R^p)$. Define R^p_{fanout} to be the expected number of distinct values of $\bar{C}$ for a fixed value of the attributes $\bar{B}$. In a relational database system, it is possible to estimate this value using statistics on the table R^p. Define the degree of unsafety of $\Pi^e_{A_1,\ldots,A_k}$ to be $max_{R^p \in PREL(Q)}(R^p_{fanout} - 1)$. Thus, a safe project has degree of unsafety 0. Also, the higher the degree of unsafety, the higher is the expected error that would result from using the extensional semantics for that project operator.

We modify Algorithm 1 to cope with unsafe queries. Recall that the algorithm tries to split a query q into two subqueries q_1, q_2 s.t. all their join attributes are in $Head(q)$. Now we relax this: we allow joins between q_1 and q_2 on attributes not in $Head(q)$, then project out these attributes. These projections will be unsafe, hence we want to minimize their degree of unsafety. To do that, we pick q_1, q_2 to be a minimum cut of the graph, where each edge representing a join condition is labeled with the degree of unsafety of the corresponding project operation[10]. The problem of finding

[8] Allowing J to be deterministic strengthens the result. The theorem remains true if J is probabilistic.

[9] Hence, the probabilistic instances are extensional.

[10] The estimator of R^p_{fanout} should make sure that the estimated value is 0 only when the FD holds, otherwise the algorithm may favor 'expected' safe plans over truly safe plans.

minimum cut is polynomial time solvable as a series of network flow problems or using the algorithm of Stoer and Wagner [23].

6.2 Monte-Carlo Approximations

As an alternative, we present an algorithm based on Monte-Carlo simulation, which can guarantee arbitrarily low errors in the probabilities of output tuples.

Given a conjunctive query q over probabilistic relations $R_1^p, R_2^p \cdots R_k^p$, let q' be its body, i.e. $Head(q') = Attr(q') = Attr(q)$ and $q = \Pi_{Head(q)}(q')$. Modify q' to also return all event attributes $\bar{E} = R_1^p.E, \ldots, R_k^p.E$. Evaluate q' over the probabilistic database, and group of tuples in the answer based on the values of their attributes $Head(q)$. Consider one such group, and assume it has n tuples $t_1, \ldots, t_n$. The group defines the following complex event expression: $\bigvee_{i=1}^n C_i$, where each C_i has the form $e_1 \wedge \ldots \wedge e_k$. We need to compute its probability, since this will be the probability of one tuple in $q^{rank}(D^p)$. For that we use the Monte Carlo algorithm described by Karp [17]: when run for $N \geq \frac{4n}{\epsilon^2} \ln \frac{2}{\delta}$ iterations, the algorithm guarantees that the probability of the error being greater that ϵ is less than δ.

7 Extensions

Additional operators So far, we have limited our discussion to conjunctive queries, or, equivalently to the algebra consisting of $\sigma, \Pi, \times$. We show now how to extend these techniques to $\cup, -, \gamma$ (union, difference, groupby-aggregate [11]. A large fragment of SQL, including queries with nested sub-queries, aggregates, group-by and existential/universal quantifiers can be expressed in this logical algebra [25]. (We omit δ (duplicate elimination) since we only consider queries with set semantics, i.e. δ is implicit after every projection and union.) We define the extensional semantics for these operators, using the functional notation.

$$
\begin{aligned}
Pr_{p \cup^e p'}(t) &= 1 - (1 - Pr_p(t)) \times (1 - Pr_{p'}(t)) \\
Pr_{p -^e p'}(t) &= Pr_p(t) \times (1 - Pr_{p'}(t)) \\
Pr_{\gamma^e_{\bar{A},\min(B)}}(t) &= Pr_p(t) \prod_{\substack{s\,:\,s.\bar{A} = t.\bar{A} \\ \wedge s.B < t.B}} (1 - Pr_p(s)) \\
Pr_{\gamma^e_{\bar{A},\max(B)}}(t) &= Pr_p(t) \prod_{\substack{s\,:\,s.\bar{A} = t.\bar{A} \\ \wedge s.B > t.B}} (1 - Pr_p(s))
\end{aligned}
$$

