Probabilistic Top-*k* and Ranking-Aggregate Queries

MOHAMED A. SOLIMAN and IHAB F. ILYAS
University of Waterloo
and
KEVIN CHEN–CHUAN CHANG
University of Illinois at Urbana-Champaign

Ranking and aggregation queries are widely used in data exploration, data analysis, and decision-making scenarios. While most of the currently proposed ranking and aggregation techniques focus on deterministic data, several emerging applications involve data that is unclean or uncertain. Ranking and aggregating uncertain (probabilistic) data raises new challenges in query semantics and processing, making conventional methods inapplicable. Furthermore, uncertainty imposes probability as a new ranking dimension that does not exist in the traditional settings.

In this article we introduce new probabilistic formulations for top-*k* and ranking-aggregate queries in probabilistic databases. Our formulations are based on marriage of traditional top-*k* semantics with possible worlds semantics. In the light of these formulations, we construct a generic processing framework supporting both query types, and leveraging existing query processing and indexing capabilities in current RDBMSs. The framework encapsulates a state space model and efficient search algorithms to compute query answers. Our proposed techniques minimize the number of accessed tuples and the size of materialized search space to compute query answers. Our experimental study shows the efficiency of our techniques under different data distributions with orders of magnitude improvement over naïve methods.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Query processing, probabilistic data, top-k, ranking, aggregation

Support for I. F. Ilyas was provided in part by the Natural Sciences and Engineering Research Council of Canada through Grant 311671-05.

Authors' addresses: M. A. Soliman and I. F. Ilyas, University of Waterloo, 200 University Ave. West, Waterloo, Ontario, Canada N2L 3G1; email: {m2ali,ilyas}@uwaterloo.ca; K. C.-C. Chang, Computer Science Department, University of Illinois at Urbana-Champaign, 306 Engineering Hall, MC 266, 1308 West Green Street, Urbana, IL 61801; email: kcchang@cs.uiuc.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2008 ACM 0362-5915/2008/08-ART13 $5.00 DOI 10.1145/1386118.1386119 http://doi.acm.org/10.1145/1386118.1386119

ACM Reference Format:
Soliman, M. A., IIyas, I. F., and Chang, K. C.-C. 2008. Probabilistic top-k and ranking-aggregate queries. ACM Trans. Datab. Syst. 33, 3, Article 13 (August 2008), 54 pages. DOI = 10.1145/1386118.1386119 http://doi.acm.org/10.1145/1386118.1386119

1. INTRODUCTION

Efficient processing of ranking and aggregation queries is a crucial requirement in many domains such as Web search and OLAP. Several techniques have been proposed to compute top-k [Hristidis et al. 2001; Natsev et al. 2001; Ilyas et al. 2004; Li et al. 2005], and ranking-aggregate queries [Li et al. 2006] in relational databases. A top-k query reports the k tuples with the highest *scores* based on some scoring function, while a ranking-aggregate query (also referred to as "top-k aggregate query") ranks groups of records by their aggregate values and returns the k groups with the highest aggregates. Most proposed techniques in this area focus on minimizing the number of accessed tuples (or groups) to conclude the top-k answers, while avoiding sorting the complete query results.

In many current applications, data involves uncertainty and imprecision. The imprecision of real world data entails different semantics for top-k and top-k aggregate queries, and motivates the need for new query formulations and efficient processing techniques to address ranking and aggregation from a probabilistic perspective.

1.1 Motivation and Challenges

In several applications, efficient support for ranking and aggregation of probabilistic data is required. In the context of the Web, information extraction techniques are used to extract instances of entities, for example, organization and person names. The imperfection of extraction tools and the inherent ambiguity of unstructured text introduce significant uncertainty in the extracted information. Ranking such probabilistic information based on some scoring measure is important to a wide range of applications. For example, a popularity survey may require ranking movies based on the extracted ratings from online reviews. In the context of sensor networks, sensor readings are usually represented using probabilistic models, derived from readings history and sensor correlations. Many applications can benefit from aggregating sensor data. For example, grouping sensor readings by location to find the top-k locations based on average temperature. Similar application examples exist in the contexts of schema matching, for example: He and Chang [2006], and data integration for example, Bhattacharya et al. [2006], Andritsos et al. [2006]. We use the following simple example to illustrate the challenges involved in ranking and aggregating probabilistic data.

