Top-k Query Processing in Uncertain Databases

Mohamed A. Soliman Ihab F. Ilyas
School of Computer Science
University of Waterloo
{m2ali,ilyas}@uwaterloo.ca

Kevin Chen-Chuan Chang
Department of Computer Science
University of Illinois at UrbanaChampaign
kcchang@cs.uiuc.edu

Abstract

Top-k processing in uncertain databases is semantically and computationally different from traditional top-k processing. The interplay between score and uncertainty makes traditional techniques inapplicable. We introduce new probabilistic formulations for top-k queries. Our formulations are based on "marriage" of traditional top-k semantics and possible worlds semantics. In the light of these formulations, we construct a framework that encapsulates a state space model and efficient query processing techniques to tackle the challenges of uncertain data settings. We prove that our techniques are optimal in terms of the number of accessed tuples and materialized search states. Our experiments show the efficiency of our techniques under different data distributions with orders of magnitude improvement over naïve materialization of possible worlds.

1 Introduction

Efficient processing of uncertain (probabilistic) data is a crucial requirement in different domains including sensor networks [21, 8], moving objects tracking [7, 9] and data cleaning [12, 3]. Several probabilistic data models have been proposed, e.g., [10, 11, 4, 15, 16, 20, 2], to capture data uncertainty at different levels. According to most of these models, tuples have membership probability, e.g., based on data source reliability [13], or fuzzy query predicates, as addressed in [18]. Tuple attributes could also contain multiple values drawn from discrete or continuous domains [20, 6], e.g., a set of possible customer names in a dirty database, or an interval of possible sensor readings.

Many uncertain data models, e.g., [15, 2, 20], adopt *possible worlds* semantics, where an uncertain relation is viewed as a set of possible instances (worlds). The structure of these worlds could be governed by underlying *generation rules*, e.g., mutual exclusion of tuples that represent the same real-world entity. Such rules could arise naturally with unclean data [13, 3], or could be customized to enforce application requirements, reflect domain semantics, or maintain lineage and data inter-dependencies [5, 20]. The following is a running example for an uncertain database that we use in this paper.

	Time	Radar Loc	Car Model	Plate No	Speed	Conf
t1	11:45	L1	Honda	X-123	130	0.4
t2	11:50	L2	Toyota	Y-245	120	0.7
t3	11:35	L3	Toyota	Y-245	80	0.3
t4	12:10	L4	Mazda	W-541	90	0.4
t5	12:25	L5	Mazda	W-541	110	0.6
t6	12:15	L6	Chevy	L-105	105	1.0

Rules : $(t2 \oplus t3), (t4 \oplus t5)$

(a)

World	Prob.
PW^1={t1,t2,t6,t4}	0.112
PW^2={t1,t2,t5,t6}	0.168
PW^3={t1,t6,t4,t3}	0.048
PW^4={t1,t5,t6,t3}	0.072
PW^5={t2,t6,t4}	0.168
PW^6={t2,t5,t6}	0.252
PW^7={t6,t4,t3}	0.072
PW^8={t5,t6,t3}	0.108

(b)

Figure 1. Uncertain Database and Possible Worlds Space

Example 1 *Consider a radar-controlled traffic, where car speed readings are stored in a database. Radar units detect speed automatically, while car identification ,e.g., by plate number, is usually performed by a human operator. In this database, multiple sources of errors (uncertainty) exist. For example, radar readings can be interfered by high voltage lines, close by cars cannot be precisely distinguished, or human operators might make identification mistakes. Figure 1(a) is a snapshot of a radar database in the last hour. Each reading is associated with a confidence field "conf" indicating its membership probability. Based on radar locations, the same car cannot be detected by radars at two different locations within 1 hour interval. This constraint is captured by the indicated exclusiveness rules.* □

1.1 Uncertain Data Model

The uncertain data model we adopt is based on possible worlds semantics with two pillars. The first pillar is *membership uncertainty*, where each tuple belongs to the database with a probability, hereafter called `confidence`. The second pillar is *generation rules*, which are arbitrary logical formulas that determine valid worlds. Tuples that are correlated with no rules are called *independent*.

Each possible world is a combination of tuples. The probability of each world is computed by assuming the existence of all tuples in the world, and the absence of all other database tuples. The outcome of this computation is determined based on tuple `confidence` values and *generation rules*. For example in Figure 1(b), the probability of the possible world PW^5 is computed by assuming the existence of $t2$, $t6$, and $t4$, and the absence of $t1$, $t3$, and $t5$. Note that the existence of $t2$ implies the absence of $t3$ (and the same for $t4$ and $t5$) based on the indicated exclusiveness rules, while $t1$ and $t6$ are *independent* of all other tuples. Therefore, $Pr(PW^5) = 0.7 \times 1.0 \times 0.4 \times 0.6 = 0.168$, where 0.7, 1.0, 0.4 are the existence probabilities of $t2, t6$, and $t4$, respectively, while 0.6 is the absence probability of $t1$.

1.2 Motivation and Challenges

Current query processing techniques for uncertain data [6, 7, 12, 3] focus on Boolean queries. However, uncertainty usually arises in data exploration, decision making, and data cleaning scenarios which all involve aggregation and ranking. Top–k queries are dominant type of queries in such applications. A traditional top–k query returns the k objects with the maximum scores based on some scoring function. In the uncertain world, such a clean definition does not exist. Reporting a tuple in a top–k answer does not depend only on its score, but also on its membership probability, and the scores and membership probabilities of other tuples. Tuple scores and uncertainty information interplay to decide the top–k answers. Consider Example 1. Two interesting top–k queries are to report:

- The top–k speeding cars in the last hour.
- A ranking over the models of the top–k speeding cars.

While the above queries are semantically clear in deterministic databases, their interpretation in uncertain databases is challenging. For example, it might be desirable that the answer to the first query be a set of cars that appear *together* in valid possible world(s), to avoid answers inconsistent with generation rules and other database constraints. While in the second query, we might not have this restriction. In uncertain databases, we are usually after the *most probable* query answers. The interaction between the concepts of "most probable" and "top–k" gives rise to different possible interpretations of *uncertain top–k queries*:

- The "top–k" tuples in the "most probable" world.
- The "most probable top–k" tuples that belong to valid possible world(s).
- The set of "most probable top–i^{th}" tuples across all possible worlds, where $i = 1 \dots k$.

