# Threshold Query Optimization for Uncertain Data

Yinian Qi, Rohit Jain, Sarvjeet Singh*, Sunil Prabhakar
Department of Computer Science, Purdue University
305 N. University Street, West Lafayette, IN 47907, USA
{yqi, jain29, sarvjeet, sunil}@cs.purdue.edu

## ABSTRACT

The probabilistic threshold query (PTQ) is one of the most common queries in uncertain databases, where all results satisfying the query with probabilities that meet the threshold requirement are returned. PTQ is used widely in nearest-neighbor queries, range queries, ranking queries, etc. In this paper, we investigate the general PTQ for arbitrary SQL queries that involve selections, projections and joins. The uncertain database model that we use is one that combines both attribute and tuple uncertainty as well as correlations between arbitrary attribute sets. We address the PTQ optimization problem that aims at improving the efficiency of PTQ query execution by enabling alternative query plan enumeration for optimization. We propose general optimization rules as well as rules specifically for selections, projections and joins. We introduce a threshold operator ($\tau$-operator) to the query plan and show it is generally desirable to push down the $\tau$-operator as much as possible. Our PTQ optimizations are evaluated in a real uncertain database management system. Our experiments on both real and synthetic data sets show that the optimizations improve the PTQ query processing time.

## Categories and Subject Descriptors

H.2 [**Database Management**]: Database applications; H.4 [**Information Systems Applications**]: Miscellaneous

## General Terms

Theory, performance, verification

## Keywords

Uncertain data, probabilistic data, threshold queries, query optimization

*Work done while at Purdue University. Current affiliation: Google Inc, Mountain View, California, USA.


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


## 1. INTRODUCTION

Due to the importance of uncertain data for a large number of applications, there has been significant recent interest in database support for uncertain data. Existing work in this area includes new models for uncertain data, prototype implementations, and efficient query processing algorithms. In order to provide meaningful semantics for queries over uncertain data, a large body of recent work has adopted the well-known *Possible Worlds Semantics* [10] (PWS) over probabilistic data. As with traditional data, efficient execution is necessary for ensuring the viability of uncertain data management systems. In fact, due to the complications of ensuring correct results (with respect to PWS), and the need for CPU-intensive operations over probability distributions, it is even more critical and challenging for uncertain data.

Existing work on query processing over uncertain data can be divided into two broad categories. The first is concerned with the correct evaluation of probabilities in accordance with PWS. These issues were first highlighted in [10] and have been addressed by most models proposed for uncertain data [2, 14, 23, 22]. This category of work is focused on correctness rather than efficiency of evaluation. Dalvi et. al demonstrated that evaluation plans that are correct for a given query for the standard relational model, may yield incorrect results for uncertain data (in particular, the probability values may be incorrect). They showed how "safe" plans can be generated for many, but not all queries. While these plans guarantee correct results, they may not necessarily be the most efficient. Most other work on uncertain query evaluation has endeavored to ensure correctness of query results by directly taking dependencies between data into account in the evaluation of query results using lineage [2], factor tables [22], world tables [14], or history [23]. These approaches also focus on correct evaluation without directly considering efficiency. Their advantage over the approach of [10] is that they always compute the correct result and are not limited to queries for which safe plans can be identified.

A number of works have addressed efficient evaluation for a single special type of query at a time, e.g., ranking [24, 13, 9], range [8], nearest-neighbors [6, 3], joins [7], and skyline queries [20, 29]. Most of these works leverage a probability threshold for efficient evaluation. However, they are limited to a single query and do not address complex query optimization (such as an arbitrary SQL query). On the other hand, recent work [18, 19] aims at optimizing some SQL queries, with a focus on efficient computation of exact and approximate confidences.

In this paper we address the important problem of optimizing arbitrary threshold select-project-join (SPJ) queries over uncertain data. Threshold queries represent an important class of queries over uncertain data [8] that return only those query results whose probabilities exceed a given threshold. Threshold queries are useful

for many applications where results with low probabilities are less relevant. For example, the probability of a result is indicative of our confidence in the result being true [10]. Thus low probability results are not of interest in many instances. To the best of our knowledge, this is the first work to address the issues of general threshold query optimization. The results are applicable to a broad range of uncertainty models with both discrete and continuous uncertain data.

The current approach to evaluating a general threshold SPJ query is to evaluate the query correctly and then discard those tuples that do not satisfy the threshold probability. This approach misses out on a significant optimization opportunity, similar to the "pushing selections, projections early" heuristic commonly used in databases. It may be the case that a large number of tuples that are produced by the query do not meet the threshold and are thus thrown out. The following important question remains unanswered: Is it possible to avoid spending resources on computing these "useless" tuples?

One of the major challenges in answering threshold queries is ensuring the correctness of query results. Due to the probabilistic nature of the data, results (and base data items too) often have correlations and dependencies that must not be ignored in order to ensure correct computation of result probabilities. Consequently, the question of how a threshold query for uncertain data can be optimized is not obvious. In this paper, we show how to leverage the threshold for efficient query execution over an arbitrary query plan while ensuring correct evaluation.

With the aim of being as general as possible, we have chosen to use the Orion uncertain data model proposed in [23] since it encompasses the other recent models (such as Trio [2], MayBMS [14], and MystiQ [5]) while having the advantage of handling continuous data as well as the capability of handling both tuple uncertainty and attribute uncertainty. Furthermore, the model, especially the use of history (introduced in Section 2.2), ensures that the dependencies and correlations are captured after each query operation such that confidence can be computed correctly for the query results. In addition, since this model has been implemented inside the DBMS (PostgreSQL [23]), it allows us to use our optimization in a real prototype and validate our claims of efficient query execution in a realistic system.

The key contributions of this paper are as follows:

1. We formalize the notion of threshold queries using a new threshold operator, $\tau_\theta$, as an addition to the set of standard relational algebra operators.

2. We establish query equivalences involving the threshold operator, and prove their correctness with respect to PWS over uncertain data. The optimization rules that we design are general enough to handle uncertain data with both discrete and continuous uncertainty and allow the uncertain data to have arbitrary dependencies. These equivalences are very similar to the standard equivalences used for regular relational query optimization. Thus they can easily be incorporated into existing query optimizers. The contribution of this paper lies in establishing the correctness of the equivalences that enables their use for optimization.

3. We experimentally validate (using real and synthetic data) the effectiveness of our optimization rules.

As with the Orion model [23], in this paper we do not handle duplicate elimination or set operations. Since the uncertain model allows continuous data with attribute uncertainty, the semantics of duplicates in this case is not clear. However, we could simplify our model to only handle discrete uncertainty as in [11, 21, 14] and support duplicate elimination under Orion. This is out of the scope of our current paper, and is left for future work.

The rest of this paper is organized as follows: Section 2 formally defines the probabilistic threshold query (PTQ) as well as the optimization problem; Section 3 presents general optimization rules that we propose for pruning during the PTQ execution, along with specific optimization rules for selection, projection and Cartesian product. We discuss how to use these optimization rules in PTQ query evaluation in Section 4 with an example. We present our experimental results in Section 5 on both synthetic and real data sets. The related work is given in Section 6. We conclude our paper and point out the future work in Section 7.

# 2. PROBLEM DEFINITION

We begin with running examples to illustrate the uncertain data model used in this paper [23], followed by a review of the data model, which supports probabilistic attributes and tuples with correlations. In the remainder of this paper, unless otherwise specified, the term "model" will refer to the Orion model introduced in [23]. We then formally define the probabilistic threshold queries and introduce the threshold operator. We finally state the goal for PTQ optimization considered in this paper.

## 2.1 Running Examples

Uncertain data is common in many applications such as sensor networks, data integration, location-based applications, etc. We present one example application of uncertain data below that we will revisit later, followed by a query on uncertain tables that serves as a running example throughout this paper.

EXAMPLE 2.1. *Consider an application where the speed of cars on a highway is monitored. Due to errors in measurement, the speed sensors report a range over which the actual speed is uniformly distributed. Based on the engine noise, the make and model of the car are inferred by classification programs. Often these inferences are only able to narrow down the make and model to a few options with associated confidences. For example, for a given vehicle, the make and model may be either Honda Civic, or Toyota Corolla. Note that these two fields are jointly distributed, i.e., we cannot have arbitrary combinations like Honda Corolla. This information is to be stored in a database with the following attributes:* Highway, speed, Make, *and* Model. *Table 1 shows the speed information for three cars stored using the Orion uncertainty model which is discussed below.*

EXAMPLE 2.2. *As a second, independent example, consider a relation R, with four discrete uncertain attributes $A, B, C$ and $D$, as shown in Figure 1. Attributes $A$ and $B$ are jointly distributed, as are $C$ and $D$. Each pair may represent, for example, two location coordinates, or values reported by different individuals or sensors. The example shows two tuples in R. The uncertainty in the first tuple is as follows: the values of $A$ and $B$ are* either 4 *and* 7, *respectively, with probability 0.9,* or 2 *and* 6, *respectively with probability 0.1; the values of $C$ and $D$ are* either 2 *and* 3 *with probability 0.3,* or 5 *and* 4 *with probability 0.7. The uncertainty in the second tuple is similar. Table $R_1$ and $R_2$ are derived from $R$ as follows: $R_1 = \pi_{A,C}(\sigma_{A<5}R)$, $R_2 = \pi_{B,D}(\sigma_{B<7}R)$. The following query is performed over $R_1$ and $R_2$:*

$$\pi_{R_1.C}((\sigma_{R_1.C<3}(R_1)) \bowtie_{R_1.A<R_2.B} (R_2))$$

*as shown in Figure 1 along with all intermediate results during the query evaluation. We explain below how this query is evaluated under the Orion model.*