Example 1. In a traffic-monitoring system, radars detect cars' speeds automatically, while car identification, for example, by plate number, is performed by a human operator, or OCR of plate number images. In this system, multiple sources contribute to data uncertainty (e.g., high voltage lines that interfere with radars affecting their precision, close by cars that cannot be distinguished,

Readings

	Time	Location	Make	PlateNo	Speed	Prob
t1	11:45	L1	Honda	X-123	130	0.4
t2	11:50	L1	Toyota	Y-245	120	0.7
t3	11:35	L2	Toyota	Y-245	80	0.3
t4	12:10	L3	Mazda	W-541	90	0.4
t5	12:25	L4	Mazda	W-541	110	0.6
t6	12:15	L4	Nissan	L-105	105	1.0

(a)

Query 1

```
SELECT PlateNo, Make, Time
FROM Readings
WHERE Time BETWEEN '11:30' AND
'12:30'
ORDER BY Speed DESC
LIMIT k
```

Query 2

```
SELECT Location, Average(Speed)
FROM Readings
GROUP BY Location
ORDER BY Average(Speed) DESC
LIMIT k
```

(b)

Fig. 1. (a) A relation with tuple uncertainty; (b) example ranking and aggregation queries.

or plate number images that are not clear enough to identify the car precisely). Figure 1(a) is a snapshot of speed `Readings` relation in the last hour. The special attribute "Prob" in each tuple indicates the probability that the whole tuple gives correct information. This probability can be obtained from various sources. For example, history of previous readings might indicate that 70% of the readings obtained from radars close to high voltage lines are actually correct. Hence, the readings of a radar unit that is close to high voltage lines can be assumed to be correct with probability 0.7. Other sources (e.g., clearness of plate number images) can be additionally incorporated to better quantify tuple's uncertainty.

Figure 1(b) shows two queries to be evaluated on the `Readings` relation in Example 1. Query 1 requests the top-*k* speeding cars in one hour interval, which can be used, for example, in an accident investigation scenario, while Query 2 requests the top-*k* locations based on average speed, which can be used, to recommend the most suitable locations of speed traps, for example. While both queries are clear in deterministic databases, we show that their interpretation and processing is challenging for probabilistic databases.

In Query 1, although tuple score (the *Speed* attribute) is deterministic, the tuples are probabilistic. Both tuples' *probabilities* and *scores* need to be factored in our interpretation of this query. This effectively introduces two interacting ranking dimensions that interplay to decide meaningful query answers. For example, it is not meaningful to report a top-scored tuple with insignificant probability. Moreover, combining scores and probabilities into one measure, using some aggregation function, eliminates uncertainty completely, and loses valuable information that can be used to get more meaningful answers conforming with probabilistic query models (we elaborate on this point in Section 2).

In the case of aggregation, additional challenges arise. In Query 2, each location (group) involves a number of probabilistic tuples. For example, for location $L1$, there is a chance that a *Honda* and a *Toyota* tuples are correctly recorded, only one of them is correctly recorded, or none of them is correctly recorded. To compute *Average*(*Speed*), the aggregate value of each group, we need to take all

possibilities into account. This means that we have, for each group, a probability distribution over its aggregate value. Computing such probability distribution is shown to be in NP [Ross et al. 2005]. This is a clear departure from the conventional semantics of aggregate queries, where the aggregate function gives a single deterministic value for each group.

We thus identify two main challenges related to formulating and processing probabilistic ranking and aggregation queries:

—*Meaningful query semantics.* The proper query semantics of probabilistic ranking and aggregation queries need to integrate the semantics of probabilistic queries with the semantics of conventional (score-based) top-k queries. Different possible query semantics arise from this integration.
—*Efficient query processing.* While minimizing the number of accessed tuples/groups is central to most score-based top-k techniques, uncertainty adds further processing complexity, making existing score-based methods inapplicable. Specifically, uncertainty generates a huge space of possible query answers that have to be explored and ranked to find the high-quality ones. In such settings, integrating tuple retrieval, ranking, and uncertainty management, within the same framework, is essential for efficient processing.

We tackle the aforementioned challenges under a general uncertainty model allowing for different forms of uncertainty and probabilistic correlations. Our techniques are based on two basic ideas: (1) identifying the necessary tuple order that can be exploited for efficient query evaluation; and (2) applying probability-guided search mechanisms to evaluate our queries, while avoiding the materialization of every possible answer.

1.2 Contributions and Outline

Our general approach is to process scores and manage uncertainty in one framework leveraging current DBMS storage, query processing capabilities, and probabilistic inference models. Our main contributions towards this goal are summarized as follows:

—*New query definitions.* We are the first to propose probabilistic formulations of top-k and top-k aggregate queries that integrate both score and probability.
—*Search space model.* We model probabilistic top-k and top-k aggregate queries as a *space search* problem. We introduce several principled space navigation algorithms, with performance guarantees, to *lazily* and *partially* materialize the answer space while searching for top-ranked query answers.
—*Processing framework.* We construct a pipelined processing framework that allows for early query termination. Our proposed framework is the first integrated solution that treats tuple retrieval, grouping, aggregation, uncertainty management, and ranking in a pipelined fashion, allowing for feasible query evaluation.