For example, to compute the groupby-min operator $\gamma_{A,\min(B)}(R^p)$ one considers each tuple (a, b) in R^p: the probability that (a, b) is in the output relation is $p(1 - p_1) \ldots (1 - p_n)$ where p is the probability of the tuple (a, b), while $p_1, \ldots, p_n$ are the probabilities of all other tuples (a, b') s.t. $b' < b$. In the case of SUM, the aggregated attribute may take values that are not in the input table. To compute the probabilities correctly one needs to iterate over exponentially many possible sums. Instead, we simply compute the expected value of the sum (details omitted). This is meaningful to the user if SUM appears in the SELECT clause, less so if it occurs in a HAVING clause. We treat COUNT similarly.

We now give sufficient conditions for these operators to be safe.

Theorem 7.1. *Let q, q' be conjunctive queries.*

1. *$\cup^e$ is safe in $q \cup^e q'$ if $PRels(q) \cap PRels(q') = \phi$.*
2. *$-^e$ is safe in $q \cap^e q'$ if $PRels(q) \cap PRels(q') = \phi$.*
3. *$\gamma_{\bar{A},agg(B)}$ is safe in $\gamma_{\bar{A},agg(B)}(q)$ if $\Pi_{\bar{A}}(q)$ is safe, where agg is* min *or* max.

Self-joins Self-joins on probabilistic relations may be a cause of $\#P$-complete data complexity [14]. However, a query $q^{\approx}$ with uncertain predicate rarely results in self-join. Even if the same table R occurs twice in $q^{\approx}$, the different uncertain predicates on the two occurrences generate distinct events, hence the system makes two probabilistic "copies": R_1^p, R_2^p. Of course, the Monte-Carlo algorithm works fine even in the presence of self-joins.

Extending the optimization algorithm SAFEPLAN is extended to handle each block of conjunctive queries separately. As an example, the query in Section 1, asking for an actor whose name is like 'Kevin' and whose first 'successful' movie appeared in 1995, has a safe plan as shown below:

$$
\Pi_{name}(A \bowtie_{actorid} (\sigma_{year=1995}(\gamma_{actorid,\min(year)}(\Pi_{actorid,year}C))))
$$

8 Atomic Predicates

Our main motivation is executing a query with uncertain predicates $q^{\approx}$ on a deterministic database D. As we saw, our approach is to apply the uncertain predicates first, and generate a probabilistic database D^p, then evaluate q (without the uncertain predicates). We discuss here briefly some choices for the uncertain predicates proposed in the literature. All proposals depend on a notion of closeness between two data values. This is domain dependent and can be classified into three categories:

Syntactic closeness This applies to domains with proper nouns, like people's names. Edit distances, q-grams and phonetic similarity can be employed. The excellent surveys on string matching techniques by Zobel and Dart [27] and Navarro [20] describe more than 40 techniques and compare them experimentally. Navarro also has a discussion on the probability of string matching. In our system, we used the 3-gram distance between words, which is the number of triplets of consecutive words common to both words.

Semantic closeness This applies to domains that have a semantic meaning, like film categories. A user query

[11] Following discussion assumes that a groupby-aggregate operator does not perform any projections, i.e. every attribute in the input to the operator belongs to either the groupby or the aggregate clause.

for the category 'musical' should match films of category 'opera'. Semantic distance can be calculated by using TF/IDF or with ontologies like Wordnet [2]. We do not have semantic distances in our system currently.

Numeric closeness This applies to domains like *price* and *age*. A distance can be just the difference of the values.

Once distances are defined between attributes, using any of the above methods, they need to be meaningfully converted into probabilities. We fitted a Gaussian curve on the distances as follows: the curve was centered around the distance 0 where it took value 1. The variance of the Gaussian curve is an indication of the importance of match on that attribute. Its correct value depends on the domain and user preferences. In our experiments, we used fixed, query independent values, for the variances.

