Top-k Query Processing in Uncertain Databases

Mohamed A. Soliman[†] Ihab F. Ilyas[†]

Kevin Chen- Chuan Chang[‡]

[†]School of Computer Science, University of Waterloo, Canada
{m2ali,ilyas}@cs.uwaterloo.ca

[‡] Department of Computer Science, University of Illinois at UrbanaChampaign
kcchang@cs.uiuc.edu

University of Waterloo
Technical Report CS-2006-25 (revised August, 2007)*

Abstract

Top-k processing in uncertain databases is semantically and computationally different from traditional top-k processing. The interplay between score and uncertainty information makes traditional top-k processing techniques inapplicable to uncertain databases. In this paper we introduce new probabilistic formulations for top-k queries. Our formulations are based on marriage of traditional top-k semantics with possible worlds semantics. In the light of these formulations, we construct a framework that encapsulates a novel probabilistic model and efficient query processing techniques to tackle the challenges raised by uncertain data settings. We prove that our techniques minimize the number of accessed tuples, and the number of materialized possible query answers. 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 [18, 22, 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 on different levels. According to most of these models, tuples can be uncertain database members, attributed by membership probabilities, e.g., based on data source reliability [13], or fuzzy query predicates, as addressed in [19]. Tuple attributes can also be defined probabilistically over 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.

Several uncertain data models, e.g., [15, 2, 20], adopt *possible worlds* semantics, where a probabilistic database is viewed as a set of possible instances (worlds) associated with their probabilities. The possible worlds space represents an enumeration of all possible views of the database, where each world is a subset of database tuples. The principles of probability theory are used to treat tuples as probabilistic events. Specifically, each tuple t is associated with a corresponding event $t.e$, such that t exists in the database with a probability of $Pr(t.e)$, and does not exist in the database with the complement probability $Pr(\neg t.e) = 1 - Pr(t.e)$. Possible worlds can thus be viewed as conjunctions of tuple events. Under possible worlds semantics, the probability of tuple t that belongs to the answer of some query Q is, *conceptually*, computed by aggregating the probabilities of all possible worlds where t is an answer to Q. Since each possible world is effectively a deterministic database, the operational semantics of relational query operators directly apply to possible worlds.

The structure and probabilities of possible worlds are potentially affected by probabilistic correlations among tuples, e.g., mutual exclusion of tuples that map to the same real world entity [20]. We call such correlations *space generation rules*. Such rules could arise naturally with unclean data [3], or could be customized to enforce application requirements, reflect domain semantics or maintain data dependencies and lineage [25, 20, 5]. Moreover, the relational

*This technical report is based on the paper [23]. The aim of this technical report is to elaborate on the uncertainty model issues, and the assumptions we make in the design of our processing framework.

	Time	Radar Loc	Car Make	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	Nissan	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

	Time	Radar Loc	Car Make	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	Nissan	L-105	105	1.0

Rules: $(t2 \oplus t3)$, $(t4 \oplus t5)$, $(t1 \rightarrow t2)$

(a)

World	Prob.
PW^1={t1,t2,t6,t4}	0.16
PW^2={t1,t2,t5,t6}	0.24
PW^3={t2,t6,t4}	0.12
PW^4={t2,t5,t6}	0.18
PW^5={t6,t4,t3}	0.12
PW^6={t5,t6,t3}	0.18

(b)

Figure 2. Uncertain Database and Possible Worlds Space with a Different Rule Set

processing of probabilistic tuples can induce additional correlations among intermediate query answers, even when the base tuples are uncorrelated [19, 21]. For example, joining the tuple pairs ($r1$, $s1$), and ($r1$, $s2$) results in two correlated join results, since they share the base tuple $r1$, even if all base tuples are uncorrelated.

The huge size of possible worlds space, and the potential complexity of the underlying space generation rules, hinder instantiating and processing worlds explicitly. However, thinking in terms of possible worlds allows for defining proper query semantics. Consider the following example to illustrate the concept of possible worlds.

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 with 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 correctness. Application-specific space generation rules, captured by the indicated exclusiveness rules, are designed to satisfy the following requirement: Based on radar locations, the same car cannot be detected by two different radar units within 1 hour interval.*

Figure 1 (b) shows the possible worlds space for the database in Example 1. Each possible world is one valid subset of the database tuples. The probability of each world is computed as the *joint* probability of the *existence* events of world's tuples, and the *absence* events of all other database tuples. This joint probability is computed based on the probabilities of tuple events, and the generation rules that define their correlations. To visualize this computation, we can think of tuple events as bubbles in a Venn diagram, where the size of each bubble reflects tuple's probability. In this visualization, the generation rules would be constraints that define the areas and the manner with which these bubbles intersect. For example, the rule $(t2 \oplus t3)$ in Example 1 constraints the two bubbles of $t2$ and $t3$ to be disjoint. Each possible world can be visualized as an intersection of some bubbles, while possibly avoiding other bubbles. Figure 1 (b) shows all possible worlds that can be realized in this manner.

To illustrate, assume that we would like to compute the probability of the possible world PW^1. This probability can be expressed as $Pr(PW^1) = Pr(t1.e \wedge t2.e \wedge t6.e \wedge t4.e \wedge \neg t3.e \wedge \neg t5.e)$. Based on the tuples's probabilities, and rules' semantics, the existence of $t2$ *implies* the absence of $t3$, which means that $(t2.e \wedge \neg t3.e) \equiv (t2.e)$. Similarly, the existence of $t4$ *implies* the absence of $t5$, which means that $(t4.e \wedge \neg t5.e) \equiv (t4.e)$. All other tuple events are uncorrelated (*independent*). We can therefore simplify $Pr(PW^1)$ as $Pr(t1.e \wedge t2.e \wedge t6.e \wedge t4.e) = 0.4 \times 0.7 \times 1.0 \times 0.4 = 0.112$. The probabilities of other worlds are computed similarly.