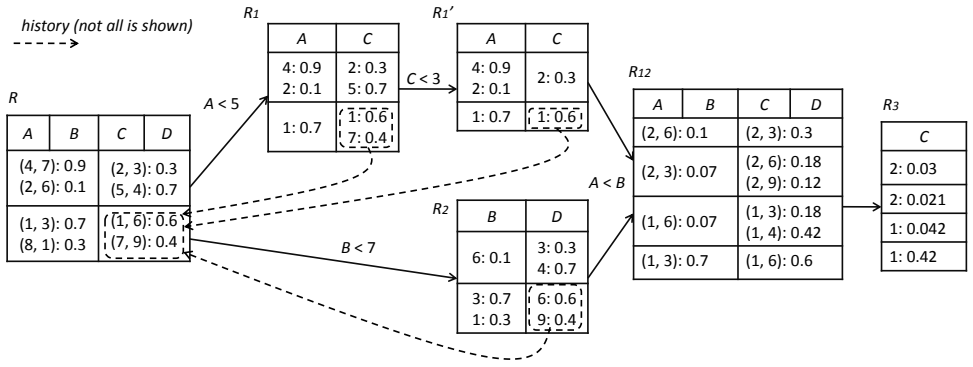


**Figure 1: Running example (the tables and the query)**

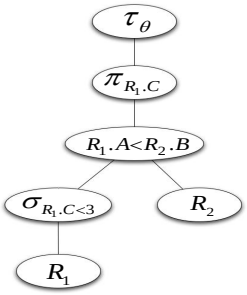


**Figure 2: PTQ plan on the running example**

| Highway | Speed (*mph*) | Make | Model |
|---|---|---|---|
| 101 | Uniform(65, 75) | (‘Honda’, ‘Civic’): 0.4<br>(‘Toyota’, ‘Corolla’): 0.2 | |
| 101 | Uniform(65, 80) | (‘BMW’, ‘Z4’): 0.3<br>(‘Ford’, ‘Mustang’): 0.3 | |
| 99 | Uniform(55, 70) | (‘Hyundai’, ‘Elantra’): 0.2<br>(‘Toyota’, ‘Camry’): 0.5 | |

**Table 1: Car speed on highways**

## 2.2 Uncertain Database Model

In this section, we briefly describe the nature of the Orion model as it is central to the contribution of this paper. Under this model, uncertainty is represented directly in a tuple using discrete and continuous probability density functions (*pdf*s). Correlations are captured in terms of joint distributions. A key aspect of the model is that it does not enumerate all possible values for an uncertain attribute or tuple (as is the case for other leading models). This enables the model to directly capture infinite possibilities (e.g., a Gaussian probability distribution) without necessarily resorting to an approximate representation. Additionally, this representation does not introduce any “holes” in the representation of a continuous attribute.

Under the Orion model, an uncertain relation $T$ is represented using a *probabilistic schema*, $(\Sigma_T, \Delta_T)$. $\Sigma_T$ is the normal relational schema (attribute names and domain types). The set of possible domains is expanded to include new data types. These data types represent continuous uncertainty (either as a symbolic representation such as a Gaussian, or a histogram), ordered discrete (e.g., integer values) and categorical or unordered discrete (e.g., colors). $\Delta_T$ captures *dependency information*. It is a partitioning of the uncertain attributes of $T$. Each partition, called a *dependency set*, declares that the attributes in that partition are jointly distributed (i.e., correlated). An uncertain attribute that is independent from all the other attributes forms its own singleton dependency set. For our car example in Table 1, $\Delta_T = \{\{Speed\}, \{Make, Model\}\}$. Similarly, for our running example in Figure 1, relation $R$'s schema defines the following dependency information: $\Delta_R = \{S_1, S_2\}$ where $S_1 = \{A, B\}$ and $S_2 = \{C, D\}$.

In the standard relational model, a tuple is a collection of exact values (one for each attribute in the schema). Under the Orion model, a tuple is a collection of exact values (one for each certain attribute, if any) and probability distributions (one for each dependency set, if any). For example, the first tuple in Table 1 consists of one certain value 101 for *Highway*, and two pdfs: Uniform(65,75) for *Speed*, and { (‘Honda’, ‘Civic’):0.4, (‘Toyota’, ‘Co-rolla’):0.2 } for $\{Make, Model\}$. This tuple represents a car on Highway 101 traveling with a speed that is uniformly distributed between 65 and 75 *mph* and is either a Honda Civic with probability 0.4, or a Toyota Corolla with probability 0.2.

From this example we can see that the model allows for missing probabilities – i.e., the sum of probability values for any distribution can be less than 1 indicating partial probability distributions[1].

In general each pdf may be multi-dimensional over any combination of uncertain domains. Given an uncertain relation $R$, a set of attributes $S$ from $R$ and a tuple $t \in R$, we define the probability of attributes $S$ in $t$ (denoted as $Pr(t.S)$) as the cumulative probability mass over the joint pdf of all pdfs in $t.S$. When it is clear from the context that $S$ is associated with $t$, we also write $Pr(t.S)$ as $Pr(S)$. The overall (tuple) probability of the tuple $t$, denoted as $Pr(t)$, is defined as the product of the cumulative probability mass of each of its dependency sets, i.e. $Pr(t) = \prod_{S \in \Delta_T} Pr(t.S)$ (here $S$ is a dependency set in $t$). Thus, for Tuple 1 in Table 1, the overall tuple probability is $1 \times 0.6 = 0.6$.

In addition to representation, a model must specify how queries are processed correctly (with respect to PWS). The major challenge for correct evaluation of probabilities is caused by dependencies among derived data [22]. The model explicitly tracks the original pdf from which each resulting pdf in a result tuple is derived. Thus for each tuple, the model stores a *history* $\Lambda$ that handles inter-tuple dependencies that result from prior database operations. History captures dependencies between dependency sets of tuples. The function $\Lambda$ maps each pdf of a dependency set $t.S$ in tuple $t$, to a set of pdfs that are its ancestors, i.e., from which the pdf of $t.S$ is derived. Only the top-level ancestors are stored, i.e., the base pdfs inserted in the database by the user (base tuples are assumed to be independent from each other). Two pdfs are called *historically independent* if their histories do not overlap, otherwise they are *historically dependent*.

To achieve correct evaluation, the model converts relational operations over uncertain attributes into operations over probability distributions. Three simple operations are defined and shown to be sufficient to support general SPJ queries: `floor`, `marginalize` and `product`.

`floor`$(f, I)$ takes an input pdf $f$ and reduces the probability to zero over all points in region $I$. It produces a partial pdf $f'$ such that values of $f'(x) = 0$ whenever $x \in I$ and $f'(x) = f(x)$ otherwise.

[1]Note that NULL values belong to each domain and can also be associated with a probability value in any pdf.

This floor operation corresponds to a selection predicate. The values in $I$ are those which do not pass the selection criteria and hence do not exist in the resulting pdf. Multiple floor operations can be successively applied over a pdf in any order and the result would be $\texttt{floor}(f, I_1 \cup ... I_k)$ regardless of the order in which they are applied. Consider for example the derivation of $R'_1$ from $R_1$ as shown in Figure 1. The query corresponds to applying a floor for the region $C \geq 3$. Since this applies only to the second dependency set, only the pdf for this dependency set is affected. Thus the first tuple in $R_1$ results in the first tuple in $R'_1$ where the pdf for $C$ is floored whenever $C \geq 3$. Similarly, the second tuple of $R_1$ results in the second tuple of $R'_1$. The schema of the resulting relation is the same as that of the input relation. The history of each resulting dependency set is equal to the corresponding source dependency set of $R$.

$\texttt{marginalize}(f, \bar{A})$ – produces the marginalized *pdf* $f'$ for a set of attributes $\bar{A}$ given their joint *pdf* $f$ with other attributes. Let $\bar{A}_f$ be the set of attributes whose pdf is $f$. Then $\bar{A} \subseteq \bar{A}_f$. We compute $f'$ as $\int_{\bar{A}_f - \bar{A}} f$. For discrete distributions, the integral is replaced by sum. The marginalization corresponds to a projection operation wherein a number of attributes are projected out. An important point to note is that the overall tuple probability does not change after marginalization. In Figure 1, $R_3$ is the result of a projection over $R_{12}$.

$\texttt{product}(f_1, f_2)$ – returns the joint *pdf* $f$ (over attribute set $S = S_1 \times S_2$) for two individual *pdf*s $f_1$ and $f_2$ (over $S_1$ and $S_2$ respectively). Two cases need to be considered. If $f_1$ and $f_2$ are historically independent, we can simply compute the joint pdf as the usual product: $f(x) = f_1(x_1)f_2(x_2)$ where $x \in S_1 \times S_2$ and $x = (x_1, x_2)$. If they are historically dependent, it is incorrect to simply take the product of the two. In this situation, we first divide the attributes in $S_1$ and $S_2$ into three sets: i) $C_j$ – the set of attributes that the common ancestors of $S_1$ and $S_2$ share with $S_1$ and $S_2$; ii) $D_1$ are those attributes of $S_1$ that are not in $C_j$; and iii) $D_2$ are those attributes of $S_2$ that are not in $C_j$. Identification of these sets is easily done by examining the history of $S_1$ and $S_2$. These three sets are independent of each other and we can use them to derive the distribution of $S$ correctly while taking the dependencies into account. To do this, we first compute their product and then apply any floor operations that were applied to derive the attribute sets in either $S_1$ or $S_2$ from $C_j$.