Finally, one issue is when to generate new probability events. For example consider the uncertain predicate `Product.category` $\approx \ldots$ and assume there are two products with the same category. Should they result in two independent probabilistic events with the same probabilities, or in the same probabilistic events? Both choices are possible in our system. In the first case the functional dependency is $\texttt{Product}^p.\texttt{key} \rightarrow \texttt{Product}^p.E$ while in the second the FD is $\texttt{Product}^p.\texttt{category} \rightarrow \texttt{Product}^p.E$. In the latter case, $\Pi_{category}$ becomes unsafe. This can be taken care of by normalizing the resulting database to 3NF, i.e. creating a separate category table that contains the events for categories.

9 Experiments

We performed some preliminary evaluation of our probabilistic query evaluation framework, addressing four questions. How often does the Safe-Plan optimization algorithm fail to find a plan? What is the performance of safe plans, when they exists? Are naive approaches to query evaluation perhaps almost as good as a safe plan? And how effectively can we handle queries that do not have safe plans?

We did not modify the relational engine, but instead implemented a middleware. SQL queries with approximate predicates were reformulated into "extensional" SQL queries, using the techniques described in this paper, and calls to a TSQL function computing 3-gram distances. These queries were then executed by the relational engine and returned both tuples and probabilities. We used Microsoft SQL Server.

We used the TPC-H benchmark, with a database of 0.1GB. We modified all queries by replacing all the predicates in the WHERE clause with uncertain matches. The constants in the queries were either misspelled or made vague. For instance, a condition like `part.container = 'PROMO PLATED GREEN'` was replace with `part.container` $\approx$ `'GREEN PLATE'`. When executed exactly, all modified queries returned empty answers.

1. Frequency of unsafe queries In our first experiment, we wanted to see how many queries do not have safe plans. Out of the 10 TPC-H queries, 8 turned out to have safe plans. Q_7 and Q_8 were the only query that were unsafe. These also become safe if not all of their predicates are uncertain.

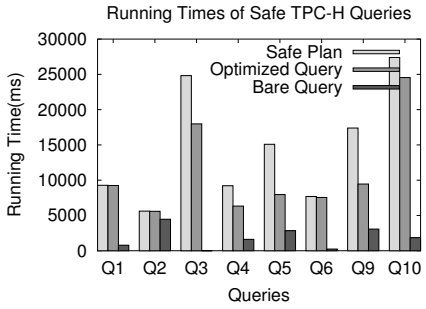

Figure 6: TPC-H Query Running Times

Figure 7: Errors on Safe TPC Queries

2. Performance Next, we measured the running times for the eight queries that have safe plans, shown in Figure 6. All times are wall-clock. The first column is the running time of the safe plan. The second column represents an optimization where at each intermediate stage, tuples with zero probability are discarded. This optimization does not affect the final answer and as we can see from the graph, it brings about considerable savings for some queries. This also suggests the use of other optimizations like an early removal of tuples with low probabilities if the user is only interested in tuples with high probability. The third column in the graph shows the time for running safe queries without taking into account the computation time for the uncertain predicate, which, in our case, is the 3-gram distance. The graphs show that most of the time is spent in computing the uncertain predicates. (For Q_3 the the running time was almost negligible.) This graph suggests that important improvements would be achieved if the predicates were implemented in the engine.

3. Naive Approaches In the next experiment we calculated the error produced by a naive extensional plan. We considered the naive plan that leaves all project operators (and the associated duplicate elimination) at the end of the plan, which are typical plans produced by database optimizers. Figure 7 shows the percentage relative error of naive plans. We only considered the 8 queries that have safe plans. The naive plans for Q_1, Q_4, Q_6 were already safe, hence had no errors (and Safe-Plan indeed returned the same plan): these queries are not shown. Queries Q_3, Q_5 and Q_{10} had large errors with Q_5 showing an average error of 150% in the tuple probabilities. Queries Q_2 and Q_9 had negligible errors. Thus, while some naive plans were bad, others were reasonable. But, in general, naive plans