Example 2 *Assume the same database from Example 1, with the following additional rule* $(t1 \rightarrow t2)$*, which means that* $t1$ *implies* $t2$*. Figure 2 shows the new possible worlds space, based on the new rule set.*

Example 2 illustrates how possible worlds and their probabilities are affected by changing the underlying rule set. Specifically, the new rule $(t1 \rightarrow t2)$ constraints the bubble of $t1$ to be contained inside the bubble of $t2$, which also implies that the bubbles of $t1$ and $t3$ are disjoint since $(t2 \oplus t3)$. This new constraint affects the structure and probabilities of all possible worlds. For example, based on the new rule we have $(t1.e \wedge t2.e \wedge \neg t3.e) \equiv (t1.e)$, which modifies the way we compute $Pr(PW^1)$ as follows: $Pr(PW^1) = Pr(t1.e \wedge t2.e \wedge t6.e \wedge t4.e \wedge \neg t3.e \wedge \neg t5.e) =$

$Pr(t1.e \wedge t6.e \wedge t4.e) = 0.4 \times 1.0 \times 0.4 = 0.16$. Moreover, there are no possible worlds containing $t1$, without containing $t2$.

Note that Examples 1 and 2 illustrate simple settings, where possible worlds' probabilities can be computed directly using membership probabilities, and the semantics of the space generation rules. We discuss more elaborate settings in Section 2.1, where we show how these probabilities can be generally computed. In the following, we refer to the value of $Pr(t.e)$ as $t.\texttt{confidence}$.

1.1 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. When uncertainty comes into the picture, such a clean definition does not exist anymore; reporting a tuple in a top–k answer does not depend only on its score, but also on its probability, and the scores and probabilities of other tuples. Tuple scores and probabilities interplay to decide on meaningful 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 can 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 $i = 1 \ldots k$.

While the first interpretation reduces to a top–k query on a deterministic database, it does not conform with possible worlds semantics. In contrast, the second and third interpretations are compliant with possible worlds semantics, however they involve processing challenges, since they involve both ranking and aggregation across worlds. We formally define and elaborate on the differences between these queries in Section 2.2.

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.

1.2 Contributions

Our approach in this paper is to process score and uncertainty within one framework that leverage current DBMS storage and query processing capabilities. Our contributions, towards this goal, 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, to find the most probable top–k answers.
- *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 Uncertainty model and Problem Definition

In this section, we describe the uncertainty model we assume in this paper, followed by our formal problem definition.

2.1 Uncertainty Model

We assume a general uncertainty model that allows for *computing the joint probability of an arbitrary combination*

of tuple events. Computing this probability is the only *interface* [1] between the uncertainty model, and our processing framework (we elaborate on this point in Section 3). This separation of model details and processing allows for great flexibility in adopting different models that describe the uncertainty in the underlying data in different forms. In the following, we describe example models, conforming with our requirements, with different specifications and implementations:

- *Model I (Independent Tuples):* When all tuples events are independent (including intermediate query answers), e.g., key-join queries over independent base tuples, the model is simply the membership probabilities of base tuples. Using such simple model, the joint probability of any combination of tuple events is computed by multiplying the probabilities of the involved tuple events.

- *Model II (Correlated Tuples with Simple Rules):* When tuple events are correlated with simple rules, e.g., implication or exclusiveness, the model maintains these rules, in addition to tuples' probabilities. The joint probability of any combination of tuple events is computed based on tuples' probabilities, and rules semantics. For example, for a rule $(t1 \oplus t2)$ that states that $t1$ is mutually exclusive with $t2$ (e.g., Example 1), we have $Pr(t1.e \wedge t2.e) = 0$, while $Pr(t1.e \wedge \neg t2.e) = Pr(t1.e)$. Similarly, for a rule $(t1 \rightarrow t2)$ that states that $t1$ implies $t2$ (e.g., RFID data where a tuple representing an inventory item *implies* the tuple representing the item's packing cell), we have $Pr(t1.e \wedge t2.e) = Pr(t1.e)$, while $Pr(t1.e \wedge \neg t2.e) = 0$. Please refer to Examples 1 and 2 for more concrete examples. Similar types of rules have been used in currently proposed uncertainty models, e.g., [20, 21], to capture different tuples' correlations. Note that tuples' probabilities have to be *consistent* with rules semantics, otherwise possible worlds semantics would be violated. For example, if $(t1 \oplus t2)$, then we must have $Pr(t1.e) + Pr(t2.e) \leq 1$. Similarly, if $(t1 \rightarrow t2)$, then we must have $Pr(t1.e) \leq Pr(t2.e)$. Studying such consistency issues is out of the scope of our study. We thus assume that tuples' probabilities are always consistent with tuples' correlations. Note also that tuples' marginal probabilities and rules semantics may not be sufficient to compute the joint probability of combinations of tuple events in all cases, as noted in [26]. For example, for a rule $(t1 \vee t2)$ that states that at least one of $t1$ and $t2$ must appear in each possible world, we cannot derive the joint probability distribution of $t1$ and $t2$ based on their marginal probabilities, and the semantics of the rule.

- *Model III (General Inference Model):* The two above models are limited in their scope to special cases. Hence, such models may be insufficient to represent and reason about probabilistic data in more general scenarios. A more general model, subsuming the above simple models, is to maintain the explicit *joint probability distribution* of all database tuples. One compact representation of such huge joint distribution is a Bayesian network, as used in [21], where tuples are the network nodes, and tuples' correlations are maintained in the form of conditional probability tables, allowing representing arbitrary tuples' correlations. Figure 3 illustrates how such model can be used for our purposes, where we show the Bayesian network for the database in Example 2. In the shown network, connected tuples are conditionally dependent, while disconnected tuples are independent. Each tuple maintains a conditional probability table representing its conditional probability distribution given its parents. For example, the third row in the table of tuple $t1$ maintains the two conditional probabilities $Pr(t1.e|t2.e \wedge \neg t3.e)$, and $Pr(\neg t1.e|t2.e \wedge \neg t3.e)$. The dependencies and conditional probability tables are inferred from the semantics of the rules. However, this model is quite general, since it can compactly encode arbitrary dependencies among tuple events. We show how to compute the probability of an arbitrary combination of tuple events using the following example: Assume that we would like to compute $Pr(t1.e \wedge t2.e \wedge \neg t3.e)$. Based on chain rule, this probability can be expressed as $Pr(\neg t3.e) \times Pr(t2.e|\neg t3.e) \times Pr(t1.e|t2.e \wedge \neg t3.e) = 0.7 \times 1.0 \times \frac{0.4}{0.7} = 0.4$, which is the same as $Pr(t2.e)$ as implied by the semantics of the rules.