Product operations are essential for cross products and also for selection conditions that have predicates that go across multiple dependency sets. In Figure 1, $R_{12}$ is derived from $R'_1$ and $R_2$ by computing their product and then applying a floor. When computing the product, the joint distribution for the resulting tuples must first consider their dependencies. Thus, the last tuple in $R_{12}$ is not simply the concatenation of the last two tuples of $R'_1$ and $R_2$ since they are historically dependent as determined from their history edges that point to the same base pdfs in $R$ (only a couple of these history edges are shown for clarity). Thus instead of a simple product of the pdfs in $R'_1$ and $R_2$, we have to obtain the pdfs from the second tuple pdf in $R$, apply all the floors that were applied to get to $R'_1$ and $R_2$ (i.e., for selections $A < 5, B < 7$ and $C < 3$). This yields the product distribution. Floors corresponding to $A < B$ are then applied to this tuple to yield the tuple seen in Table $R_{12}$. In this example, $C_1 = \{A, B\}$ and $C_2 = \{C, D\}$, and both $D_1$ and $D_2$ are empty as there are no attributes in $R'_1$ or $R_2$ that are independent of this common ancestor.

## 2.3 Threshold Query Optimization

DEFINITION 1. (The Probabilistic Threshold Query (PTQ)) *Given a probability threshold $\theta$, and a regular query, a PTQ returns all tuples satisfying the query with tuple probabilities greater than or equal to $\theta$.*

| Notation | Meaning |
|---|---|
| $\theta, \tau_\theta$ | probability threshold, threshold operator |
| $c$ | selection predicate |
| $\Sigma_T$ | probabilistic schema of table $T$ |
| $\Delta_T$ | dependency sets of table $T$ |
| $t.S$ | an attribute value set $S$ in tuple $t$ |
| $Pr(t.S)$ | probability mass over pdf defined on $t.S$, also written as $Pr(S)$ if $t$ is clear from context |
| $\Lambda(t.S)$ | history of $t.S$ |
| $t'$ | a tuple in the query result that originates from $t$ |
| $A/U$ | attribute/uncertain attribute |

**Table 2: Summary of main notations**

EXAMPLE 2.3. *In Example 2.1, suppose the speed limit on Highway 101 is 70 miles per hour (*mph*), the local police want to find all speeding cars with probability at least 0.4. This is a PTQ where $\theta = 0.4$. To answer the query, we first find out all cars on Highway 101: Tuple 1 and Tuple 2, then compute their tuple probabilities after the selection '$Speed > 70$' is performed, which are 0.5*0.6=0.3 and 2/3*0.6=0.4, respectively. Note that the tuple probability is computed from the two dependency sets ({Speed}, {Make, Model}) after the selection on $Speed$ "floors" out part of the uniform distribution where $Speed \leq 70$. The result of this PTQ is Tuple 2.*

In this example, we take a two-stage approach for the PTQ execution: First we obtain the tuples satisfying the query (Tuple 1 and 2), then among the resulting tuples, we choose those whose probabilities meet the threshold (Tuple 2). We call the first stage "*evaluation stage*" and the second "*pruning stage*". However, for complicated queries, this direct approach can be very inefficient. As with other query operators (e.g., selection, projection), we could perform significantly better if we could prune out tuples at early stages of the query evaluation based upon the probability threshold operator. This can be especially beneficial for uncertain data for which probability computations can be CPU-intensive.

Our solution is to treat the threshold as a regular algebra operator and study its relationship to the standard operators (viz. selection, project, and join). The goal is to identify equivalences involving this new operator that allow us to enumerate alternative plans that are guaranteed to give the same results for uncertain data as a starting plan. This is exactly how regular queries are optimized. As a first step, we introduce the threshold operator:

DEFINITION 2. (The Threshold Operator) *The threshold operator $\tau_\theta$ when applied on an input relation, only retains those tuples with tuple probabilities greater than or equal to the threshold $\theta$. Formally, we have $\tau_\theta(R) = \{t | t \in R \wedge Pr(t) \geq \theta\}$, where $t$ is a tuple and $R$ is a relation.*

We can apply the threshold operator after selections, projections, and Cartesian products on uncertain relations as follows (let $t'$ be a tuple in the resulting table):

$$\tau_\theta(\sigma_c(R)) = \{t' | t' \in \sigma_c(R) \wedge Pr(t') \geq \theta\}$$
$$\tau_\theta(\pi_{\bar{A}}(R)) = \{t' | t' \in \pi_{\bar{A}}(R) \wedge Pr(t') \geq \theta\}$$
$$\tau_\theta(R_1 \times R_2) = \{t' | t' \in R_1 \times R_2 \wedge Pr(t') \geq \theta\}$$

where $c$ is the selection predicate (i.e., condition), $\bar{A}$ is a set of attributes in $R$, and $R_1$ and $R_2$ are two relations.

In this paper we are interested in identifying equivalences among the standard algebra operators and the new threshold operator $\tau_\theta$. The goal is to enable enumeration of alternative plans that can be exploited by an optimizer. The key idea is that pushing the $\tau_\theta$ operator earlier in a plan could potentially result in a more efficient plan by reducing the number of tuples that need to be evaluated.

The query plan can be viewed as a tree with the root being the last operation to perform. The threshold operator $\tau_\theta$ sits at the root to filter out results that satisfy the query but do not meet the threshold requirement. This is illustrated in Figure 2, which is the PTQ version of our running example (see Example 2.2). We can think of the PTQ optimization process as one that "trickles down" $\tau_\theta$ along the tree so that unqualified tuples are pruned earlier at lower levels of the query plan tree.

EXAMPLE 2.4. *As seen in Example 2.2, the original query in Figure 1 before applying any threshold operators can be defined as*

$$\pi_{R_1.C}((\sigma_{R_1.C<3}(R_1)) \bowtie_{R_1.A<R_2.B} (R_2))$$

*Its PTQ version is to return all tuples with probabilities at least $\theta$ after the original query is executed. Notice that if we can place the threshold operator before the join and successfully prune tuples from either $R_1$ or $R_2$, the expensive join execution will be much more efficient since there are less tuples to evaluate for the join predicate. We will later prove that such pruning does not prune away any potential result and will come back to the example for a more detailed discussion in Section 3.*

In summary, the task of the PTQ optimization is to decide where to put $\tau_\theta$ in the query plan to preserve the correctness of the query result while maximizing the pruning of unqualified tuples. Before considering complex queries with multiple operators, we first study the optimization problem for individual operators in Section 3. The main notations used in our paper are summarized in Table 2.

# 3. OPTIMIZATION RULES

In this section, we give the optimization rules for PTQ based on selection ($\sigma$), projection ($\pi$), Cartesian product ($\times$) and join ($\bowtie$). The idea is to perform the threshold pruning at earlier stages during the query execution so that tuples that cannot meet the threshold can be discarded without further evaluation. Note that among the five basic operations for relational algebra, we only discuss three in this paper (selection, projection and Cartesian product), because the set difference and union require a clear definition for equality of two tuples with uncertain attributes, which is beyond the scope of this paper and is left for future work.

## 3.1 General Rules

We first give general optimization rules for threshold queries and their correctness proofs, from which specific optimization rules for query operators can be deduced.

**Optimization Rule 1.** $\tau_\theta(op(R)) = \tau_\theta(op(\tau_\theta(R)))$, where *op* stands for an operator ($\sigma$ or $\pi$), i.e., we can apply the threshold operator to the relation $R$ first to filter out tuples with a tuple probability less than $\theta$ before evaluating *op*.

PROOF. Let $t$ be a tuple in $R$ and $Pr(t)$ be the tuple probability of $t$. Let $t' = op(t)$. For $t'$ to be a tuple in the PTQ result, the tuple probability $Pr(t')$ after evaluating the operator must be at least $\theta$. Since $Pr(t') \leq Pr(t)$, we can prune $t$ immediately if $Pr(t) < \theta$. □

When executing a PTQ with threshold $\theta$, we can first apply $\tau_\theta$ to $R$, thus saving efforts to evaluate the query operator for tuples whose probabilities are already below $\theta$. The efficiency can be significantly improved especially when the query operator is expensive to perform (e.g. selection with a complex predicate).

EXAMPLE 3.1. *In Table 1 (call it R), suppose we want to find all Toyotas driving at a speed over 70 mph with probability at least 0.7. This PTQ can be written as: $\tau_{0.7}(\sigma_{Speed>70 \wedge Make='Toyota'} R)$. With Optimization Rule 1, we can immediately prune the first two tuples without evaluating the selection predicates, as their tuple probabilities (both 0.6) are already below the threshold.*

To further improve the efficiency of executing threshold queries, we can leverage the indexing techniques. For example, a $B$-tree built on the tuple probabilities can facilitate the inner threshold pruning on the original relation $R$ to avoid sequential scanning of all tuples, which brings down the complexity of pruning based on Optimization Rule 1 from $O(n)$ to $O(\log n)$, where $n$ is the number of tuples in $R$. However, the index structure must be updated whenever there are deletions and insertions of tuples. Furthermore, the index must also be updated whenever the probabilities of uncertain attributes change.

Next we introduce a theorem upon which many of our optimization rules are built. It lays a solid foundation for ensuring the correctness of many PTQ optimization rules that we present later in the paper. It also ensures the safety to avoid tracking dependencies between attribute sets in many cases, which greatly simplifies the PTQ optimization for uncertain data with arbitrary correlations.