While the first interpretation reduces to a top–k query on a deterministic database, the second and third interpretations are challenging since they involve both ranking and aggregation across worlds. We formally define and elaborate on the differences between these queries in Section 2.

A naïve approach to obtain answers to the above queries is to materialize the whole possible worlds space, find top–k answer in each world, and aggregate the probabilities of identical answers. Flattening the database into all its worlds is prohibitively expensive because of the huge number of possible worlds and the potential complexity of generation rules. Processing the "compact" database, i.e. without materializing its world space, is the main focus of this paper.

1.3 Contributions

Our approach in this paper is to process score and uncertainty in one framework leveraging current DBMS storage and query processing capabilities. Our contributions are summarized as follows:

- *New Query Definitions:* "Top-k processing in uncertain relational databases" is to the best of our knowledge a previously unstudied problem with unclear semantics. We propose new formulations for top–k queries in uncertain databases.
- *Search Space Model:* We model uncertain top–k processing as a state space search problem, and introduce several space navigation algorithms with optimality guarantees on the number of accessed tuples and materialized search states.
- *Processing Framework:* We construct a framework integrating space navigation algorithms and data access methods leveraging existing DBMS technologies.
- *Experimental study:* We conduct an extensive experimental study to evaluate our techniques under different data distributions.

2 Problem Definition

Based on the uncertain data model described in Section 1.1, and assuming some scoring (ranking) function to order tuples, the probability of a k-length tuple vector T to be the top–k is the summation of possible worlds probabilities where T is the top–k. Similarly, the probability of a tuple t to be at rank i is the summation of possible worlds probabilities where t is at rank i. We now formally define uncertain top–k queries based on "marriage" of possible worlds and traditional top–k semantics.

Definition 1 ***Uncertain Top–k Query (U-Top$k$):*** *Let $\mathcal{D}$ be an uncertain database with possible worlds space $\mathcal{PW} = \{PW^1, \dots, PW^n\}$. Let $\mathcal{T} = \{T^1, \dots, T^m\}$ be a set of k-length tuple vectors, where for each $T^i \in$*

$\mathcal{T}$: (1) Tuples of T^i are ordered according to scoring function $\mathcal{F}$, and (2) T^i is the top–k answer for a non empty set of possible worlds $PW(T^i) \subseteq \mathcal{PW}$. A U-Topk query, based on $\mathcal{F}$, returns $T^* \in \mathcal{T}$, where $T^* = argmax_{T^i \in \mathcal{T}}(\sum_{w \in PW(T^i)}(Pr(w)))$. □

U-Top$k$ query answer is a tuple vector with the maximum aggregated probability of being top–k across all possible worlds. This type of queries fits in scenarios where we need to restrict all top–k tuples to belong together to the same world(s), e.g., for compliance requirements with model rules. Consider Figure 1 again. U-Top2 query answer is $\{t1, t2\}$ with probability 0.28, which is the summation of PW^1 and PW^2 probabilities.

Definition 2 ***Uncertain k Ranks Query (U-kRanks):*** *Let $\mathcal{D}$ be an uncertain database with possible worlds space $\mathcal{PW} = \{PW^1, \ldots, PW^n\}$. For $i = 1 \ldots k$, let $\{x_i^1, \ldots, x_i^m\}$ be a set of tuples, where each tuple x_i^j appears at rank i in a non empty set of possible worlds $PW(x_i^j) \subseteq \mathcal{PW}$ based on scoring function $\mathcal{F}$. A U-kRanks query, based on $\mathcal{F}$, returns $\{x_i^*; i = 1 \ldots k\}$, where $x_i^* = argmax_{x_i^j}(\sum_{w \in PW(x_i^j)}(Pr(w)))$.* □

U-kRanks query answer is a set of tuples that might not form together the most probable top–k vector. However, each tuple is a clear winner at its rank over all worlds. This type of queries fits in data exploration scenarios, where the most probable tuples at top ranks are required without restricting them to belong to the same world(s). Consider Figure 1 again. U-2Ranks query answer is $\{t2 : 0.42, t6 : 0.324\}$, since $t2$ is at rank 1 in PW^5 and PW^6 with aggregated probability 0.42, while $t6$ is at rank 2 in PW^3, PW^5, and PW^8 with aggregated probability 0.324.

For clarity, we focus only on the "most probable" top–k answers. However, the above two definitions could be extended to represent answers as probability distribution, i.e., a set of top–k answers associated with their probabilities, rather than single answers. This is explained in more details in our space navigation algorithms in Section 4.

3 Processing Framework

Since uncertain data is likely to be stored in a traditional database, most of current uncertain database system prototypes rely on relational DBMSs for efficient retrieval and query processing, e.g., Trio [5], uses an underlying DBMS to store and process uncertain data and lineage information.

In this section, we propose a novel processing framework (Figure 2) that leverages RDBMS storage, indexing, and query processing techniques to compute uncertain top–k query answers. Our proposed processing framework consists of two main layers:

- *Storage Layer:* Tuple retrieval, indexing and traditional query processing (including score-based ranking) are the main functionalities provided by the storage layer. Uncertain data and generation rules are stored in a relational database and different data access methods are provided to allow the upper processing layer to retrieve uncertain tuples.
- *Processing Layer:* The processing layer retrieves uncertain tuples from the underlying storage layer, and efficiently navigates the partial space of possible worlds to compute the most probable top–k answers.

Problem Space. Before describing the processing layer details, we formulate our problem as searching the space of states that represents all possible top–k answers. Definition 3 gives a formal definition of the search state.

Definition 3 ***Top-l State:*** *A top–l state s_l is a tuple vector of* length l *that appears as the top–l answer in one or more possible worlds based on scoring function $\mathcal{F}$.* □

A top–l state s_l is *complete* if $l = k$. *Complete* states represent possible top–k answers. The *probability of state* s_l is the summation of the probabilities of the possible worlds where s_l is the top–l answer. Our search for uncertain top–k query answers starts from an empty state (with *length* 0) and ends at a goal state that is a *complete* state with a probability greater than any other state.