We discuss some details regarding the implementation of our adopted uncertainty model in Section 3.1. However, we emphasize that model specifications, expressiveness and implementation are not the main focus of our study.

2.2 Problem Definition

Based on possible worlds semantics, 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.

[1] We assume the specifications of the underlying model can be used to compute joint probabilities, without having specific restrictions on these specifications, as shown in the following examples

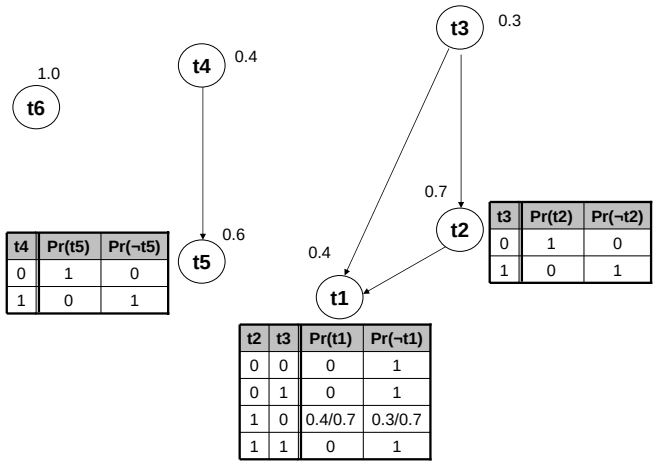

Figure 3. A Bayesian Network Encoding Joint Distribution of Tuple Events

Definition 1 ***Uncertain Top–k Query (U-Top$k$):*** *Let $\mathcal{D}$ be an uncertain database with possible worlds space $\mathcal{PW} = \{PW^1, \ldots, PW^n\}$. Let $\mathcal{T} = \{T^1, \ldots, T^m\}$ 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 $PW(T^i) \subseteq \mathcal{PW}$. U-Top$k$ query over $\mathcal{D}$, 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 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 $\mathcal{F}$. U-kRanks query over $\mathcal{D}$, 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 be the most probable top–k as a set. However, each tuple is a clear winner in its rank over all possible worlds regardless other tuples. This type of queries fits in data exploration scenarios, where the most probable tuples at the 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 $t2$ appears in rank 1 in PW^5 and PW^6 with aggregated probability 0.42, while $t6$ appears in rank 2 in PW^3, PW^5, and PW^8 with aggregated probability 0.324.

In the above definitions, we focus on the "most probable" top-k query answers, for the sake of clarity. However, our definitions extend to probability-ordered answers, i.e., a set of possible answers ranked on probability (the most probable answer, the second most probable answer, etc.). Our methods are thus not restricted to only a single (the most probable) query answer. 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 uncertain top–k processing framework (depicted in Figure 4) that leverages RDBMS storage, indexing and query processing techniques to compute the most probable top–k answers in an uncertain database. The main motivations behind the design of our framework are summarized in terms of the following design principles:

- **DP1:** To build on top of an RDBMS as our tuple access layer. We use an underlying RDBMS to store and query probabilistic data and uncertainty information. Our processing framework leverages RDBMS storage, indexing and query processing capabilities to compute probabilistic top-k queries. Similar arguments are made in the design of the TRIO system [25, 5].
- **DP2:** To leverage current algorithms for top-k query processing of deterministic data. In particular, our framework takes advantage of rank-aware query processing (if supported by the underlying DBMS), e.g., [14, 17], to minimize the number of needed-to-access tuples.
- **DP3:** To implement efficient probability-guided search algorithms to realize query answers. The algorithms lazily materialize the search space, by maintaining only ordered *prefixes* of possible worlds with the highest chances to be among query answers.

These design principles are realized by the two-layer architecture in Figure 4. We next give a detailed description for framework components and describe their interactions (Section 3.1), followed by discussing our tuple retrieval model (Section 3.2), and our problem space (Section 3.3).

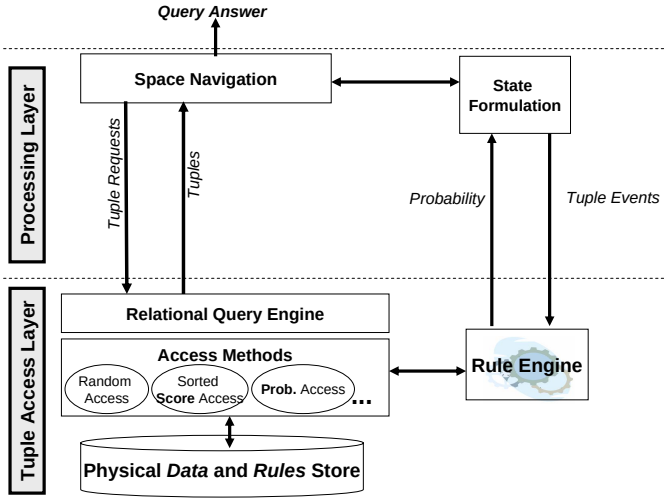

Figure 4. Uncertain Top–k Processing Framework

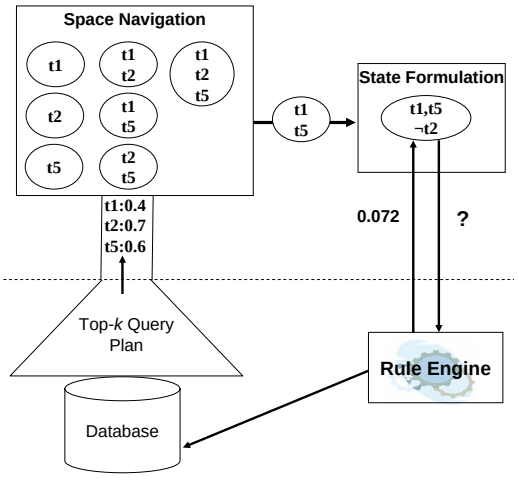

Figure 5. Interaction of Framework Components

3.1 Framework Details

Tuple Access Layer. Tuple retrieval, indexing and traditional query processing (including score-based ranking) are the main functionalities supported by the *Tuple Access Layer*. The *Tuple Access Layer* executes an SPJ query, which acts as the tuple source to the upper layer. We show in Section 3.2 that sorted score access for output tuples of such query is essential for efficient processing.