THEOREM 3.2. *Given two arbitrary sets of attributes that are disjoint, the probability of the cross product of the two sets is no more than the probability of either set. Formally, let $S_1$ and $S_2$ be two arbitrary attribute sets in tuple $t_1$ and $t_2$ respectively ($S_1 \subseteq t_1$, $S_2 \subseteq t_2$, and $t_1$ and $t_2$ can come from different relations). Let $Pr(S_1, S_2)$ be the probability of the cross product of the two sets and $Pr(S_1)$, $Pr(S_2)$ be the probability of $S_1$ and $S_2$ respectively (see Section 2.2 for the probability definitions). Then we have: $Pr(S_1, S_2) \leq \min(Pr(S_1), Pr(S_2))$.*

PROOF. Our proof consists of a proof for $Pr(S_1, S_2) \leq Pr(S_1)$ and a proof for $Pr(S_1, S_2) \leq Pr(S_2)$, from which we can deduce that $Pr(S_1, S_2) \leq \min(Pr(S_1), Pr(S_2))$. For simplicity of notations, we write $Pr(S_{12})$ instead of $Pr(S_1, S_2)$ in our proof. We only show the proof for $Pr(S_{12}) \leq Pr(S_1)$ below, as the proof for $Pr(S_{12}) \leq Pr(S_2)$ is exactly the same. Without loss of generality, we partition $S_i$ ($i \in \{1, 2\}$) into two parts[2]:

- *Historically dependent attributes:* $C_j, 1 \leq j \leq m$, where $C_j = N_j \cap S_{12}$, $N_j$ is a common ancestor of $S_1$ and $S_2$ (i.e., $N_j \in \Lambda(S_1) \cap \Lambda(S_2)$, the intersection of the histories of $S_1$ and $S_2$), and $m$ is the number of such common ancestors. Thus $C_j$ is the set of attributes that the ancestor $N_j$ shares with either $S_1$ or $S_2$.
- *Historically independent attributes:* $D_i = S_i - \bigcup C_j$ is the set of attributes in $S_i$ that are not shared with any common ancestor of $S_1$ and $S_2$.

Let $X_S^t$ be the random variable for an attribute set $S$ in $t$. Let $x_S^t$ be an instance of $X_S^t$. If $t$ is omitted in $X_S^t$ (i.e., $X_S$), we mean the random variable for the attribute set in $S$. Particularly, if $S$ refers to $C_j$, we interpret $t.C_j$ as the common attribute set between tuple $t$ and the ancestor $C_j$. Let $f(x_S^t)$ be the *pdf* of $x_S^t$, then we have:

$$f(x_{S_{12}}) = \begin{cases} 0 & \text{if } f(x_{S_i}^{t_i}) = 0 \\ f(x_{D_1}^{t_1}) f(x_{D_2}^{t_2}) \prod_{j=1}^{m} f(x_{C_j}) & \text{otherwise} \end{cases}$$

[2]See Section 2.2 for definitions of historical dependency

$$f(x_{S_1}^{t_1}) = f(x_{D_1}^{t_1}) \prod_{j=1}^{m} f(x_{C_j}^{t_1})$$

With the above *pdf*, we can compute the probability of the set $S_{12}$ and $S_1$ respectively as follows. Note that the attribute sets $D_1$, $D_2$ and $C_j$, $\forall j$ are independent of each other.

$$\begin{aligned} Pr(S_{12}) &= \int f(x_{S_{12}}) dx_{S_{12}} \\ &= \int_{f(x_{S_i}^{t_i})\neq 0} f(x_{D_1}^{t_1}) f(x_{D_2}^{t_2}) \prod_{j=1}^{m} f(x_{C_j}) \\ &= \int f(x_{D_1}^{t_1}) \int f(x_{D_2}^{t_2}) \prod_{j=1}^{m} \int f(x_{C_j}) \\ &= Pr(t_1.D_1) Pr(t_2.D_2) \prod_{j=1}^{m} Pr(C_j) \end{aligned} \quad (1)$$

Note that (1) $\leq Pr(t_1.D_1) \prod_{j=1}^{m} Pr(C_j)$.

Likewise, we can compute $Pr(S_1)$ as follows:

$$\begin{aligned} Pr(S_1) &= Pr'(t_1.D_1) \prod_{j=1}^{m} Pr'(t_1.C_j) \\ &= Pr'(t_1.D_1) \prod_{j=1}^{m} Pr'(C_j) \end{aligned} \quad (2)$$

Note that although $t_1.C_j \subseteq C_j$, their total probabilities are the same (this can be easily proved by integrating over their respective *pdf*s, where $f(x_{C_j}^{t_1})$ is the marginalized *pdf* obtained from $f(x_{C_j})$).

Comparing (1) with (2), we notice that $Pr(t_1.D_1) \leq Pr'(t_1.D_1)$ and $Pr(C_j) \leq Pr'(C_j)$ due to more floors when computing $Pr(S_{12})$ than computing $Pr(S_1)$ (when computing $Pr(S_{12})$, we need to consider floors resulting from selection predicates to obtain both $S_1$ and $S_2$ while we only consider floors to get $S_1$ when computing $Pr(S_1)$, i.e., the former considers either $f(x_{S_1}^{t_1}) = 0$ or $f(x_{S_2}^{t_2}) = 0$ while the latter considers only $f(x_{S_1}^{t_1}) = 0$), we have:

$$(1) \leq Pr(t_1.D_1) \prod_{j=1}^{m} Pr(C_j) \leq Pr'(t_1.D_1) \prod_{j=1}^{m} Pr'(C_j)$$

i.e., $Pr(S_{12}) \leq Pr(S_1)$.

*Note*: If $S_1$ and $S_2$ are from the same tuple (i.e., $t_1 = t_2$) and are dependent within the tuple, we can think of their common ancestor $N_j$ as their dependency set in the tuple, and the rest of the proof is the same as the above. □

Theorem 3.2 empowers us to avoid tracking histories and dependencies between attribute sets when pruning – we can always prune according to either $S_1$ or $S_2$ regardless of whether the two sets are correlated or how they are correlated.

EXAMPLE 3.3. *As a concrete example to illustrate the use of Theorem 3.2, let us revisit our running example in Figure 1 and 2. We have explained how to prune intuitively in Example 2.4 without giving the reason why the pruning is correct, now let us examine the pruning more closely. The reason that we can discard the first tuple in $R_1'$ (call it $t_{11}'$) as well as the first tuple in $R_2$ (call it $t_{21}$), hence avoiding the join operation that would have otherwise produced Tuple 1 through Tuple 3 in $R_{12}$, is that by Theorem 3.2, the probability of any tuple $t$ in $R_{12}$ containing either $t_{11}'$ or $t_{21}$ must not exceed the probability of $t_{11}'$ or $t_{21}$ themselves (both below the threshold). Since projections do not change the tuple probabilities (see Section 3.3 for details), the tuples in $R_3$ projected from Tuple 1 to Tuple 3 in $R_{12}$ also have probabilities below the threshold, hence cannot be in the final results. Note that in pruning Tuples 1 to 3 in $R_{12}$, we do not need to worry about the dependency between attributes A and B or that between C and D. Using Theorem 3.2, we simply prune based on the probabilities of $t_{11}'$ and $t_{21}$.*

From Theorem 3.2, we obtain the following corollary:

COROLLARY 3.4. *Given a tuple $t$ and any set of attributes $t.S \subseteq t$, we have $Pr(t) \leq Pr(t.S)$.*

PROOF. The tuple probability $Pr(t) = Pr(t.S, t.S')$ where $t.S' \cup t.S = t$ and $t.S' \cap t.S = \emptyset$. From Theorem 3.2, we know that $Pr(t) \leq \min(Pr(t.S), Pr(t.S')) \leq Pr(t.S)$. □

The optimization rule below can be deduced immediately from Corollary 3.4:

**Optimization Rule 2.** Given table $T(\Sigma_T, \Delta_T)$ and PTQ with threshold $\theta$, $\forall S_i \in \Delta_T$ in tuple $t$, $Pr(t.S_i) < \theta \Rightarrow Pr(t) < \theta$.

In other words, if there exists any dependency set with probability below $\theta$, we can immediately prune the tuple away knowing that there is no way for the whole tuple to meet the threshold.

## 3.2 Selection

For selection operator $\sigma$, Optimization Rule 1 and Rule 2 both apply. We can first use them to prune away tuples without evaluating the selection predicate. For the remaining tuples, however, we have to compute the final probability that the tuple satisfies the predicate. Our optimization goal here is then to estimate this probability earlier to facilitate pruning.

Let $S_c$ be the set of attributes involved in the selection predicate $c$. We refer to the probability that $c$ holds for attributes $S_c$ in tuple $t$ as $Pr(t.c)$. Note that $Pr(t.c)$ is not a tuple-level probability; rather, it is a probability that is computed solely from $t.S_c$. The following optimization rule holds for any selection predicate $c$ (let $t' = \sigma_c(t)$ where $t \in R$):

**Optimization Rule 3.** $Pr(t.c) < \theta \Rightarrow Pr(t') < \theta$.

PROOF. We first compute $Pr(t')$ from $Pr(t)$ as follows:

$$Pr(t') = \frac{Pr(t)}{Pr(t.S_c)} \cdot Pr(t.c)$$

The formula is based on the fact that the only difference between $Pr(t')$ and $Pr(t)$ results from the requirement that $c$ should hold. From Corollary 3.4, we have $Pr(t) \leq Pr(t.S_c)$, hence $Pr(t') \leq Pr(t.c) < \theta$. □

From Rule 3, we obtain the following equivalence: $\tau_\theta(\sigma_c(R)) = \tau_\theta(\sigma_{\tau_\theta(c)}(R))$, where $\tau_\theta(c)$ means applying the threshold operator to $Pr(t.c)$ for relation $R$. Now we seek further optimizations based on the form of the predicate $c$:

### 3.2.1 Simple Predicate

A simple predicate $c$ involves at least one uncertain attribute (e.g. $U, U'$) and has one of the following forms: i) $U$ *op* $k$ ii) $U$ *op* $A$ iii) $U$ *op* $U'$ , where $A$ is a certain attribute, $k$ is a constant number and *op* is a comparison operation. For a simple predicate $c$, we use Optimization Rule 3 for pruning. To further improve the efficiency of pruning, we can build a Probabilistic Threshold Index (PTI) on the uncertain attribute $U$ [8]. PTI is built based on the concept "$x$-bound" proposed by Cheng et al. [7], which is a probability bound maintained in the nodes of an R-tree based index to facilitate pruning for probabilistic threshold range queries. Such an index exploits both the range predicate over an attribute and the threshold predicate over the probability of the attribute within the range.