The main components of the processing layer are depicted in Figure 2. The *Rule Engine* is responsible for computing the state probabilities. The *Rule Engine* can be an intelligent business process or a sophisticated model, e.g., Bayesian network [10]. The details of rule engine models are out of the scope of this paper. The formulation of states is performed by the *State Formulation* module. The *Space Navigation* module uses navigation algorithms to partially materialize the possible worlds.

Iterator Interface. We assume an *iterator* interface, widely used in RDBMS's, to retrieve tuples from *storage layer* sequentially and pass them to the *processing layer*. We further prove that *sorted score access* is the optimal sequential access method in the number of retrieved tuples to answer uncertain top–k queries. Efficient rank-aware query processing technique [14, 17] could be used in *storage layer* to report tuples in score order.

Theorem 1 *Among all sequential access methods, sorted score access is optimal in the number of retrieved tuples to answer uncertain top–k queries.* □

Proof: Assume an algorithm $\mathcal{A}$ that retrieves tuples sequentially out of score order. $\mathcal{A}$ cannot decide whether a seen tuple t belongs to any possible top–k answer or not. This is because there could be k unseen tuples independent with t, all having higher scores than t, and all having a `confidence` value of 1, which means that t does not belong to any possible top–k answer. $\mathcal{A}$ cannot assert this fact

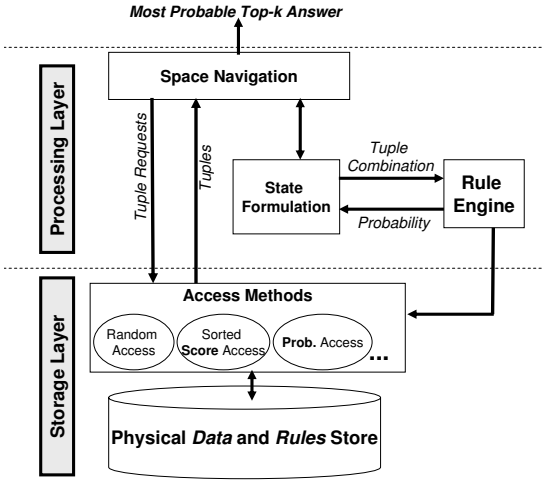

Figure 2. Processing Framework

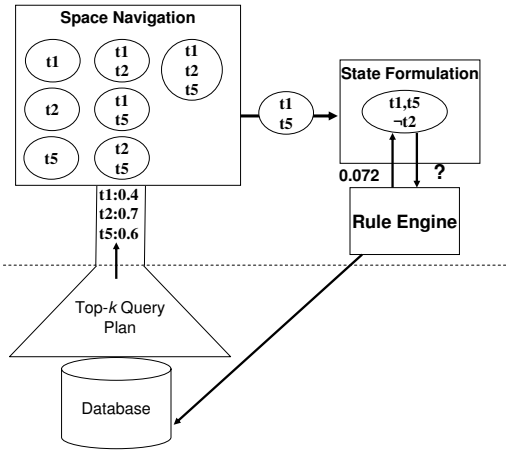

Figure 3. Components Interaction

unless it sees all database tuples. Then, $\mathcal{A}$ cannot answer uncertain top–k queries while consuming less tuples than a sorted score access method. □

Retrieving tuples in `confidence` is also not optimal. Assume an algorithm $\mathcal{A}$ that retrieves tuples in `confidence` order. In this case $\mathcal{A}$ cannot compute the probability of some seen tuple t to belong to any possible top–k answer. This is because $\mathcal{A}$ cannot guarantee that it has seen all tuples with higher scores than t.

Processing Overview. We give an overview of framework components interaction in Figure 3, which describes the processing of an uncertain top–k query for the database in Example 1. In Figure 3, three tuples are produced by a top–k query plan and submitted to the *Space Navigation* module, which materializes all possible states based on the three *seen* tuples. In order to compute state probability, the *State Formulation* module formulates each state and computes its probability by contacting the *Rule Engine* (details are in Section 4.1). For example, to formulate a state for the tuple vector $< t1, t5 >$, the intermediate tuple $t2$ must be absent. The number of maintained states is exponential in the number of seen tuples. Therefore, our primary *cost metric* is the number of accessed tuples from the *storage layer*. Furthermore, our *Space Navigation* algorithms avoid materializing any *useless* states, i.e., states that do not have a chance to lead to query answer.

In Section 4, we give optimal search algorithms that partially materialize the space of top–k answers by retrieving the least possible number of tuple, and materializing the least possible number of search states.

4 Navigating the Search Space

In this section we describe how to navigate the space to obtain uncertain top–k query answers. We start by showing how to compute state probabilities in Section 4.1. We then describe our proposed U-Top$k$ and U-kRanks query processing algorithms in Sections 4.2 and 4.3, respectively.

4.1 Computing State Probabilities

Each uncertain tuple t is a source of two events: (1) tuple existence, denoted t, with probability t`.confidence`, and (2) tuple absence, denoted $\neg t$, with probability `1 - `t`.confidence`. The probability of any combination of tuple existence/absence events is the summation of possible worlds probabilities where this combination is satisfied. For example in Figure 1, the probability of the combination ($t1$ and $\neg t2$) is the same as $Pr(PW^3) + Pr(PW^4) = 0.12$. We next explain an important property that we exploit while navigating the search space.

Property 1 ***Probability Reduction.*** *Extending a combination of tuple events by adding another tuple existence/absence event results in a new combination with at most the same probability* □

Property 1 is clear from theoretical set operations, where a set can never be larger than its intersection with another set. This property holds in our model since for any two sets of tuple events E_n and E_{n+1} (with lengths n and $n+1$), where $E_n \subset E_{n+1}$, the set of possible worlds satisfying $E_{n+1} \subseteq$ the set of possible worlds satisfying E_n.

In the following we use the notation $(\neg X)$ where X is a tuple set/vector to refer to the conjunction of negation events of tuples in X.