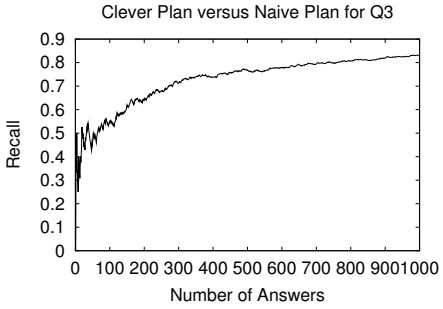

Figure 8: Recall Plot for Q_3

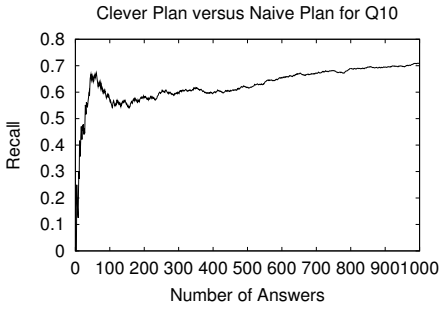

Figure 9: Recall Plot for Q_{10}

can be arbitrarily bad. However, we argue that the low extra complexity of searching for a safe plan is a price worth paying in order to avoid the (admittedly rare) possibility of arbitrarily large errors.

However, since we are only interested in *ranking* the results, not in the actual probabilities, it is worth asking whether high errors in the probabilities translate into high ranking results. We plotted the recall graphs for queries Q_3 and Q_{10} (for which the naive plan produced only medium errors). We defined recall as the fraction of answers ranked among top N by the naive plan that should actually have been in top N. We plotted this as a function of N. Figures 8 and 9 show the recall graphs. By definition, the recall approaches to 1 when N approaches the total number of possible tuples in the answer. However, as the graphs show, the recall was bad for small values of N. A user looking for top 50 or 100 answers to Q_3 would miss half of the relevant tuples. For smaller values of N (say, 10) the naive approach misses 80% of the relevant tuples.

4. Unsafe Queries Finally, we tested our approach to handle queries with no safe plans on Q_7 and Q_8. We ran the Monte Carlo simulation to compute their answer probabilities and used them as baseline. Figure 10 shows the errors in evaluating them with a naive plan and the least unsafe plan (using min-cut, Sec. 6). The graphs show that the plan chosen by the optimizer was better, or significantly better than a naive one. Still, from two data points it is hard to judge the improvement over a naive plan. To see a third data point we wrote a new unsafe query, QQ, where the relation `lineitem` is joined with `orders` and `suppliers`. Here the fanout is larger, and the difference between the naive plan and the optimal break is more pronounced.

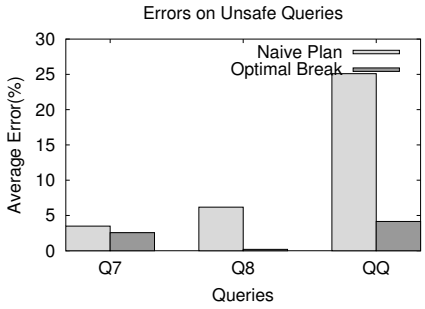

Figure 10: Errors on Unsafe Queries

10 Related Work

The possible worlds semantics, originally put forward by Kripke for modal logics, is commonly used for representing knowledge with uncertainties. Halpern, Baccus et al [11, 4] have showed the use of possible worlds semantics to assign degrees of beliefs to statements based of the probabilities in the knowledge base.

Though there has been extensive work on probabilities in AI, relatively little work has been done on probabilistic databases. There are probabilistic frameworks [7, 6, 13, 18] proposed for databases, but each makes simplifying assumptions for getting around the problem of high query evaluation complexity that lessens their applicability.

Fuhr and Rolleke [13] define probabilistic NF2 relations and introduce the intensional semantics for query evaluation. As we saw, this is correct, but impractical.

Many of these works specialize on logic programming in deductive databases. Ng and Subrahmaniam [21] extend deductive databases with probabilities and give fixed point semantics to logic programs annotated with probabilities, but they use absolute ignorance to combine event probabilities.