While our techniques can benefit largely from efficient support to ranking in the *Tuple Access Layer*, we emphasize that our framework is still valid if such support is limited or lacking. However in this case, more tuples would need to be accessed to realize query answers. For example, a complete sorting of query results may be required if rank-aware processing is not supported.

Rule Engine. This module is responsible for computing the probabilities of arbitrary combinations of tuple events. We assume an *interface* to the *Rule Engine* receiving, as input, an arbitrary combination of tuple events, and producing, as output, the probability of such combination. The details of the *Rule Engine* is not the focus of our study, since they vary according to how sophisticated the underlying uncertainty model is, as discussed in Section 2.1. To give an example, reference [19] shows how to generate *safe* plans for some class of SPJ queries, where the independence of involved tuple events is exploited to facilitate the probability computation of query output tuples. A simple *Rule Engine* that maintains the membership probabilities of base tuples can be sufficient in this case. Alternatively, reference [21] proposes using a full-fledged Bayesian network to materialize the conditional probability distributions of dependant tuples. A much more sophisticated *Rule Engine* needs to be built in this case to query the network for arbitrary combinations of tuple events. Hence, we believe that "black-boxing" the *Rule Engine* adds versatility to our framework, since it does not restrict our techniques to a specific implementation of probabilistic inference techniques.

In our prototype, we experimented with different implementations of the *Rule Engine* module including: (1) a simple engine that supports probability computation over independent tuple events; (2) an engine compliant with the x-tuple model [20, 5], where tuples are correlated with exclusiveness rules only; and (3) a sophisticated engine that implements and indexes a Bayesian network that is used progressively during query processing to load relevant dependency information on demand, and compute the probabilities of tuple combinations through Bayesian inference techniques. The Bayesian network implementation is adopted in our system demonstration [24]. We refrain from extensively discussing the implementation details of the *Rule Engine* module in this paper, and abstract such details using our interface to the underlying uncertainty model.

Processing Layer. The processing layer retrieves uncertain tuples from the underlying *Tuple Access Layer*, and efficiently navigates the possible worlds space to compute query answers. The components of this layer are the *State Formulation* module, which formulates search states as combinations of tuple events; and the *Space Navigation* module, which uses search algorithms to partially materialize the necessary parts in the possible worlds space, while looking for query answers. We give the formal definition of the problem space in Section 3.3.

Assumptions. We make the following assumptions in our framework design:

- *Assumption 1 (Sequential Access):* We assume that tuples are consumed sequentially, i.e., one by one, from the output of a query plan running in the *Tuple Access Layer*. That is, the *Processing Layer* does not have random access to some tuple t unless all the tuples preceding t in the output of the *Tuple Access Layer* are already produced. This assumption implies that we do not have prior information on the total number of output tuples from the *Tuple Access Layer*. Such lack of information is an important justification for our tuple retrieval model discussed in Section 3.2.

- *Assumption 2 (Unknown Future Dependencies):* We assume that the dependencies among query output tuples are computed *lazily* only when these tuples are produced from the *Tuple Access Layer*. This assumption implies that we do not know whether the current output tuple would be correlated with other future output tuples or not.

The justification for our *Sequential Access* assumption is that we build on top of a query plan running in the *Tuple Access Layer*. These settings do not allow for random access to arbitrary tuples, unless all query output tuples are fully computed. Clearly, computing the full query output should be avoided in our settings, since top-k queries can be usually computed by *seeing* only a small fraction of query output tuples. We thus assume an *iterator* interface, widely used in RDBMS's, to retrieve tuples from the *Tuple Access Layer* on demand, and pass them to the *Processing Layer*. Since processing each tuple request by the *Tuple Access Layer* can be costly, we aim at pipelining tuple requests so that we consume only the necessary tuples to compute probabilistic top-k queries. We elaborate on the implication of this requirement on our retrieval model in Section 3.2.

The justification for our *Unknown Future Dependencies* assumption is that dependencies among currently consumed tuples from the *Tuple Access Layer*, and future tuples cannot be known a priori, since dependencies may arise during query execution. For example, assume a simple query plan $(R \bowtie S)$. Assume a rule $(s1 \rightarrow s2)$ that applies to tuples $s1$ and $s2$ in S. Let $j1 = (r1, s1)$, $j2 = (r1, s2)$, and $j3 = (r3, s3)$ be three different join results. Then, $j1$ and $j3$ are independent, while $j1$ and $j2$ are correlated by the rule $(j1 \rightarrow j2)$ that is induced from the existing rule $(s1 \rightarrow s2)$. Therefore, by the time we consume $j1$, we cannot predict whether it would be correlated with other future tuples or not, unless we compute the full join results.

3.2 Retrieval Model

Since we would like to minimize the number of tuple requests to the *Tuple Access Layer*, we need to incrementally consume the *necessary* tuples to compute query answers. A tuple t is *necessary* if it satisfies at least one of the two following conditions: (1) t modifies the set of currently known top-k answers, and (2) t modifies our information about the yet unknown top-k answers. To illustrate, consider the next example.