State Probability. Assume an uncertain database $\mathcal{D}$, and an arbitrary scoring function $\mathcal{F}$. After d accesses to $\mathcal{D}$ in $\mathcal{F}$ order, let s_l be some search state, and $I(s_l, d)$ be the current set of retrieved tuples from $\mathcal{D}$ that are not in s_l. The

probability of state s_l, denoted $\mathcal{P}(s_l)$, can be computed as: $\mathcal{P}(s_l) = Pr(s_l \cap \neg I(s_l, d))$

For example in Figure 1, if the current set of retrieved tuples are $\{t1, t2, t5\}$, then for a state $s_2 =< t1, t5 >$, we have $\mathcal{P}(s_2) = Pr(\{t1, t5\} \cap \neg\{t2\}) = 0.072$. This result can be verified from the possible world PW^4.

State Extension. Assume a state s_l. After retrieving a new tuple t from $\mathcal{D}$, we extend s_l into two possible states: (1) a modified version of s_l with the same tuple vector, assuming the event $\neg t$, and (2) a state s_{l+1} with the tuple vector of s_l appended by $\{t\}$, assuming the event t. Notice that the summation of the probabilities of the modified s_l and s_{l+1} is the same as the probability of the old s_l state.

4.2 Processing U-Top$k$ Queries

We now describe `OPTU-Top`k, an optimal algorithm in the numbers of accessed tuples and visited search states to answer a U-Top$k$ query. `OPTU-Top`k buffers the ranked tuples retrieved from $\mathcal{D}$, and adopts a *lazy* materialization scheme to extend the state space. Hence, a state might not be extended by all seen tuples. At each step, the algorithm extends only the state with the *highest probability*. The extension is performed using the next tuple drawn either from the buffer or from $\mathcal{D}$.

We overload the definition of a search state s_l to be $s_{l,i}$, where i is the position of the last seen tuple by $s_{l,i}$ in the score-ranked tuple stream. Note that i can point to a buffered tuple or to the next tuple to be retrieved from $\mathcal{D}$. Furthermore, we define e as an *empty state* of *length* 0. Algorithm `OPTU-Top`k starts by initializing e with state $s_{0,0}$, where $\mathcal{P}(s_{0,0}) = 1$. The empty state e is used to upper-bound the probability of any non-materialized state, since any non-materialized state must be driven from e.

Let Q be a priority queue of states ordered on their probabilities, where ties are broken by state *length*. We initialize Q with e. Let d be the number of seen tuples from $\mathcal{D}$ at any point. Algorithm `OPTU-Top`k iteratively retrieves the top state of Q, say $s_{l,i}$, extends it into the two next possible state (Section 4.1), and inserts the resulting two states back to Q based on their probabilities. Extending $s_{l,i}$ will lead to consuming a new tuple from $\mathcal{D}$ only if $i = d$, otherwise $s_{l,i}$ is extended with the buffered tuple pointed to by $i + 1$.

The termination condition of `OPTU-Top`k is when the top state of Q is a *complete* state. If a *complete* state c is on top of Q, i.e., all materialized and non-materialized states have smaller probabilities than c, then there is no way to generate another *complete* state to beat c, based on Property 1. Algorithm 1 describes the details of `OPTU-Top`k. We next prove the optimality guarantees of `OPTU-Top`k.

Theorem 2 *Among all algorithms that access tuples ordered on score, `OptU-Top`k is optimal in the number of accessed tuples.*□

Algorithm 1 `OptU-Top`k($source$,k)

Require:
  $source$: Score-ranked tuple stream
  k: Answer length
Ensure: U-Top$k$ query answer
1: $Q \leftarrow$ empty priority queue for states ordered on probabilities
2: $e \leftarrow s_{0,0}$ where $\mathcal{P}(e) = 1$ {init empty state}
3: $d \leftarrow 0$ {scan depth in $source$}
4: Insert e into Q
5: **while** ($source$ is not exhausted AND Q is not empty) **do**
6: $s_{l,i} \leftarrow$ dequeue (Q)
7: **if** ($l = k$) **then**
8: **return** $s_{l,i}$
9: **else**
10: **if** ($i = d$) **then**
11: $t \leftarrow$ next tuple from $source$
12: $d \leftarrow d + 1$
13: **else**
14: $t \leftarrow$ tuple at pos $i + 1$ from seen tuples
15: **end if**
16: Extend $s_{l,i}$ using t into $s_{l,i+1}$, $s_{l+1,i+1}$ {Section 4.1}
17: Insert $s_{l,i+1}$, $s_{l+1,i+1}$ into Q
18: **end if**
19: **end while**
20: **return** dequeue(Q)

Proof: `OptU-Top`k consumes score-ranked tuples until no state has a higher probability than the current *complete* state. Assume another algorithm, $\mathcal{A}$, that also consumes tuples in score order but reports U-Top$k$ answer while consuming $d \geq k$ tuples, where d is strictly less than the number of tuples consumed by `OptU-Top`k. Assume x is the *complete* state reported by $\mathcal{A}$. Based the seen d tuples, Algorithm `OptU-Top`k would have a state $s_{l,i}$ where $l < k$ and $\mathcal{P}(s) > \mathcal{P}(x)$, since otherwise `OptU-Top`$k$ would have terminated. Assume that there exist $k - l$ tuples with `confidence` 1, not yet seen by $\mathcal{A}$, and independent of tuples in s. We can augment s with these $k - l$ tuples to create another *complete* state $\grave{s}$, where $\mathcal{P}(\grave{s}) = \mathcal{P}(s) > \mathcal{P}(x)$. Then x, the answer reported by $\mathcal{A}$, is incorrect. □

Theorem 3 *Let x be the reported answer by `OptU-Top`k. Among all algorithms that access tuples ordered on score, there is no algorithm that can skip a state visited by `OptU-Top`k and reports x as the U-Top$k$ query answer.*□

Proof: Assume an algorithm $\mathcal{A}$, that consumes tuples ordered on score and concludes x as the U-Top$k$ answer, where x is also the answer reported by `OptU-Top`k. Let s_l be a state skipped by $\mathcal{A}$, and visited by `OptU-Top`k, where $\mathcal{P}(s) = p$. By definition s_l has the highest probability among all states that can be accessed by `OptU-Top`k and $\mathcal{A}$. Assume there exist $k - l$ tuples independent with s_l, and all with `confidence` 1. The state s_l could therefore be extended to a *complete* state s_k with probability p. Hence, `OptU-Top`k reports $x = s_k$ as its final answer.