Non-probabilistic approaches to imprecise queries have also been considered. Keyword searches in databases are discussed in [16, 8, 15]. Fagin [12] gives an algorithm to rank objects based on its scores from multiple sources: this applies only to a single table. The VAGUE system [19] supports queries with vague predicates, but the query semantics are ad hoc, and apply only to a limited SQL fragments. Surajit et al. [3] consider ranking query results automatically: this also applies to a single table. The WHIRL system [9] computes ranked results of queries with similarity joins, but uses an extensional semantics. Theobald and Weikum [24] describe a query language for XML that supports approximate matches with relevance ranking based on ontologies and semantic similarity.

11 Conclusions

In this paper, we introduce a query semantics on probabilistic databases based on *possible worlds*. Under this semantics, every query that is well defined over a deterministic databases has a meaning on a probabilistic database. We describe how to evaluate queries efficiently under this new semantics. Our theoretical results capture fundamental properties of queries on probabilistic databases, and lead to efficient evaluation techniques. We showed how this approach can be used to evaluate arbitrarily complex

SQL queries with uncertain predicates.

References

[1] Movie database: Uci kdd archive. *http://kdd.ics.uci.edu/databases/movies/movies.html*.

[2] Wordnet 2.0: A lexical database for the english language: *http://www.cogsci.princeton.edu/ wn/*, Jul. 2003.

[3] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *Proceedings of the First Biennial Conf. on Innovative Data Systems Research*, 2003.

[4] Fahiem Bacchus, Adam J. Grove, Joseph Y. Halpern, and Daphne Koller. From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87(1-2):75–143, 1996.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[6] Daniel Barbará, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, 1992.

[7] Roger Cavallo and Michael Pittarelli. The theory of probabilistic databases. In *VLDB'87, Proceedings of 13th Int. Conf. on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 71–81, 1987.

[8] S. Chaudhuri, G. Das, and V. Narasayya. Dbexplorer: A system for keyword search over relational databases. In *Proceedings of the 18th Int. Conf. on Data Engineering, San Jose, USA*, 2002.

[9] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the 1998 ACM SIGMOD Int. Conf. on Management of data*, pages 201–212. ACM Press, 1998.

[10] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. University of Washington Technical Report (TR 04-03-04), January 2004. http://www.cs.washington.edu/research/tr/techreports.html.

[11] Ronald Fagin and Joseph Y. Halpern. Reasoning about knowledge and probability. In *Proceedings of the Second Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 277–293, San Francisco, 1988.

[12] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–113, 2001.

[13] Norbert Fuhr and Thomas Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[14] Erich Gradel, Yuri Gurevich, and Colin Hirch. The complexity of query reliability. In *Symposium on Principles of Database Systems*, pages 227–234, 1998.

[15] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *Proceedings of the 2003 ACM SIGMOD Int. Conf. on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 16–27, 2003.

[16] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proc. 28th Int. Conf. Very Large Data Bases, VLDB*, 2002.

[17] Richard Karp and Michael Luby. Monte-carlo algorithms for enumeration and reliability problems. In *Proceedings of the annual ACM symposium on Theory of computing*, 1983.

[18] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3):419–469, 1997.

[19] Amihai Motro. Vague: a user interface to relational databases that permits vague queries. *ACM Trans. Inf. Syst.*, 6(3):187–214, 1988.

[20] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[21] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[22] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983.

[23] M. Stoer and F. Wagner. A simple min cut algorithm. *Algorithms–ESA '94*, pages 141–147, 1994.

[24] Anja Theobald and Gerhard Weikum. The xxl search engine: ranked retrieval of xml data using indexes and ontologies. In *Proceedings of the 2002 ACM SIGMOD Int. Conf. on Management of data*, pages 615–615, 2002.

[25] Jeffrey D. Ullman and Jennifer Widom. *First Course in Database Systems, 2nd ed.* Prentice Hall, 1997.

[26] L. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8:410–421, 1979.

[27] J. Zobel and P. W. Dart. Phonetic string matching: Lessons from information retrieval. In *Proceedings of the 19th Int. Conf. on Research and Development in Information Retrieval*, pages 166–172, Zurich, Switzerland, 1996. ACM Press.