### 3.2.2 Complex Predicate

$c$ is a boolean combination of predicates $c_1$ and $c_2$ using $AND(\wedge)$, $OR(\vee)$, $NOT(\neg)$. We discuss the optimization for each combination below.

i) $c_1 \wedge c_2$: The following rule holds in this case.

**Optimization Rule 4.** Given a PTQ $\tau_\theta(\sigma_{c_1 \wedge c_2}(R))$ and a tuple $t'$ in the result table originated from tuple $t$ in $R$, $Pr(t.c_1) < \theta \vee Pr(t.c_2) < \theta \Rightarrow Pr(t') < \theta$.

PROOF. Let $Pr(t.c_i) < \theta$ for some $i \in \{1, 2\}$, then $Pr(t.(c_1 \wedge c_2)) \leq Pr(t.c_i) < \theta$. Let $c = c_1 \wedge c_2$, we have $Pr(t.c) < \theta$. By Optimization Rule 3, we conclude that $Pr(t') < \theta$. □

From Rule 4, we can deduce the equivalence:

$$\begin{aligned}\tau_\theta(\sigma_{c_1 \wedge c_2}(R)) &= \tau_\theta(\sigma_{\tau_\theta(c_1)}(\sigma_{\tau_\theta(c_2)}(R))) \\ &= \tau_\theta(\sigma_{\tau_\theta(c_2)}(\sigma_{\tau_\theta(c_1)}(R)))\end{aligned}$$

EXAMPLE 3.5. *Back to Example 3.1. The two selection predicates are $Speed > 70$ and $Make$ = 'Toyota'. As seen earlier, with Optimization Rule 1, we can prune the first two tuples away with the 0.7 threshold. By further computing $Pr(Make$ = 'Toyota'$) = 0.5$ for the last tuple, we can immediately claim that no tuple in the relation satisfies this PTQ.*

ii) $\neg c_1$: Let $S_{c_1}$ be the set of attributes in $c_1$, then we have:

**Optimization Rule 5.** Given a PTQ $\tau_\theta(\sigma_{\neg c_1}(R))$ and a tuple $t$, if $Pr(t.c_1) > 1 - \theta$ or $Pr(t.c_1) > Pr(t.S_{c_1}) - \theta$, then $Pr(t') < \theta$.

PROOF. We first compute $Pr(t')$ from $Pr(t)$:

$$Pr(t') = \frac{Pr(t)}{Pr(t.S_{c_1})} \cdot (Pr(t.S_{c_1}) - Pr(t.c_1))$$

Since $Pr(t) \leq Pr(t.S_{c_1})$ from Corollary 3.4, we have:

$$Pr(t') \leq Pr(t.S_{c_1}) - Pr(t.c_1) \leq 1 - Pr(t.c_1) \qquad (3)$$

If either $Pr(t.c_1) > 1 - \theta$ or $Pr(t.c_1) > Pr(t.S_{c_1}) - \theta$ holds, then (3) $< \theta$. Hence $Pr(t') < \theta$. □

iii) $c_1 \vee c_2$: Since $c_1 \vee c_2 = \neg(\neg c_1 \wedge \neg c_2)$, the probability

$$\begin{aligned}&Pr(t.c_1 \vee c_2) = Pr\,(t.(\neg(\neg c_1 \wedge \neg c_2))) \\ &= Pr(t.S_{c_1}, t.S_{c_2}) - Pr(t.(\neg c_1 \wedge \neg c_2)) \leq Pr(t.S_{c_1}, t.S_{c_2})\end{aligned}$$

where $S_{c_1}$ and $S_{c_2}$ are the set of attributes in $c_1$ and $c_2$. Intuitively, probability $Pr(t.S_{c_1}, t.S_{c_2})$ is the joint probability mass of the attributes involved in $c_1$ and $c_2$ without imposing either predicate – applying $c_1$ or $c_2$ will only decrease this probability.

**Optimization Rule 6.** Given a PTQ $\tau_\theta(\sigma_{c_1 \vee c_2}(R))$ and a tuple $t$, $Pr(t.S_{c_1}, t.S_{c_2}) < \theta \Rightarrow Pr(t') < \theta$.

PROOF. We know from the above equation that $Pr(t.(c_1 \vee c_2)) \leq Pr(t.S_{c_1}, t.S_{c_2}) < \theta$. Let $c = c_1 \vee c_2$. From Rule 3, we conclude that $Pr(t') < \theta$. □

EXAMPLE 3.6. *Suppose a tuple $t$ in relation $R$ with two uncertain attributes $a$ {2: 0.1, 4: 0.2} and $b$ {1: 0.5, 2: 0.1}. Consider PTQ $\tau_{0.2}(\sigma_{c_1 \vee c_2}(R))$ where $c_1$ is $a > 3$ and $c_2$ is $b < 2$. Since $Pr(a, b) = 0.3 \times 0.6 = 0.18 < 0.2$, we can immediately discard $t$ without evaluating the predicates.*

The corollary below follows from Rule 6 and Theorem 3.2:

COROLLARY 3.7. *Given a PTQ $\tau_\theta(\sigma_{c_1 \vee c_2}(R))$ and a tuple $t$, $Pr(t.S_{c_1}) < \theta \vee Pr(t.S_{c_2}) < \theta \Rightarrow Pr(t') < \theta$.*

PROOF. From Theorem 3.2, we know that

$$\begin{aligned}Pr(t.S_{c_1}, t.S_{c_2}) &\leq \min(Pr(t.S_{c_1}), Pr(t.S_{c_2})) \\ &\leq Pr(t.S_{c_i}) < \theta\end{aligned}$$

where $i \in \{1, 2\}$. From Rule 6, we know $Pr(t') < \theta$. □

## 3.3 Projection

For projections $\pi_{\bar{A}}$, where $\bar{A}$ is the set of attributes to be projected, let $Pr(t)$ and $Pr(t')$ be the tuple probability of $t$ before and after projection, we introduce the lemma below, which comes from [23] and is also clear from the possible world semantics:

LEMMA 3.8. *For a given tuple $t$, any projection on $t$ does not change the tuple probability.*

From Optimization Rule 1 and Lemma 3.8, we can easily deduce the following optimization rule for projections:

**Optimization Rule 7.** For threshold queries based on projections, we have $\tau_\theta(\pi_{\bar{A}}(R)) = \pi_{\bar{A}}(\tau_\theta(R))$.

The above rule can be regarded as a special case of Rule 1, where the outer $\tau_\theta$ is no longer needed since the projection does not change the tuple probability, if no duplicate elimination is enforced. Otherwise, the above optimization rule does not hold, as the probabilities after duplicate elimination may increase.

## 3.4 Cartesian Product

Cartesian product between two relations $R_1$ and $R_2$ is one of the most expensive operators. If $R_1$ has $m$ tuples and $R_2$ has $n$ tuples, the complexity of performing Cartesian product is $O(mn)$. Our optimization goal is then to reduce the number of tuples that need to be evaluated from either relation and prune away as many tuples as possible without dropping any potential result. Let $t_1$, $t_2$ be tuples in $R_1$ and $R_2$. Let $t_{12}$ be a tuple in $R_1 \times R_2$. We have:

**Optimization Rule 8.** If $Pr(t_1) < \theta$ or $Pr(t_2) < \theta$, then $Pr(t_{12}) < \theta$.

PROOF. From Theorem 3.2, we know that

$$Pr(t_{12}) = Pr(t_1, t_2) \leq \min(Pr(t_1), Pr(t_2))$$

If either $Pr(t_1)$ or $Pr(t_2)$ is below $\theta$, then $Pr(t_{12}) < \theta$. □

From Rule 8, we obtain the equivalence $\tau_\theta(R_1 \times R_2) = \tau_\theta(\tau_\theta(R_1) \times \tau_\theta(R_2))$. By applying this rule, we may filter out a large number of tuples from $R_1$ and $R_2$ before performing $R_1 \times R_2$. If the two relations are huge, this reduces many I/O operations that are otherwise unavoidable during the Cartesian product execution, thus making PTQ processing much more efficient.

## 3.5 Join

The join operator $\bowtie_c$ can be considered as a selection after performing the Cartesian product, i.e., $R_1 \bowtie_c R_2 = \sigma_c(R_1 \times R_2)$, where $R_1$ and $R_2$ are two relations. We can employ optimization rules for Cartesian product and selection to do the join. Moreover, if $c$ only involves attributes from a single relation $R$, we can perform $\sigma_c(R)$ before the Cartesian product to reduce the number of tuples from $R$ that need to be checked.