Since $\mathcal{A}$ did not visit s_l, the final answer reported by $\mathcal{A}$ cannot be any extension of s_l, which contradicts the original assumption that both algorithms returned the same answer. Moreover, $\mathcal{A}$ cannot report an answer with a higher probability than p. □

Note that under the assumption that all states probabilities are distinct (e.g., by assuming a unified tie-breaker mechanism), no other algorithm can reach a correct solution without visiting all the states visited by `OptU-Top`k. This observation can be easily derived from the above proof since in this case, x is the only correct answer.

Algorithm 1 can be extended to return the n most probable U-Top$k$ answers by keeping a priority queue of size n, and inserting any *complete* state with a probability above the queue probability lower bound. The termination condition will be changed to "the probability of any state is strictly less than the queue probability lower bound".

4.3 Processing U-kRanks Queries

In this section, we describe `OPTU-`k`Ranks`, an optimal algorithm in the number of accessed tuples to answer U-kRanks queries. Algorithm `OPTU-`k`Ranks` extends maintained states based on each seen tuple. When a new tuple is retrieved, it is used to extend all states causing all possible ranks of this tuple to be recognized. Let t be a tuple seen after retrieving d tuples from the score-ranked stream. Let $P_{t,i}$ be the probability that tuple t appears at rank i, based on scoring function $\mathcal{F}$, across all possible worlds. It follows from our state definition that $P_{t,i}$ is the summation of the probabilities of all states with *length* i whose tuple vectors end with t, provided that t is the last retrieved tuple from $\mathcal{D}$. In other words, we can compute $P_{t,i}$, for $i = 1 \dots k$, as soon as we retrieve t from the database.

For each rank i, we need to remember *only* the most probable answer obtained so far, since any unseen tuple u cannot change $P_{t,i}$ of a seen tuple t because u can never appear before t in any possible world. The remaining question is when can we conclude an answer at each rank i? We need to take unseen tuples into account to answer this question. An unseen tuple u will appear at rank i only if we extend states with *length* $i-1$ by appending u to their tuple vectors. Therefore, the summation of the probabilities of states with *length* $i-1$ is an upper-bound over $P_{u,i}$ for any unseen tuple u. We now can state our stopping condition. Let t^i be the current U-kRanks answer for rank i, let S_{i-1} be the set of all maintained states with *length* $i-1$. The termination condition of `OPTU-`k`Ranks` algorithm, at rank i, is $P_{t^i,i} > \sum_{s \in S_{i-1}} \mathcal{P}(s)$. Algorithm 2 formally describes `OPTU-`k`Ranks`. The next Theorem formalizes the optimality of `OPTU-`k`Ranks`.

Theorem 4 *Among all algorithms that access tuples ordered on score,* `OptU-`k`Ranks` *is optimal in number of accessed tuples.*□

Algorithm 2 `OptU-`k`Ranks`($source$,k)

Require:
  $source$: Score-ranked tuple stream
  k: Answer length
Ensure: U-kRanks query answer
1: $answer[\,] \leftarrow$ empty vector of length k
2: $ubounds[\,] \leftarrow$ vector of length k initialized with 1's {prob. upper bound of any unseen tuple at each rank $1 \dots k$}
3: $reported \leftarrow 0$ {No. of reported answers}
4: $depth \leftarrow 1$
5: $space \leftarrow \phi$ {current set of states}
6: **while** ($source$ is not exhausted AND $reported < k$) **do**
7: $t \leftarrow$ next tuple from $source$
8: **for** i=1 to $min(k, depth)$ **do**
9: Extend $space$ states with length $i-1$ using t
10: Compute $P_{t,i}$
11: Update $ubounds[i]$ based on states of length $i-1$
12: **if** ($answer[i]$ was previously reported) **then**
13: continue
14: **end if**
15: **if** ($P_{t,i} > answer[i].prob$) **then**
16: $answer[i] \leftarrow t$
17: $answer[i].prob \leftarrow P_{t,i}$
18: **if** ($answer[i].prob > ubounds[i]$) **then**
19: Report $answer[i]$
20: $reported \leftarrow reported + 1$
21: **end if**
22: **end if**
23: **end for**
24: $depth \leftarrow depth + 1$
25: **end while**

Proof: Assume another algorithm, $\mathcal{A}$, that also consumes tuples ordered on score but reports t^i as the U-kRanks answer for rank i while $P_{t^i,i} < \sum_{s \in S_{i-1}} \mathcal{P}(s)$. Assume an unseen tuple u with `confidence` 1, and u is independent of all seen tuples. The tuple u can be appended to all states in S_{i-1} to create a new set of sates S_i with exactly the same probabilities, where in every $s \in S_i$, u is at rank i. Then, $P_{u,i} = \sum_{s \in S_i} \mathcal{P}(s) = \sum_{s \in S_{i-1}} \mathcal{P}(s) > P_{t^i,i}$. Hence t^i, the answer reported by $\mathcal{A}$, is incorrect. □

5 Cutting Down the Computation

In this section, we introduce other algorithms that make use of tuple independence to cut down the state materialization significantly. Under arbitrary generation rules, the states materialized by `OPTU-Top`k and `OPTU-`k`Ranks` algorithm are generally incomparable even if they have the same *length*. This is because each state could be extended in a different manner to a *complete* state. For example, the tuples in one state might imply all other unseen tuples.

The materialized states by `OPTU-Top`k and `OPTU-`k`Ranks` algorithms could be reduced significantly if we have an ability to prune looser states from our search space early. In general, an incomplete state s can be

pruned if there exists a *complete* state c with $\mathcal{P}(c) > \mathcal{P}(s)$. Hence, if we can compute the maximum probability of a *complete* state generated from s, denoted $p_{max}(s)$, we can safely prune all states with probability less than $p_{max}(s)$. The *Rule Engine* might be able to compute $p_{max}(s)$ of a given state s, however, this operation is sensitive to the complexity of generation rules, and the *Rule Engine* design. Alternatively, we show in the next sections how to make use of tuple independence to do much efficient space pruning while keeping the *optimality* in the number of accessed tuples.