Example 3 *Assume $t1$ is the output tuple from the* Tuple Access Layer *with the highest score, and probability $p1$. Alternatively, assume $t2$ is the output tuple with the highest probability $p2$. Consider a U-Top1 query.*

In Example 3, $t1$ is *necessary* since its retrieval reveals two pieces of information: (1) $< t1 >$ is a candidate U-Top1 answer with probability $p1$, and (2) 1-$p1$ is an upper bound over the probability of any other possible U-Top1 answer. Consider alternatively $t2$. We cannot create a new candidate U-Top1 answer based only on $t2$, since we do not know the tuples higher in score than $t2$ (*Sequential Access* assumption). In fact, a bound on the number of such tuples is the same as the number of all non-retrieved tuples, which is not known in advance (without completely executing the query in the *Tuple Access Layer*). Estimating such bound boils down to selectivity estimation of the underlying query, which cannot guarantee accurate bounding. Moreover, even if such bound is available, e.g., based on worst case estimation, we still cannot compute the probability that $t2$ is a U-Top1 answer, since we are unaware of any potential dependencies between $t2$ and non-retrieved tuples with higher scores (*Unknown Future Dependencies* assumption). The only *safe* probability bounds we can derive for $< t2 >$, being a U-Top1 answer, are thus $[0, p2]$. For the same reason, $t2$ does not modify our information about the yet unknown top-k answers. We thus conclude that retrieving $t2$ is not *necessary* as long as it is out of score order.

Based on the above discussion, we prove that consuming tuples from the *Tuple Access Layer* one by one in *score order* minimizes the number of retrieved tuples to answer probabilistic top-k queries. That is, retrieving tuples in any other sequential order is useless, and cannot be used to obtain query answers unless all tuples are seen. Rank-aware query processing techniques, e.g., [14, 17], could be used in the *Tuple Access Layer* to pipeline tuples in score order efficiently.

Theorem 1 *Without having information about the total number tuples that would be produced by the underlying relational engine, and the potential dependencies between the current and future tuples, sorted score order yields the minimum number of retrieved tuples to answer probabilistic top-k queries, while retrieving tuples sequentially.*

Proof: Assume an algorithm $\mathcal{A}$ that retrieves tuples sequentially out of both *score* order and `confidence` 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 a sufficient number of unseen tuples, whose events are independent with $t.e$ and scores are higher than t, to make the probability that t belongs to any possible top-k answer arbitrarily small. Algorithm $\mathcal{A}$ cannot assert this fact unless it sees all tuples. Then, $\mathcal{A}$ cannot answer uncertain top-k queries while consuming less tuples than a sorted score access method.

Assume alternatively that $\mathcal{A}$ retrieves tuples out of score order, but in `confidence` order. In this case $\mathcal{A}$ cannot also 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. The yet unknown potential dependencies between t, and the non-retrieved tuples with higher scores, do not allow computing non-trivial bounds over the probability that t belongs to any top-k answer. □

Note that if further information is available, score order may not be the optimal order. Specifically, knowing the total number of tuples in query output, and that all output tuples are independent, we may retrieve a smaller number of tuples using probability order [26]. For example, assume the number of output tuples is 10, and that the two top tuples in probability order are $t1$ and $t2$ with probabilities 1.0 and 0.1, and scores 1 and 10, respectively. Then, at this point we can find the U-Top1 answer, since the probability of $t1$ to be the U-Top1 is at least $(1 - 0.1)^9 > 0.1$, while any other tuple would have a maximum probability of 0.1 to be the U-Top1. However, as discussed earlier, knowing the number of output tuples in advance defeats the essence of optimization of top-k processing, which is limiting the number of accessed tuples. Furthermore, assuming independence among query output tuples applies only to simple queries (please refer to our discussion in Section 2.1).

Cost Metric. Based on problem definition in Section 2.2, the total number of possible top-k answers that can be obtained from n retrieved tuples is bounded by $\binom{n}{k}$, which is in $O(n^k)$. Since k is a constant, our primary cost metric is n, the number of consumed tuples from the *Tuple Access Layer*. Additionally, since we aim at searching the space of possible answers, we would like to minimize the size of the materialized space.

3.3 Problem Space

We formulate our problem as searching the space of states that represent 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 valid 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 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.

Based on the above space definition, we give an overview of the interaction of framework components using Figure 5, which describes how to process an uncertain top–k query for the database in Example 1. In Figure 5, 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 the probability of each state, the *State Formulation* module formulates that state and compute its probability by contacting the *Rule Engine* (details are given in Section 4.1). For example, to formulate a state for the tuple vector $< t1, t5 >$, the intermediate tuple $t2$, based on score order, must be absent.

In Section 4, we describe our search algorithms that partially materialize the space of top–k answers to compute the most probable answers by retrieving the least possible number of tuple, and by materializing the least number of search states.

4 Navigating the Search Space

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

4.1 Computing State Probabilities

We consider each 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 the probabilities of the possible worlds 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:*** *When extending any combination of tuple events by adding another tuple existence/absence event, the resulting combination will have 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 under our model since for any two sets of tuple events E_n and E_{n+1} (with lengths n and $n+1$, respectively), where $E_n \subset E_{n+1}$, the set of possible worlds where E_{n+1} is satisfied is $\subseteq$ the set of possible worlds where E_n is satisfied.

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 m accesses to $\mathcal{D}$ in $\mathcal{F}$ order, let s_l be some search state, and $I_{l,m}$ 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: $\mathcal{P}(s_l) = Pr(s_l \cap \neg I_{l,m})$

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.

Search states can be extended as follows. Assume a current 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, our query processing algorithm for U-Top$k$ queries. `OPTU-Top`k keeps all ranked tuples retrieved from the *storage layer* in a buffer of seen tuples. `OPTU-Top`k 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 the state with the highest probability of contributing to the final top–k answer. The extension is performed using the next tuple drawn either from the buffer or from the underlying database.

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 *Tuple Access Layer*. Furthermore, we define $s_{0,0}$ as an *empty state* of *length* 0, where $\mathcal{P}(s_{0,0}) = 1$. The state $s_{0,0}$ is used to upper bound the probability of any non-materialized state, since any non-materialized state must be produced from an extension of $s_{0,0}$.