# 4. PLAN OPTIMIZATION

Now that we have optimization rules for individual operators, we can apply them to queries with combined operators. Let us first review our running example (let $\theta = 0.4$) in Figure 2. The query is: $\tau_{0.4}(\pi_{R_1.C}((\sigma_{R_1.C<3}(R_1)) \bowtie_{R_1.A<R_2.B} R_2))$. Let predicate $c_1$ be $R_1.C < 3$ and $c_{12}$ be $R_1.A < R_2.B$. Using Optimization

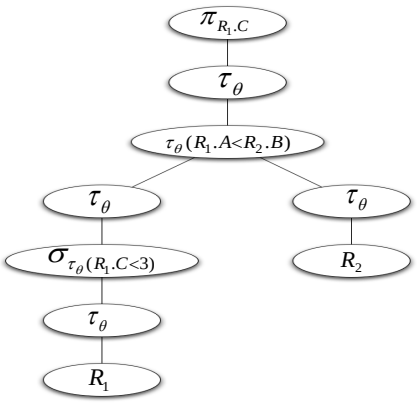


**Figure 3: The PTQ query plan after optimizations**

| TS | SID | $xpos$ | $ypos$ |
|---|---|---|---|
| 14:16:20.50 | 2242 | prod(Gaus(327, 20), Gaus(296, 20)) | |
| 14:16:20.50 | 2243 | prod(Gaus(338, 61), Gaus(293, 61)) | |
| 14:16:20.50 | 2244 | prod(Gaus(319, 17), Gaus(110, 17)) | |
| 14:16:20.50 | 2245 | prod(Gaus(315, 19), Gaus(101, 19)) | |
| 14:16:20.50 | 2246 | prod(Gaus(327, 42), Gaus(287, 42)) | |

**Table 3: Sensor Data Set Schema**

| TupleID | $xpos$ | $ypos$ |
|---|---|---|
| 1 | (11.34, 978.30): 0.12<br>(13.74, 965.47): 0.24<br>(9.68, 972.12): 0.15 | |
| 2 | (120.89, 201.55): 0.40<br>(119.45, 195.72): 0.38 | |

**Table 4: Synthetic Data Set Schema**

Rules 1, 3, 7 and 8, we "trickle down" $\tau_{0.4}$ along the query plan tree and equivalently, we execute the following plan $P$ instead:

$$P = \pi_{R_1.C}(\tau_{0.4}(\sigma_{\tau_{0.4}(c_{12})}(\tau_{0.4}(\sigma_{\tau_{0.4}(c_1)}(\tau_{0.4}(R_1)))\times\tau_{0.4}(R_2)))) \quad (4)$$

Note that we execute $\tau_{0.4}$ from inside out ($\tau_{0.4}$s in inner parentheses are executed first): If a tuple can be pruned with some inner $\tau_{0.4}$, we can discard the tuple right away without completing the whole evaluation. The query plan tree corresponding to Equation 4 is shown in Figure 3. We show below how various optimization rules work together in pruning for a complicated PTQ with selections, projections and joins all present:

In Equation 4, though nothing is pruned by executing $\tau_{0.4}(R_1)$ (See Figure 1: The two tuples of $R_1$ have probabilities 1 and 0.7 each), we can use $\tau_{0.4}(c_1)$ to prune the first tuple of $R_1'$ away, as a result of Optimization Rule 3. The first tuple of $R_2$ is also pruned away with Rule 1 by executing $\tau_{0.4}(R_2)$. The pruning of tuples in $R_1'$ and $R_2$ before the join is a result of applying Rule 8, which leaves us with only one join operation to do: Joining Tuple 2 from $R_1'$ and Tuple 2 from $R_2$. Tracing back to their history in Figure 1, we correctly compute the joining tuple to be:

| A | B | C | D |
|---|---|---|---|
| (1, 3): 0.7 | | (1, 6): 0.6 | |

Its tuple probability is $0.7 * 0.6 = 0.42 > 0.4$. According to Optimization Rule 7, the projection does not change the tuple probability. We hence return the projected tuple as the final answer (Tuple 4 of $R_3$ in Figure 1).

As we can see from Section 3, our contributions lie in providing a new way of optimizing probabilistic threshold queries that is general, and similar to the traditional approach for optimization. The merit of the approach is not in the complexity of the proposed rules, rather in their simplicity and easy applicability while ensuring correct evaluation with respect to possible world semantics.

Generally, given a query plan $P$ of a PTQ in which the threshold is placed at the end of the query, our goal for PTQ optimization is to find an equivalent query plan $P'$ such that we can prune as many tuples as possible by leveraging the threshold (i.e., applying the threshold operator) during the query evaluation while keeping the total cost low. Our approach is to start with a plan $P$ that is guaranteed to be safe (through the use of histories and dependency sets) and then apply optimization rules to generate equivalent plans that are also all guaranteed to be safe (the rules ensure correct evaluation). Since there are usually multiple optimization rules applicable for $P$, we can generate multiple equivalent plans by applying different sets of rules or by applying the same set of rules in different order. Currently we apply the heuristic of pushing down the threshold operator along the query plan as much as possible. For future work, we plan to design a cost estimation model for threshold queries on uncertain data, and integrate it into the query optimizer in PostgreSQL for full automation of our threshold query optimization.

# 5. EXPERIMENTAL EVALUATION

We use our PTQ optimization rules in the Orion system [23] (implemented within PostgreSQL). The goal of our experiments is to validate the effectiveness of our optimization rules proposed in Section 3 on both synthetic and real data sets.

## 5.1 Data Sets

We use two data sets in our experiment: One is a real data set for attributes with continuous uncertainty, and the other is a synthetic data set for attributes with discrete uncertainty.

The real data set comes from a sensor application that monitors the movement of people within a building using 802.11-based sensors that report approximate locations in real-time. Each user is tagged with sensors that report their readings to a central database. Each tuple consists of a sensor ID (SID) that identifies the sensor, the time stamp (TS) of the measurement, and the measured location $(xpos, ypos)$. Due to the calibrated errors with the sensors, the positions are reported with uncertainty represented as Gaussian distributions around the reported locations. We use it as an example of continuous uncertain attributes. Table 3 shows the first 5 tuples in this sensor data set. The notation $prod(Gaus(\mu_1, \sigma_1^2), Gaus(\mu_2, \sigma_2^2))$ stands for the joint *pdf* of the Gaussian distributions of $xpos$ and $ypos$ (as introduced in [23]), where $\mu_1$ ($\mu_2$) and $\sigma_1^2$ ($\sigma_2^2$) are the mean and the variance of $xpos$ ($ypos$). The cumulative probability of the joint *pdf* is 1, hence the tuple probability is 1 for all tuples.

The synthetic data set that we generate is a simulation of the real sensor data set with $xpos$ and $ypos$, having discrete uncertainty. We generate 100,000 tuples in total. Each tuple has a TupleID, along with $xpos$ and $ypos$ values that are jointly distributed as one dependency set, as shown in Table 4. The number of instances in the dependency set, $k$, is uniformly distributed between 1 and 10. Unlike the real data set, the tuple probability of the synthetic data set (equals the total probability of the dependency set) is randomly generated from 0.001 to 1. The probabilities of the instances are generated randomly and sum up to this total probability. The values

of both attributes $xpos$ and $ypos$ are in the range [1, 1000]. For each uncertain attribute in each tuple, we randomly pick a central point `center` in [1, 1000]. We also generate the `spread` of its instance values in the tuple that obeys a Gaussian distribution with mean 10 and variance 2, which roughly corresponds to 1% of the entire range. With the center and spread fixed, we can randomly generate the values of the $k$ instances such that they are within the range [`center - spread` /2, `center + spread` /2]. Unless specified otherwise, the default value of the threshold is 0.4 for all experiments, and the default size of the real and synthetic data sets are 10,000 and 100,000 tuples each.

## 5.2 Query Examples

Below we describe the PTQ queries used in our experiment to test the performance of our optimization rules. We denote the table as $T$, and uncertain attributes as $U$ and $U'$. The value of an uncertain attribute is denoted as $u$ or $u'$. We compare our optimizations against the unoptimized evaluations of the queries. Since probabilistic query evaluation involves using non-standard relational operators (viz. *floor, product, marginalize*), the optimization available in standard PostgreSQL cannot optimize these operations or the threshold operator. Thus the base naïve case that we compare our optimizations to executes the query using Orion operators and then applies the threshold operator to all resulting tuples, retaining only those that meet the overall probability threshold.

The list of queries used in our experiments are given below.

```
Q1: SELECT * FROM T
```

This simple query illustrates the power of Optimization Rule 1. The result should only return those tuples with tuple probability greater than the threshold. To make use of the rule, a B-tree index is created on tuple probabilities and used to prune out all tuples with probabilities below the threshold $\theta$.

```
Q2: SELECT * FROM T WHERE U > u
```

This query benefits from Rule 3 in addition to Rule 1. In order to use Rule 3, it is necessary to support threshold range queries using a PTI index, which is built on attribute $U$ to prune out all tuples with $Pr(U > u) < \theta$. A B-tree index on the original tuple probabilities is also maintained as above for Rule 1 to be applicable.

```
Q3: SELECT * FROM T WHERE U > u AND U' < u'
```

This query benefits from Optimization Rule 4. We build PTI indices on attributes $U$ and $U'$, separately, to prune out tuples with either $Pr(U > u) < \theta$ or $Pr(U' < u') < \theta$.

```
Q4: SELECT * FROM T WHERE U > u OR U' < u'
```

This query demonstrates the effectiveness of Optimization Rule 6. By pruning tuples whose attribute set $U \cup U'$ has a probability below the threshold.

```
Q5: SELECT U FROM T
```

This query benefits from both Rule 1 and Rule 7. Projection does not affect the tuple probabilities. Hence a B-tree index on tuple probabilities is enough for pruning unqualified tuples.

```
Q6: SELECT * FROM T1 INNER JOIN T2
    ON T1.TupleID = T2.TupleID
```

A B-tree index on the tuple probabilities of $T_1$ and another for $T_2$ would suffice for leveraging Optimization Rule 8 to reduce the number of join evaluations that are needed for the inner join query.

```
Q7: SELECT TT1.U FROM (
     (SELECT * FROM T1 WHERE T1.U > u AS TT1)
     INNER JOIN
     (SELECT * FROM T2
      WHERE T2.U > u AND T2.U' < u' AS TT2)
     ON TT1.TupleID = TT2.TupleID)
```

This is an example of a complicated query similar to our example query in Figure 1. It uses several optimization rules: Optimization Rules 1, 3, 4, 7 and 8. These rules work together to ensure that the threshold operator is pushed down the query plan tree as far as possible so that unqualified tuples from either table $T_1$ or $T_2$ can be pruned away before the join and unqualified tuples from the joined table can also be discarded promptly.