5.1 U-Top$k$ Queries with Tuple Independence

With tuple independence, state space can be aggressively pruned to keep only the states that could lead to the answer. We describe Algorithm `IndepU-Top`k, which prunes the space based on the following criterion.

Lemma 1 ***Comparable States.*** *Under tuple independence, a state x_n is probability-comparable to any state y_m, where x_n and y_m are maintained after seeing the same set of score-ranked tuples, and $n \geq m$.* □

Lemma 1 states that under tuple independence, each state can be compared, based on probability, to other states of the same or smaller length if all states are maintained after seeing the same tuples. This is because for *comparable* states x_n and y_m, the most probable *complete* states derived from each of them would be obtained using the same set of existence/absence events of unseen tuples. That is, x_n and y_m would follow the same path to reach a *complete* state. However, x_n will reach a *complete* state at most at the same time as y_m, since $n \geq m$. Based on the above and according to Property 1, we have guarantees that if $\mathcal{P}(x_n) > \mathcal{P}(y_m)$, the *complete* state derived from x_n would have a higher probability than the one derived from y_m, and therefore we can safely prune y_m from our search space.

`IndepU-Top`k exploits Lemma 1 by grouping states into equivalence classes based on their *lengths*. `IndepU-Top`k keeps *at most one* state for each *length* value $0 \ldots k$ in a candidate set. The candidate set is extended on receiving each new tuple from $\mathcal{D}$. `IndepU-Top`k terminates when at least k tuples have been retrieved, and the probability of any current state is not above the probability of the current *complete* candidate.

Consider for example the score-ranked stream of independent tuples shown in Figure 4 (fractions indicate `confidence`, and scores are omitted for brevity), where we are interested in U-Top3 answer. We represent each state s_l with its tuple vector, and distinguish tuples seen but not included by s_l with the $\neg$ symbol. In step (a), after retrieving the first tuple $t1$, we construct two states $< \neg t1 >$ and $< t1 >$ with *length* values 0 and 1, respectively. In step (b), the candidate set is updated based on the new tuple $t2$, where two possible candidates with *length* 1, $< t1, \neg t2 >$ and $< \neg t1, t2 >$, are generated. However, we keep only the candidate with the highest probability since both candidates are probability comparable.

Step (c) continues in the same manner by updating the candidate set based on tuple $t3$, and pruning the less probable candidate from each equivalence class. Note that the candidate $< \neg t1, \neg t2, \neg t3 >$ is pruned because there is another candidate $< \neg t1, \neg t2, t3 >$ with a larger length and higher probability. In step (c) we have constructed the first *complete* candidate, $< t1, t2, t3 >$, and the first termination condition is met. In step (d) we update the candidate set based on $t4$. Notice that we cannot stop after step (d) because the second termination condition is not met yet – there are candidates with higher probabilities than the current *complete* candidate – and so, there is a chance that $< t1, t2, t3 >$ will be beaten. Space reduction by exploiting state comparability property results in *huge* performance improvements for large values of k. We illustrate the scalability of `IndepU-Top`k in our experimental section.

5.2 U-kRanks Queries with Tuple Independence

When tuples are independent, a U-kRanks query exhibits the *optimal substructure* property, i.e. the optimal solution of the larger problem is constructed from solutions of smaller problems. This allows using a dynamic programming algorithm. We next describe `IndepU-`k`Ranks`, a dynamic programming algorithm with optimality guarantees in the number of accessed tuples.

Consider the example depicted by Figure 5, where we are interested in U-3Ranks query answer. In the shown table, a cell at row i and column x indicates the value of $P_{x,i}$. The value of $P_{x,1}$ is computed as $Pr(x) \times \prod_{z:\mathcal{F}(x)<\mathcal{F}(z)}(1 - Pr(z))$, which is the probability that x exists and all tuples with higher scores do not exist. The values of $P_{x,i}$, where $i > 1$, are computed based on the following property:

Property 2 *For independent tuples and $i > 1$, $P_{x,i} = Pr(x) \times \sum_{y:\mathcal{F}(y)>\mathcal{F}(x)}(\prod_{z:\mathcal{F}(x)<\mathcal{F}(z)<\mathcal{F}(y)}(1 - Pr(z)) \times P_{y,i-1}$.* □

The rationale behind Property 2 is that with independent tuples, for a tuple x to appear at rank i, we need only to consider the probability that x is consecutive to every other tuple y at rank $i-1$. This probability is computed using the probability that x exists, each intermediate tuple z between x and y does not exist, and y appears at rank $i-1$.

For example, in Figure 5, $P_{t2,2} = 0.9 \times 0.3 = 0.27$, while $P_{t3,2} = (0.6 \times 0.63) + (0.6 \times 0.1 \times 0.3) = 0.396$. The shaded cells indicate the U-3Ranks query answers at each rank. Notice that the summation of the probabilities of each row will be 1 if we completely exhaust the tuple

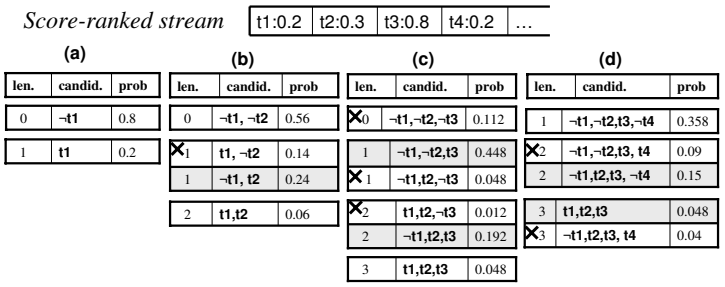

Figure 4. `IndepU-Top`k **Processing**

Figure 5. `IndepU-`k`Ranks` **Processing**

stream. This is because each row actually represents a horizontal *slice* in all the possible worlds. This means that we can report an answer from any row whenever the maximum probability in that row is greater than the row probability remainder. Notice also that the computation in each row depends solely on the row above.