Let $\mathcal{Q}$ be a priority queue of states ordered on their probabilities, where ties are broken by state *length*. We initialize $\mathcal{Q}$ with $s_{0,0}$. Let d be the number of seen tuples from the database at any point. `OPTU-Top`k iteratively retrieves the top state in $\mathcal{Q}$, say $s_{l,i}$, extends it into the two next possible state (Section **??**), and inserts the resulting two states back to $\mathcal{Q}$ according to their probabilities. Extending $s_{l,i}$ will lead to consuming a new tuple from the database only if $i = d$, otherwise $s_{l,i}$ can be extended using the buffered tuple pointed to by $i+1$.

The termination condition of `OPTU-Top`k is when the top state in $\mathcal{Q}$ is a *complete* state. If a *complete* state $s_{k,n}$ is on top of $\mathcal{Q}$, then both materialized and non-materialized states have smaller probabilities than $s_{k,n}$. This means that there is no way to generate another *complete* state that will beat $s_{k,n}$, based on Property 1. Algorithm 1 describes the details of `OPTU-Top`k.

Note that in addition to extending the state space *lazily*, i.e., only the top state in $\mathcal{Q}$ is extended, Algorithm `OPTU-Top`k also applies the following *pruning criterion* to significantly reduce the number of buffered states in $\mathcal{Q}$ (line 17): As soon as a *complete* state $s_{k,n}$ is reached, all the buffered states, whether complete or not, with probabilities less than $\mathcal{P}(s_{k,n})$ can be safely pruned, based on Property 1.

We prove the optimality guarantees of `OPTU-Top`k regarding the number of accessed tuples (Theorem 2), and the number of materialized states (Theorem 3).

Theorem 2 *Among all algorithms that retrieve tuples ordered on score, Algorithm `OptU-Top`k retrieves the minimum number of tuples to report U-Top$k$ query answer.*

Proof: Algorithm `OptU-Top`k consumes score-ranked tuples until no state has a higher probability than some *complete* state. Assume another algorithm, $\mathcal{A}$, that also consumes tuples sequentially ordered on score but reports U-Top$k$ answer by consuming $d \geq k$ tuples, where d is strictly less than the number of tuples consumed by `OptU-Top`k. Assume x_k is the *complete* state reported by $\mathcal{A}$. Assume `OptU-Top`k runs until it consumes d tuples. At this time, there exists some state s_l where $l < k$ and $\mathcal{P}(s_l) > \mathcal{P}(x_k)$. Assume that there exist $k - l$ tuples with `confidence` 1, not yet seen by s_l, and independent with tuples in s_l. We can augment s_l with these $k - l$ tuples to compose another *complete* state s_k, where $\mathcal{P}(s_k) = \mathcal{P}(s_l) > \mathcal{P}(x_k)$. Then x_k, the answer reported by $\mathcal{A}$, is incorrect. □

Theorem 3 *Algorithm `OptU-Top`k visits only the necessary states to compute U-Top$k$ query answer.*

Proof: Let x_k be the answer reported by `OptU-Top`k. Assume another algorithm $\mathcal{A}$ that also concludes x_k as the

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

Require:
$source$: Score-ranked tuple stream
k: Answer length
Ensure: U-Top$k$ query answer

```
1:  Q ← empty priority queue for states ordered on probabilities
2:  d ← 0
3:  Insert s_{0,0} into Q {init empty state}
4:  while ( source is not exhausted AND Q is not empty) do
5:     s_{l,i} ← dequeue (Q)
6:     if (l = k) then
7:        return s_{l,i}
8:     else
9:        if (i = d) then
10:          t ← next tuple from source
11:          d ← d + 1
12:       else
13:          t ← tuple at pos i + 1 from seen tuples
14:       end if
15:       Extend s_{l,i} using t into s_{l,i+1}, s_{l+1,i+1} {Section 4.1}
16:       Insert s_{l,i+1}, s_{l+1,i+1} into Q
17:       if (l + 1 = k) then
18:          remove any state s ∈ Q where P(s) < P(s_{l+1,i+1})
19:       end if
20:    end if
21: end while
22: return dequeue(Q)
```

final U-Top$k$ answer. Let s_l be a state skipped by $\mathcal{A}$, and visited by OptU-Top$k$, and let $p = \mathcal{P}(s_l)$. By definition s_l has the highest probability among all states accessible to OptU-Top$k$ and $\mathcal{A}$. Assume there exist $k - l$ tuples with confidence 1 and independent with tuples in s_l. Then, s_l can be extended to a *complete* state s_k with probability p. Hence, Algorithm OptU-Top$k$ reports $x_k = s_k$ as its final answer. However, 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 probability higher than p □

Note that under the assumption that all states probabilities are distinct (e.g., by assuming a unified tie-breaker mechanism), there is 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 proof of Therorem 3 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-kRanks, our query processing algorithm for U-kRanks queries. Algorithm OPTU-kRanks 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 m 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 seen tuple from the database. In other words, we can compute $P_{t,i}$, for $i = 1 \dots m$, as soon as we retrieve t from the database.

For each rank i, we need only to remember the most probable answer obtained so far. This is because any unseen tuple u cannot change $P_{t,i}$ of any seen tuple t, since u can never appear before t in any possible world. The remaining question is when can we conclude an answer for each rank i. To be able to report an answer for rank i, we need to be sure that any unseen tuple will not beat the current answer. Let S_j be the current set of states with length j, and let $Z_j = \sum_{s \in S_j} \mathcal{P}(s)$. Note that for any rank i, the value of $\sum_{j<i} Z_j$ can never increase when new tuples are consumed. Therefore, the maximum probability for an unseen tuple u to be at rank i is $\sum_{j<i} Z_j$. We formally prove this bound in Theorem 4. Let t^* be the current U-kRanks answer for rank i. The termination condition of Algorithm OPTU-kRanks, for rank i, is thus $P_{t^*,i} > \sum_{j<i} Z_j$. Algorithm 2 describes the details of Algorithm OPTU-kRanks. Theorem 4 formalizes the optimality of OPTU-kRanks based on the above discussion.