Symbol	Description	Symbol	Description
$\mathcal{D}$	Probabilistic database	GLW^i	Global world i
n	No. of tuples used in query evaluation	$\mathcal{G}_{i,k}$	A prefix of length k in GLW^i
$\mathcal{F}$	Scoring (or group aggregate) function	s_g^i	State i of group g
s_l	A state of length l	G_l	Global state with length l
$\mathcal{P}(s_l)$	Probability of state s_l	S_l	Inter-groups state with length l
$P_{x,i}$	Pr(tuple (or group) x to be at rank i)	$\mathcal{R}^{\mathcal{F}}(s_g)$	Agg. range of group state s_g

Fig. 2. Frequently used notations.

—*Experimental study.* We conduct an extensive experimental study to evaluate our techniques based on a prototype implementation on top of PostgreSQL. The experiments evaluate our techniques in different configurations, showing the efficiency and robustness of our techniques in different practical settings.

Our earlier work [Soliman et al. 2007a] (described in Sections 5 and 6.1) studies top-*k* queries for probabilistic tuples with deterministic scores. The work in Soliman et al. [2007a] does not address ranking-aggregate queries, or top-*k* queries with probabilistic scores. In this article, we significantly extend Soliman et al. [2007a]. In addition, we study formulating and computing probabilistic ranking-aggregate queries and the applicability of our techniques to several other extensions and special cases.

Figure 2 gives the frequently used notations in this article. The remainder of this article is organized as follows. Section 2 gives necessary background on uncertain databases. Section 3 describes the uncertainty model we adopt in this article, and gives the problem definition. Section 4 gives the high-level view of our processing framework and assumptions. Section 5 describes the processing techniques for probabilistic top-*k* queries. Section 6 describes special cases resulting from relaxing our framework assumptions. Section 7 describes the formulation and processing of ranking-aggregate queries. Section 8 discusses other related query types. Section 9 discusses handling expensive probability computation. Section 10 is our experimental study. Section 11 describes related works. Finally, Section 12 concludes the article with final remarks.

2. BACKGROUND: MODELING AND QUERYING UNCERTAIN DATA

Several probabilistic data and query models have been proposed [Madden et al. 2003; Fuxman et al. 2005; Cheng et al. 2007; Sarma et al. 2006]. Most of these models assume two basic uncertainty types. The first type, usually referred to as “membership uncertainty” [Lakshmanan et al. 1997; Dalvi and Suciu 2007; Sarma et al. 2006] treats tuples as uncertain events capturing the belief that they belong to the database. The probabilities of such events originate from different sources, for example, reliability of data source in data integration environments [Hernandez and Stolfo 1998], or similarity measures in approximate-matching [Dalvi and Suciu 2007]. The second uncertainty type, referred to as “value uncertainty” [Cheng et al. 2007; Sarma et al. 2006] represents attributes as probability distributions on continuous or discrete domains

of possible values (e.g., modeling the readings of sensing devices). In the following, we mean by a "probabilistic database" a database with uncertain tuple membership.

Many proposed models [Imieliński and Witold Lipski 1984; Abiteboul et al. 1987; Sarma et al. 2006], adopt *possible worlds* semantics, where a probabilistic database $\mathcal{D}$ is viewed as a set of possible worlds $\{PW^1, \ldots, PW^n\}$ that represent an enumeration of all possible instances of the database. Each world is a subset of database tuples. The principles of probability theory are used to support this view by modeling tuples as probabilistic events. Specifically, a tuple t is associated with an event $t.e$ such that t exists in the database with probability $\Pr(t.e)$, and does not exist in the database with probability $\Pr(\neg t.e) = 1 - \Pr(t.e)$. Possible worlds are thus viewed as conjunctions of tuple events.

Let Q be a query to be executed over $\mathcal{D}$, where $Q(\mathcal{D})$ is the output of Q, and $Q(PW^i)$ is the output of Q restricted to PW^i. In possible worlds semantics, the probability of an output tuple $t_q \in Q(\mathcal{D})$ is computed as the summation of the probabilities of the possible worlds where t_q is reported as a query answer, as given in the following equation:

$$\Pr(t_q.e) = \sum_{PW^i : t_q \in Q(PW^i)} \Pr(PW^i). \tag{1}$$

A possible world is effectively a deterministic database, and hence query processing in individual worlds follows the operational semantics of conventional query operators. The huge number of possible worlds hinders instantiating and processing worlds explicitly. However, thinking in terms of possible worlds allows defining proper query semantics.