The above description gives rise to the following dynamic programming formulation. We construct a matrix M with k rows, and a new column is added to M whenever we retrieve a new tuple from the score-ranked stream. Upon retrieving a new tuple t, the column of t in M is filled downwards based on the following equation:

$$M[i,t] = \begin{cases} i = 1: \ \ Pr(t) \times \prod_{z:\mathcal{F}(t)<\mathcal{F}(z)}(1 - Pr(z)) \\ \\ i > 1: \ \ Pr(t) \times \sum_{y:\mathcal{F}(y)>\mathcal{F}(t)}((\prod_{z:\mathcal{F}(t)<\mathcal{F}(z)<\mathcal{F}(y)} \\ (1 - Pr(z))) \times M[i-1,y]) \end{cases} \quad (1)$$

For example in Figure 5, $M[2,3] = Pr(t3) \times (M[1,2] + (1 - Pr(t2)) \times M[1,1])$. Algorithm `IndepU-`$k$`Ranks` returns a set of k tuples $\{t_1 \dots t_k\}$, where $t_i = argmax_x \ M[i,x]$.

6 Experiments

We built our framework on top of RankSQL [17]. All experiments were run on a 3GHz Pentium IV PC with 1 GB of main memory, running Debian GNU/Linux3.1. Space navigation algorithms, and a rule engine prototype were implemented in C, and they interact with database through cursor operations. We conducted extensive experiments evaluating the efficiency of our techniques in different settings. We used synthesized datasets of different data distributions generated by the R-statistical computing package [1]. Our primary performance metrics are: (1) query execution time, and (2) tuple scan depth in $\mathcal{D}$. In all experiments we used rank-aware plans as the source of score-ranked tuple stream. We emphasize, however, that our techniques are transparent from the underlying top–k algorithm.

6.1 The Naïve Approach

We illustrate the infeasibility of applying the naïve approach of *materializing* possible worlds space, *sorting* each world individually, and *merging* identical top–k answers. Due to space explosion, we applied this approach to small databases of sizes less than 30 tuples with different sets of generation rules. The *materialization* phase was the bottleneck in this approach consuming, on average, an order of magnitude longer times than the *merging* phase. For example, processing a database of 28 tuples with exclusiveness rules yielded 524,288 possible worlds, and top–k query answer was returned after 1940 seconds of which 1895 seconds were used to materialize the world space.

6.2 Effect of `Confidence` Distribution

We evaluate here the effect of `confidence` distribution on execution time and scan depth. We used datasets with the following (score,`confidence`) distribution pairs: (1)*uu*: score and `confidence` are uniformly distributed, (2)*un (mean x)*: score is uniformly distributed, and `confidence` is normally distributed with mean x, where $x = 0.5, 0.9$, and standard deviation 0.2, and (3)*uexp (x)*: score is uniformly distributed, and `confidence` is exponentially distributed with mean x, where $x = 0.2, 0.5$.

Figures 6 and 7 show the time and scan depth of `IndepU-Top`k, respectively, while Figures 8, 9 show the time and scan depth of `IndepU-`k`Ranks`, respectively with k values up to 1000. The best case for both algorithms is to find highly probable tuples frequently in the score-ranked stream. This allows obtaining strong candidates to prune other candidates aggressively, and thus terminate the search quickly. This scenario applies to *un(mean 0.9)* distribution pair where a considerable number of tuples are highly probable. The counter scenario applies to *uexp(0.2)* whose mean value forces `confidence` to *decay* relatively fast leading to small number of highly probable tuples. `IndepU-Top`k execution time is under 10 seconds for all data distributions, and it consumes a maximum of 15,000 tuples for k=1000 under exponentially-skewed distribution. The maximum scan depth of `IndepU-`k`Ranks` is 4800 tuple, however the execution time is generally larger (a maximum of 2 minutes). This can be attributed to the design of both algorithms where bookkeeping and candidate maintenance operations are more extensive in `IndepU-`k`Ranks`.

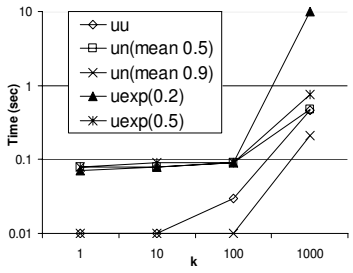

Figure 6. IndepU–Top$k$ time (different distributions)

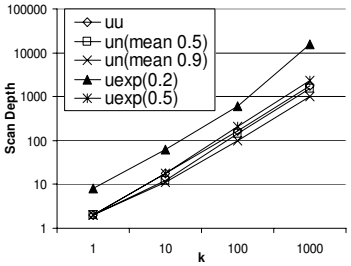

Figure 7. IndepU–Top$k$ depth (different distributions)

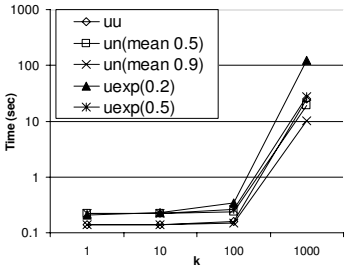

Figure 8. IndepU–kRanks time (different distributions)

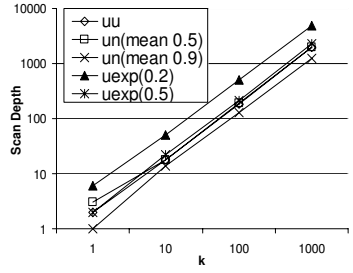

Figure 9. IndepU–kRanks depth (different distributions)

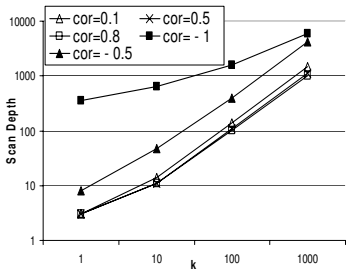

Figure 10. IndepU–Top$k$ (correlations)

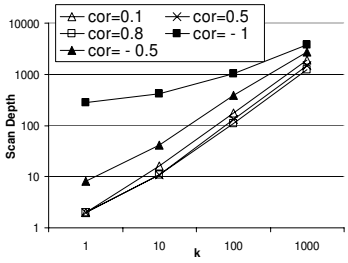

Figure 11. IndepU–kRanks (correlations)

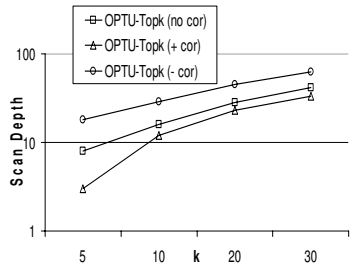

Figure 12. OPTU–Top$k$ (depth)