Theorem 4 *Among all algorithms that retrieve tuples ordered on score, Algorithm OptU-kRanks retrieves the minimum number of tuples to report U-kRanks query answer.*

Proof: Assume another algorithm, $\mathcal{A}$, that also consumes tuples sequentially ordered on score but reports t as the U-kRanks answer for rank i while consuming less tuples than Algorithm OPTU-kRanks. Assume $\mathcal{A}$ has consumed d tuples. Assume OPTU-kRanks runs until it consumes the same d tuples. At this time, we have $P_{t,i} < \sum_{j<i} Z_j$. Let $\{u_1, \dots, u_{i-1}\}$ be the next $i-1$ unseen tuples. Assume each tuple u_j is *implied* by each state in S_{j-1}, and is (almost) *exclusive* with any other state s_l, where $l \neq j-1$. It follows that u_j will extend all states in S_{j-1} into states of length j with exactly the same probabilities, and no state s_{j-1} will be remaining. Additionally, the probability and the length of any other state s_l with $l \neq j-1$ will not change. By induction, it follows that $P_{u_{i-1},i} = \sum_{j<i} Z_j$.

Algorithm 2 OptU-kRanks($source$,k)

Require:
$source$: Score-ranked tuple stream
k: Answer length
Ensure: U-kRanks query answer
1: $answer[1 \dots k] \leftarrow \phi$ {Answer vector}
2: $ubounds[1 \dots k] \leftarrow [1, 1 \dots 1]$ {unseen upper-bound probabilities for different ranks}
3: $reported \leftarrow 0$ {No. of reported answers}
4: $depth \leftarrow 1$
5: $candidates \leftarrow \phi$ {current set of states}
6: **while** ($source$ is not exhausted AND $reported < k$) **do**
7: $t \leftarrow$ next tuple from $source$
8: Update $canididates$ based on t
9: Update $ubounds$ based on $canididates$
10: **for** i=1 to $min(k, depth)$ **do**
11: **if** ($answer[i]$ was previously reported) **then**
12: continue
13: **end if**
14: Compute $P_{t,i}$
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**

Then u_{i-1} is the correct answer that should have been reported by $\mathcal{A}$. □

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-kRanks 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-kRanks 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, for a partial state s, if we can compute the maximum probability of a valid *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 any given state s to be used for pruning, 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* on the number of accessed tuples.

5.1 U-Top$k$ Queries Under Independence

Under tuple independence, we can aggressively prune the state space to keep only the states that could lead to the answer. Algorithm IndepU-Top$k$ exploits this property by pruning the space based on the following state-comparability criterion.

Definition 4 ***Comparable States:*** *Under tuple independence, two states x_l and y_l, maintained after seeing the same number of score-ranked tuples, are comparable* □

Definition 4 states that under tuple independence, if two states are maintained after seeing m tuples, and they have the same *length*, then they can be compared based on their probabilities regardless how they will be extended to *complete* states. This is because for two *comparable* states x and y to generate the most probable *complete* states c_x and c_y, from x and y, respectively, the same set of tuples will be appended to both x and y. The consequence of Definition 4 is that if states x and y are *comparable*, and $\mathcal{P}(x) < \mathcal{P}(y)$, then we can safely prune x from our search space.

IndepU-Top$k$ exploits Definition 4 by grouping states into equivalence classes based on their *lengths*. IndepU-Top$k$ keeps *only one* state for each *length* value $0 \dots k$ 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 6(fractions indicate confidence values, and scores are omitted for brevity), where we are interested in the U-Top3 answer. We represent each state s_l with its tuple vector, and distinguish tuples seen from $\mathcal{D}$ but not included in s_l by 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. In this step 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. The search continues until the second termination condition is met. Space reduction by exploiting the 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 Under Independence

Under tuple independence, 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 now describe `IndepU-`k`Ranks`, our processing algorithm for U-kRanks queries under independence.

We illustrate `IndepU-`k`Ranks` algorithm using the example depicted by Figure 7, 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 rank 1 probability of a tuple x 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 computation of the probabilities in the remaining rows is based on the following property:

Property 2 *Under tuple independence and for* $i > 1$, $P_{x,i} = Pr(x) \times \sum_{y:\mathcal{F}(y)>\mathcal{F}(x)}(\prod_{z:\mathcal{F}(x)<\mathcal{F}(z)<\mathcal{F}(y)}(1 - Pr(z)) \times P_{y,i-1}$ □

The rationale behind Property 2 is that under independence for 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 7, $P_{t2,2} = 0.9 \times 0.3 = 0.27$, 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 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 top to bottom based on the following equation:

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

For example in Figure 7, $M[2,3] = Pr(t3) \times (M[1,2] + (1 - Pr(t2)) \times M[1,1])$. Algorithm `IndepU-`$k$`Ranks` returns a set of k tuples $\{t_1 \dots t_k\}$, where $t_i = arg\ Max_x\ M[i,x]$.

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) Scan Depth: the number of sequentially accessed tuples to report uncertain top–k answers. In all our 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.

Since the study of efficient dependency evaluation techniques is beyond the scope of this paper, we implemented an *example Rule Engine* that computes the probabilities of partial states under tuple exclusiveness. However, our methods are not restricted to exclusiveness rules only, as discussed in Section 2.1. In [24], we demonstrated our methods with more sophisticated dependencies implemented using a Bayesian network *Rule Engine*. We do not discuss the experimental results of this prototype in this paper.

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

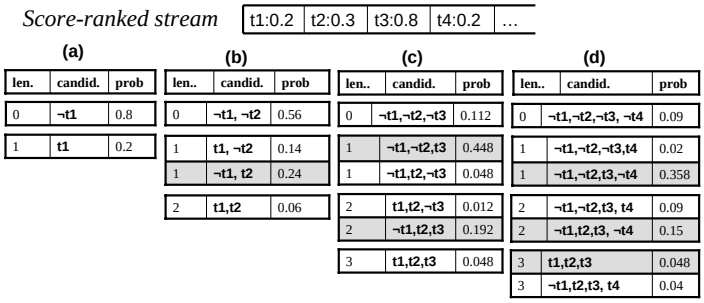

Figure 6. IndepU-Top$k$ Processing

Figure 7. IndepU-kRanks Processing

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 and some tuple 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 8, 9 show the time and scan depth of IndepU-Top$k$, respectively, while Figures 10, 11 show the time and scan depth of IndepU-kRanks, 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-kRanks 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-kRanks.