## 5.3 Experimental Results

Our experiments compare the optimized PTQ execution with the base case, i.e., the naïve approach that does not use any optimization rules that we proposed earlier. We call them "optim" and "naïve" respectively. We now show how our optimization rules are actually written in the form of SQL queries. Consider query Q7. If the threshold is $p$, this query in the naïve form is written as:

```
SELECT TT1.U FROM (
  (SELECT * FROM T1 WHERE T1.U > u AS TT1)
  INNER JOIN (SELECT * FROM T2 WHERE T2.U > u
              AND T2.U' < u' AS TT2)
  ON TT1.TupleID = TT2.TupleID)
WHERE mass(TT1.U) >= p
```

In the optimized version, this query is written like this:

```
SELECT TT1.U FROM (
  (SELECT TupleID, floor(U, U <= u), U' FROM T1
    WHERE prob > p AND T1.U >? (u, p) AS TT1)
  INNER JOIN
  (SELECT TupleID, floor(U, U <= u),
          floor(U', U' >= u') FROM T2
   WHERE prob > p AND T2.U >? (u, p)
   AND T2.U' <? (u', p) AS TT2)
  ON TT1.TupleID = TT2.TupleID)
WHERE mass(TT1.U) >= p
```

">? (x,p)" and "<? (x,p)" are operators defined in Orion which use the PTI index for value $x$ and threshold $p$. $mass$ is a function that calculates the probability mass of an uncertain variable. The function $floor$ zeroes out part of the uncertain attribute's pdf that does not satisfy the predicate (Section 2.2). We see that $TT_1$ in the optim query is defined using Rules 1 and 3, $TT_2$ is defined using Rules 1, 3 and 4, the join is done with Rule 8 and the projection uses Rule 7. We evaluate all queries from Q1 through Q7 on both real and synthetic data sets in the following aspects:

### 5.3.1 Effect of Data Set Size

Figure 4 and Figure 5 show the effect of data set size on the run time of selection query Q1 and Q3. The threshold is fixed at 0.4. Due to the small size of the real sensor data set we have, we choose to perform this test on the synthetic data set alone. Let the synthetic data set we generated be $T$. We vary the data set size by selecting the desired number of tuples from $T$. We can see in Figure 4 that the time it takes for the naïve approach to execute the query is approximately twice as long as that with optimizations. As the data size increases, the time cost of both the naïve approach and the optimization approach increases steadily, and the optimized query runs consistently faster than the original query. The same holds for Q3, as shown in Figure 5.

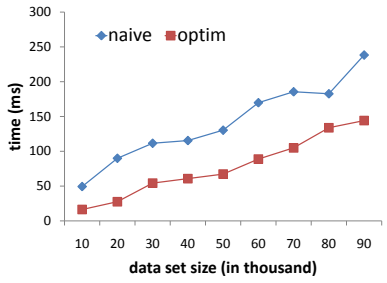


**Figure 4: Effect of data size on Q1**

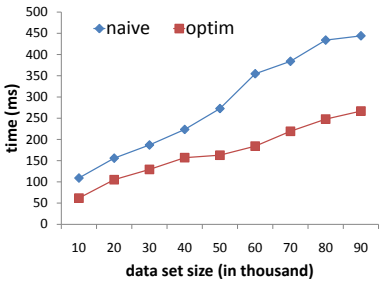


**Figure 5: Effect of data size on Q3**

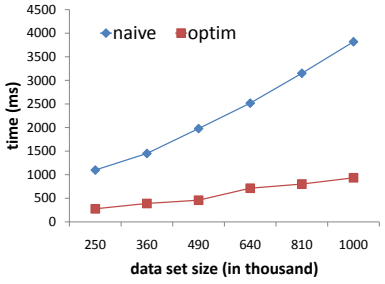


**Figure 6: Effect of data size on Q6**

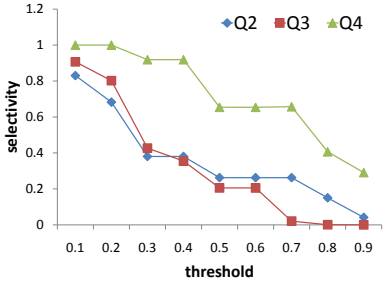


**Figure 7: Query selectivity of Q2, Q3, Q4**

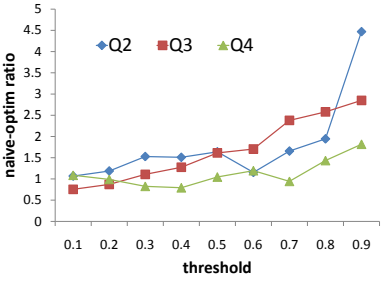


**Figure 8: Naive-optim ratio for Q2-4**

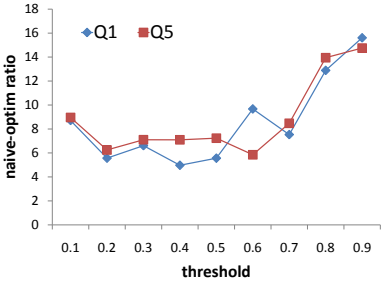


**Figure 9: Naive-optim ratio for Q1, Q5**

For the join query Q6, we generate two tables $T_1$ and $T_2$ from $T$. $T_1$ contains all tuples from $T$ whose $xpos$ is greater than 300 while $T_2$ contains all tuples from $T$ whose $ypos$ is less than 600. From $T_1$ and $T_2$, we obtain two tables by selecting the desired number of tuples from $T_1$ and $T_2$ (the number varies from 500 each to 1000 each) as the two tables for the join operation. We compare the times for running optim and naïve, where optim utilizes B-trees on the tuple probabilities of both tables (i.e., optim implements our Optimization Rule 8). In contrast, naïve does not apply the optimization rule, and always needs to check every pair of tuples from the two tables for the join operation. The result in Figure 6 (the data set size here is the size of the table after the join) shows that Optimization Rule 8 significantly contributes to better performance of Q6 in terms of run time. The difference between the run time of naïve and that of optim increases as data size increases. Overall, the time increase of naïve is much more significant than that of optim.

### 5.3.2 *Effect of Threshold*

The threshold of a PTQ plays an important part in answering the query: Different thresholds result in different sizes of the PTQ result set – the larger the threshold $\theta$, the smaller the set. This is because a PTQ returns all tuples in a table that satisfy the query with probability at least $\theta$. Figure 7 gives a comparison between the selectivity of different thresholds for Q2, Q3 and Q4 on the sensor data set. All three queries are selections with predicates. We define selectivity as the ratio of the size of the PTQ result set and the size of the original data set. With an increasing threshold, all queries observe a consistent decrease in the selectivity of query results, as more tuples become unqualified for the threshold.

Moreover, the threshold also affects the run time of the query, as shown in Figure 8 on the sensor data set for the same queries. We compute the ratio of naïve's run time and our optim's run time for thresholds from 0.1 to 0.9. Obviously, the higher the ratio is, the better our PTQ optimizations are. Figure 8 shows that as the threshold increases, the run time ratio also increases in general and reaches its peak when the threshold is 0.9. Below we refer to this ratio as the "naïve-optim ratio".

We perform the same experiment for simple selection and projection (i.e., Q1 and Q5) on the sensor data set, and the result is shown in Figure 9. Compared with Figure 8, Figure 9 has a similar increasing trend of the naïve-optim ratio for both Q1 and Q5. However, the actual value of the ratio is much higher in Figure 9 than in Figure 8. This is because for Q1 and Q5 that use Optimization Rule 1 and Rule 7 respectively, no PTI index is needed. Only a B-tree on the tuple probability is used for the optimization. Since querying a PTI is more complex than querying a B-tree (see [8] for PTI details), the time involved in executing a threshold query with PTI is much longer, i.e., the run time of optim for Q2, Q3 and Q4 is much longer. This is verified in Figure 10 along with the corresponding run time of naïve (threshold fixed at 0.6). We can see that naïve takes significantly longer times to execute Q1 and Q5 than optim, while the difference is not as big for Q2 through Q4, resulting in higher naïve-optim ratios in Figure 9 than those in Figure 8. In fact, the ratio is always well above 1 for Q1 and Q5 regardless of the threshold (Figure 9) while the ratio is sometimes below 1 for Q2, Q3 and Q4 when the threshold is small (Figure 8). The reason for optim to run slower than naïve in the latter case is the overhead incurred by querying PTI exceeds the benefit from using PTI to prune tuples - when the threshold is small, it is less likely for the PTI to prune large number of tuples away.