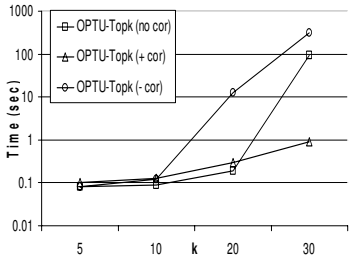

Figure 13. OPTU–Top$k$ (time)

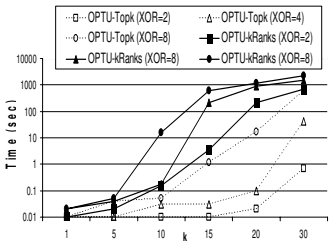

Figure 14. Rule set complexity

6.3 Score-Confidence Correlations

We evaluate here the effect of score-confidence correlation. We generated bivariate gaussian data over score and confidence, and controlled correlation coefficient by adjusting bivariate covariance matrix. Positive correlations result in large savings since in this case high scored tuples are attributed with high confidence, which allows reducing the number of needed-to-see tuples to answer uncertain top–k queries. Figures 10 and 11 show the effect of correlation coefficient on the scan depth of IndepU–Top$k$ and IndepU–kRanks, respectively. Increasing the correlation coefficient from 0.1 to 0.8 reduced the scan depth of IndepU–Top$k$ and IndepU–kRanks by an average of 20% and 26%, respectively. On the other hand, reversed correlation has negative effects on the performance since it leads to consuming more tuples to report answers. Decreasing the correlation coefficient from -0.5 to -1 resulted in an average of 1.5 order of magnitude increase in scan depth for IndepU–Top$k$, and 1 order of magnitude increase for IndepU–kRanks. The effect on execution time is similar.

6.4 Evaluating Space Navigation Techniques

In this experiment, we evaluate the efficiency of OPTU–Top$k$ algorithm. We used databases of exclusive tuples with uncorrelated, positively correlated, and negatively correlated score and confidence values. Figures 12 and 13 show scan depth and execution time of OPTU–Top$k$. Execution time is less than 100 seconds for k reaching 30. Most of this time is spent in maintaining materialized states in the priority queue. For positively correlated data, the time is only under 1 second for all k values. The scan depth of OPTU–Top$k$ increased by an average of 1 order of magnitude between positively and negatively correlated datasets. This is explained based on the fact that highly probable states are obtained quickly by retrieving a small number of tuples from positively correlated data. Algorithm OPTU–kRanks shows similar results which we omit due to space constraints.

6.5 Rule Set Complexity

We evaluate here the impact of model rules complexity on system performance. We implemented a rule engine prototype to compute state probabilities with tuple exclusiveness. We experimented with different rule sets of different *XOR* degrees; which is defined as the number of tuples that are exclusive with some given tuple. Figure 14 shows the execution times of `OPTU-Top`k and `OPTU-`k`Ranks` for different *XOR* degrees. Increasing the *XOR* degree leads to increasing the execution time with an average of one order of magnitude between *XOR=2* and *XOR=4*, and between *XOR=4* and *XOR=8* at the same k values. Increasing *XOR* degrees raises the cost of each request to the rule engine since it increases the possibility that each tuple is exclusive with other tuples in the currently processed state, which leads to larger computational overhead.

7 Related Work

Uncertain data management [15, 4, 16, 18] has received an increasing importance with the emergence of practical applications in domains like sensor networks, data cleaning, and location tracking. The TRIO system [22, 20, 5] introduced different *working models* to capture data uncertainty at different levels, with an elegant perspective of relating uncertainty with lineage with an emphasis on uncertain data modeling. The ORION project [6, 7], deals with constantly evolving data as continuous intervals and presents query processing and indexing techniques to manage uncertainty over continuous intervals. However, it does not address possible worlds semantics under membership uncertainty and generation rules. The ConQuer project [12, 3] introduced query rewriting algorithms for queries on uncertain data to generate consistent answers under possible worlds semantics, and proposed methods to derive probabilities of uncertain data items. The difficulties of top–k processing in sensor networks was addressed in [21] by introducing sampling techniques to guide data acquisition from promising sensors, while illustrating the infeasibility of applying traditional top–k techniques in this setting.

In parallel to this work, the problem of finding the k most probable query answers in probabilistic databases was addressed in [19]. An approach was presented to generate the top–k probable query answers using Monte-Carlo simulation, where computing the exact probability of an answer is relaxed in favor of computing the correct answers efficiently. However, this approach does not address top–k queries with a scoring dimension, where each uncertain data object has both score and probability.

8 Conclusions

To the best of our knowledge, this is the first paper to address top–k query processing under possible worlds semantics. We introduced new formulations interpreting the semantics of top–k queries under uncertainty. We formulated the problem as a state space search, and introduced several query processing algorithms with optimality guarantees on the number of accessed tuples and materialized search states. Our processing framework leverages existing storage and query processing techniques and can be easily integrated with existing DBMSs. Our experimental study shows the efficiency and scalability of our algorithms.

References

[1] The r project for statistical computing: www.r-project.org.

[2] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. In *SIGMOD*, 1987.

[3] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.

[4] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

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

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

[7] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Querying imprecise data in moving object environments. *IEEE Trans. on Knowledge and Data Eng.*, 16(9):1112–1127, 2004.

[8] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

[9] V. de Almeida and R. Hartmut. Supporting uncertainty in moving objects in network databases. In *GIS*, 2005.

[10] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *IJCAI*, 1999.

[11] N. Fuhr. A probabilistic framework for vague queries and imprecise information in databases. In *VLDB*, 1990.

[12] A. Fuxman, E. Fazli, , and R. J. Miller. Conquer: Efficient management of inconsistent databases. In *SIGMOD*, 2005.

[13] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1), 1998.

[14] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[15] T. Imielinski and J. Witold Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[16] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3):419–469, 1997.

[17] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[18] D. S. Nilesh N. Dalvi. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

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

[20] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.

[21] A. Silberstein, R. Braynard, C. Ellis, K. Munagala, and J. Yang. A sampling-based approach to optimizing top-k queries in sensor networks. In *ICDE*, 2006.

[22] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.