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 12 and 13 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 the General Algorithms

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 14 and 15 show the scan depth and execution time of OPTU-Top$k$ algorithm. The execution time is under 100 seconds for values of k reaching 30. The time is spent by the algorithm in maintaining the materialized states in the priority queue before concluding an answer. However, 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 when going from positively to negatively correlated datasets. This can be explained based on the fact that for positively correlated data, highly probable states are obtained quickly after retrieving a small number of tuples, while for negatively correlated data more tuples

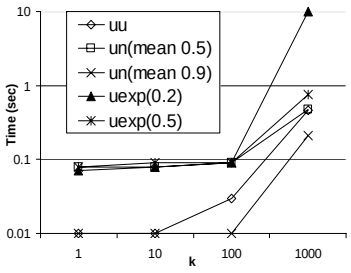

Figure 8. IndepU-Top$k$ time (different distributions)

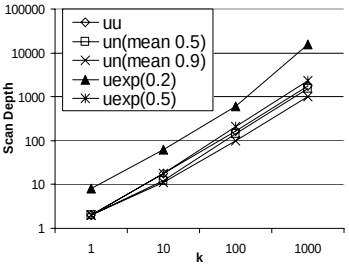

Figure 9. IndepU-Top$k$ depth (different distributions)

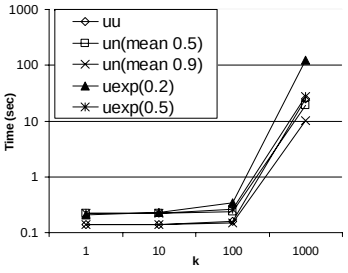

Figure 10. IndepU-kRanks time (different distributions)

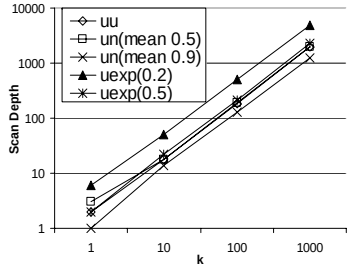

Figure 11. IndepU-kRanks depth (different distributions)

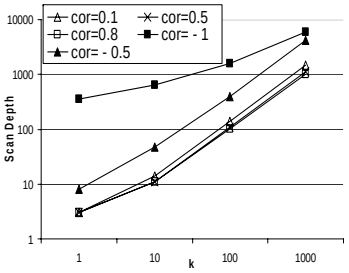

Figure 12. IndepU-Top$k$ (correlations)

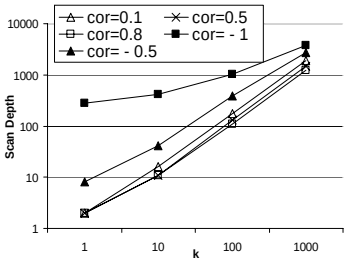

Figure 13. IndepU-kRanks (correlations)

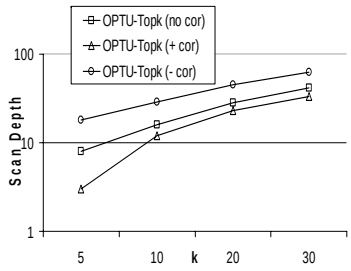

Figure 14. OPTU-Top$k$ (depth)

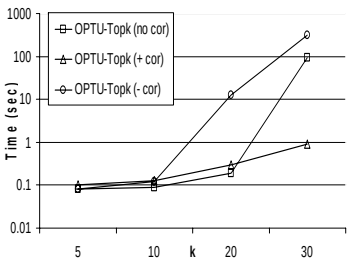

Figure 15. OPTU-Top$k$ (time)

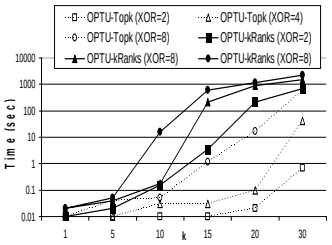

Figure 16. Rule set complexity

need to be seen before concluding an answer leading to materializing more states in the priority queue. We omit the results of OPTU-kRanks due to space constraints.

6.5 Rule Set Complexity

We evaluated the effect of potential complexity of model rules on the performance. We experimented with different rule sets with different *XOR* degrees; which is the number of tuples that are exclusive with some given tuple. Figure 16 shows the execution times of OPTU-Top$k$ and OPTU-kRanks with different *XOR* degrees. Increasing the *XOR* degree results generally in increasing the execution time with an average of one order of magnitude when going from *XOR=2* to *XOR=4*, or *XOR=4* to *XOR=8* at the same value of k. Increasing *XOR* degrees raises the cost involved in each request to the rule engine since it increases the possibility that a newly seen 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, 19] has received an increasing importance with the emergence of different practical applications in domains like sensor networks, data cleaning, and location tracking. The Trio system [25, 20, 5] introduced different *working models* to capture data uncer-

tainty at different levels, and presented an elegant perspective of relating uncertainty with lineage and leveraging existing DBMSs, 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 to extract clean and consistent answers from unclean data 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 [22] by introducing sampling techniques to guide the acquisition of data from promising sensors, while illustrating the infeasibility of applying traditional top–k techniques in this domain because of the interplay of uncertainty and score information.

A recent work in [26] builds on our query definitions, and presents efficient algorithms to handle special cases of our general search algorithms, where tuples are either independent, or mutually exclusive, and only the most probable U-Top$k$ or U-kRanks answer is required. The work in [26] adopts sorted score access to allow for probability computation, and uses special properties of independent and exclusive tuples to heavily prune the answer space achieving efficient execution.

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 experiments show the efficiency and scalability of our proposed 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] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

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

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

[21] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. 2007.

[22] 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.

[23] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.

[24] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Urank: formulation and efficient evaluation of top-k queries in uncertain databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

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

[26] K. Yi, F. Li, D. Srivastava, and G. Kollios. Efficient processing of top-k queries on uncertain databases. Technical Report, AT&T Labs. 2007.