Finally, for joins we plot the run time of optim and naïve on the real and synthetic data sets in Figure 11 and Figure 12 respectively. For both sets, the advantage of optim is apparent. While the threshold has little effect on the run time of naïve, it has dramatic impact on the run time of optim: the bigger the threshold is, the faster the query runs. This is due to the fact that the run time of optim heavily depends on thresholds: with a high threshold, Optimization Rule 8 is able to prune a large number of tuples away from both sides, pre-

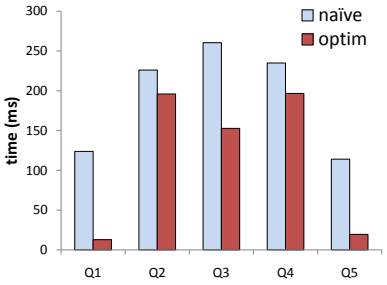


**Figure 10: Run time of naïve and optim**

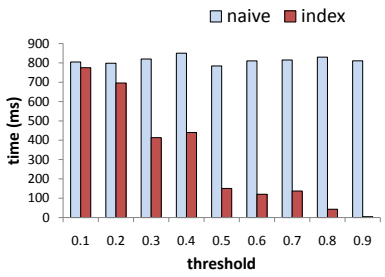


**Figure 11: For Q6 on sensor data**

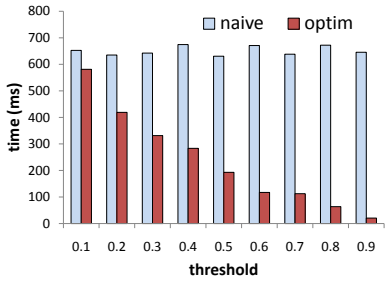


**Figure 12: For Q6 on synthetic data**

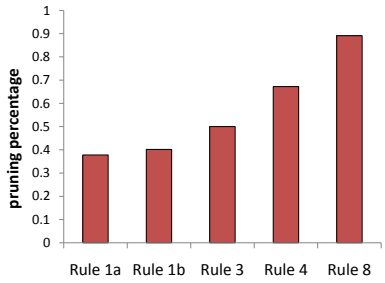


**Figure 13: Effect of pruning by optimization rules**

venting large numbers of tuples from further evaluation. This saves significant time in join processing. On the contrary, naïve always needs to fully evaluate every pair of join tuples regardless of the threshold.

### 5.3.3 *Effect of Optimization Rules*

Having evaluated basic SQL queries for selection, projection and join (Q1 through Q6), we now evaluate a more complicated SQL query that combines all the above operations in one query and leverages multiple optimization rules for faster query execution.

We use Q7 as an example of such queries. The goal of our experiment on Q7 is to verify the effectiveness of different optimization rules in pruning away tuples so that they will not be further evaluated. As we know from Section 5.2, Q7 benefits from Optimization Rules 1, 3, 4, 7 and 8. In Figure 13, we show the pruning percentage (number of join tuples pruned over the total number of join tuples) by applying Rules 1, 3, 4 and 8 separately. Note that here, we do not measure the pruning percentage for Rule 7, since Rule 7 is equivalent to Rule 1 in terms of pruning (projections do not change the tuple probability). While the first three rules are measured individually, Rule 8 is actually a combination of all these rules, since we need them to optimize for $T_1$ and $T_2$ before joining them together. Also, for Rule 1, we measure the pruning percentage for both $T_1$ and $T_2$. To distinguish the two, we call the first Rule 1a and the second Rule 1b. As we can see from Figure 13, Rule 8 has the most powerful pruning capability, discarding almost 90% of all tuples due to its high selectivity from both $T_1$ and $T_2$.

## 6. RELATED WORK

Much research has been done in designing uncertain databases [2, 5, 23, 28, 14, 22, 18]. A key challenge is to ensure the correctness of query processing given dependencies between uncertain data that are either inherent in data or arise during query evaluation (e.g., as a result of joins). For example, MystiQ [5] limits queries evaluation to *safe plans* which are guaranteed to generate correct results (with respect to PWS) [10]. However, safe plans are not always the most efficient plans and may not even exist for certain queries. Trio [2, 21] uses *lineage* to explicitly capture data correlations and efficiently compute confidence. Their query evaluation is not restricted to safe plans and is separated from confidence computation. Other tools to capture dependencies are also proposed, such as factor tables [22], world tables [14] and history [23] (proposed in the Orion model [23]). Unlike MystiQ [5], Trio [2] or MayBMS [14], Orion currently does not support duplicate elimination (partly due to the difficulty of defining duplicates with continuous uncertainty), i.e., there are no disjunction dependencies between tuples. History is updated with each query operation and the confidence of a tuple is computed at the end of the query according to history, similar to the lineage approach [21]. History also ensures that the confidence computed at the end is always correct (without eliminating duplicates), hence we do not need to identify a "safe plan" first from the query before executing the plan. The optimization rules that we propose for threshold queries in this paper aim to avoid unnecessary confidence computations if we already know during the query evaluation that the final confidences of certain tuples will not meet the given threshold. Our optimization rules are designed for the current Orion model without duplicate elimination. In fact, the threshold may no longer be able to "trickle" through the query plan tree with correctness guarantees if duplicate elimination is supported.

While many uncertain data models assume tuple independence or tuple-level dependencies, Orion [23] and MayBMS [14] are able to capture uncertainty at attribute level. In particular, Orion handles dependencies between arbitrary attribute sets and is the first model to handle continuous uncertainty as seen in sensor networks.

Apart from modeling uncertain data, much work focuses on solving specific problems for uncertain data, such as the nearest-neighbor problem [3, 4], indexing [8, 1, 15], ranking [24, 13, 16, 12, 9], range queries [27, 26, 25], skyline queries [17, 20, 29], join processing [7], etc. Among these, the probabilistic threshold query is one of the most common queries over probabilistic data, which returns results satisfying the query with probabilities that meet the threshold. Optimizations can be employed to leverage the threshold for pruning during query evaluation so that all results that have no hope of meeting the threshold can be discarded as early as possible. For example, [6] proposed the concept of the "constrained probabilistic nearest-neighbor query" with a probability threshold and an error tolerance to save expensive computations of the exact nearest-neighbor probabilities. Other examples of threshold queries include the probabilistic threshold approach to ranking queries [13], range queries [8] and skyline queries [20, 29].

While threshold queries have been studied in various settings for probabilistic data, no previous work has focused on optimizing threshold queries for basic database operations, such as selections, projections and joins. Our work falls in this category, and is based on the uncertain data model proposed in [23] which supports uncertain attributes with probability density functions (pdfs). The closest work to ours is [18, 19], which aims at optimizing conjunctive queries without self-joins on data with discrete probabilistic distributions. In our model, both discrete and continuous uncertainty exists, i.e., uncertain attributes have pdfs that are either discrete or continuous. In addition, we use dependency sets to capture inherent dependencies within uncertain data and history to capture dependencies generated from query operations. We propose various optimization rules for efficient execution of probabilistic threshold queries under this model.

# 7. CONCLUSION AND FUTURE WORK

In this paper, we identify query equivalences for probabilistic threshold queries (PTQ) and establish the correctness of pushing down the threshold operator in a query plan. To the best of our knowledge, this is the first attempt to solve the optimization problem of general SQL-based PTQ queries that involve selections, projections and joins. Our PTQ optimization works for the Orion uncertain database model that supports both attribute and tuple uncertainty as well as dependencies between arbitrary attribute sets. The optimization rules are shown to be effective in reducing query processing time through experiments on both real and synthetic data sets. For future work, we plan to estimate the cost of threshold query optimization for full automation within the PostgreSQL query optimizer as well as to support duplicate elimination for projections and the set operations under Orion.

## Acknowledgements

We thank Dr. Amol Deshpande for proposing the problem of general threshold query optimization for uncertain data. The work in this paper is supported by National Science Foundation Grant IIS-0534702 and Grant IIS-0916874.

# 8. REFERENCES

[1] P. K. Agarwal, S. W. Cheng, Y. Tao, and K. Yi. Indexing uncertain data. In *PODS*, 2009.

[2] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. Uldbs: databases with uncertainty and lineage. In *VLDB*, 2006.

[3] G. Beskales, M. A. Soliman, and I. F. Ilyas. Efficient search for the top-k probable nearest neighbors in uncertain databases. In *VLDB*, 2008.

[4] C. Bohm, A. Pryakhin, and M. Schubert. The gauss-tree: Efficient object identification in databases of probabilistic feature vectors. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.

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

[6] R. Cheng, J. Chen, M. Mokbel, and C. Y. Chow. Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In *ICDE*, 2008.

[7] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *CIKM*, 2006.

[8] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, 2004.

[9] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *ICDE*, 2009.

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

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

[12] T. Ge, S. Zdonik, and S. Madden. Top-k queries on uncertain data: On score distribution and typical answers. In *SIGMOD*, 2009.

[13] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *SIGMOD*, 2008.

[14] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: A probabilistic database management system (demonstration). In *SIGMOD*, 2009.

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

[16] F. Li, K. Yi, and J. Jestes. Ranking distributed probabilistic data. In *SIGMOD*, 2009.

[17] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, 2008.

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

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

[20] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, 2007.

[21] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2008.

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

[23] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *ICDE*, 2008.

[24] M. A. Soliman, I. F. Ilyas, and K. C. Chang. Urank: formulation and efficient evaluation of top-k queries in uncertain databases. In *SIGMOD*, 2007.

[25] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB*, 2005.

[26] Y. Tao, X. Xiao, and R. Cheng. Range search on multidimensional uncertain data. *ACM Trans. Database Syst.*, 32(3), 2007.

[27] G. Tracevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, 2004.

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

[29] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *ICDE*, 2009.