Possible worlds probabilities are determined based on the probabilistic correlations among tuples (e.g., mutual exclusion of tuples that map to the same real world entity [Sarma et al. 2006]). We call such correlations *generation rules*, since they control how the possible worlds space is generated. Such rules could naturally arise with unclean data [Andritsos et al. 2006], or could be enforced to satisfy application requirements or reflect domain semantics [Widom 2005; Sarma et al. 2006; Benjelloun et al. 2006]. Moreover, the relational processing of probabilistic tuples induces correlations among intermediate query results, even when base tuples are uncorrelated [Dalvi and Suciu 2007].

To illustrate, Figure 3(a) shows the `Readings` relation, from Example 1, augmented with *generation rules* that enforce the following requirement: "based on radar locations, the same car cannot be detected at two different locations within 1 hour interval." Figure 3(b) shows the possible worlds and their probabilities. Each world can be seen as a *joint* event of the *existence* of a world's tuples and the *absence* of all other database tuples. The probability of this joint event is determined by tuple probabilities and the generation rules that correlate them. The given xor rules in Figure 3(a) mean that in any possible world the existence of $t2$ *implies* the absence of $t3$, and, similarly, the existence of $t4$ *implies* the absence of $t5$. All other tuples are uncorrelated (*independent*). Consequently, $\Pr(PW^1) = \Pr(t1.e \wedge t2.e \wedge t6.e \wedge t4.e \wedge \neg t3.e \wedge \neg t5.e) = 0.4 \times 0.7 \times 1.0 \times 0.4 = 0.112$. The probabilities of other worlds are computed

	Time	Location	Make	PlateNo	Speed	Prob
t1	11:45	L1	Honda	X-123	130	0.4
t2	11:50	L1	Toyota	Y-245	120	0.7
t3	11:35	L2	Toyota	Y-245	80	0.3
t4	12:10	L3	Mazda	W-541	90	0.4
t5	12:25	L4	Mazda	W-541	110	0.6
t6	12:15	L4	Nissan	L-105	105	1.0

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

(a)

PW¹	PW²	PW³	PW⁴
t1	t1	t1	t1
t2	t2	t6	t5
t6	t5	t4	t6
t4	t6	t3	t3
0.112	0.168	0.048	0.072

PW⁵	PW⁶	PW⁷	PW⁸
t2	t2	t6	t5
t6	t5	t4	t6
t4	t6	t3	t3
0.168	0.252	0.072	0.108

(b)

Fig. 3. Probabilistic database: (a) probabilistic relation and generation rules; (b) possible worlds space.

similarly. Any possible world, other than $PW^1 \dots PW^8$, has zero probability based on tuple probabilities and generation rules.

We now discuss computing SPJ queries over probabilistic relations. The semantics of SPJ operators are overloaded to handle probability computation. Let R and S be two probabilistic relations containing the tuples r and s, respectively. Let σ^p, π^p, and $\bowtie^p$ be the probabilistic selection, projection, and join operators, respectively. The tuples produced by these operators are associated with the following events:

$$(\sigma_c^p(r)).e = \begin{cases} r.e & \text{if } c(r) = true \\ undefined & \text{if } c(r) = false \end{cases} \text{; where } c(r) \text{ is the selection condition.} \quad (2)$$

$$\left(\pi^p_{\{A_1 \dots A_n\}}(r)\right).e = \bigvee_{\acute{r} \in R : \pi_{\{A_1 \dots A_n\}}(\acute{r}) = \pi_{\{A_1 \dots A_n\}}(r)} \acute{r}.e \quad (3)$$

$$\left(r \bowtie^p_{c(r,s)} s\right).e = \begin{cases} r.e \wedge s.e & \text{if } c(r,s) = true \\ undefined & \text{if } c(r,s) = false \end{cases} \text{; where } c(r,s) \text{ is the join condition} \quad (4)$$

The probability of each query output tuple t_q is computed as $\Pr(t_q.e)$, which is equivalent to the probability computed under possible worlds semantics, as given in Eq. (1) [Dalvi and Suciu 2007]. Such computation is done using the formulation of $t_q.e$ (as given in Eqs. (2), (3), and (4)), and the correlations (rules) that bind the involved events in $t_q.e$.

Computing the probabilities of output tuples in SPJ queries is generally feasible under bag semantics. However, under set semantics, probability computation is hard in some cases even if the base tuple events are independent [Dalvi and Suciu 2007]. In particular, computing the probabilities of the output tuples of the π^p operator is as hard as computing the satisfiability ratio of a DNF formula, which is #P-Complete [Valiant 1979]. Other proposals [Sen and Deshpande 2007], use factored representation of tuples dependencies to compute the probabilities of output tuples using probabilistic inference. The computational cost is reduced using variable elimination and factor decomposability, however the problem remains generally intractable.

3. UNCERTAINTY MODEL AND PROBLEM DEfiNITION

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

3.1 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 between the uncertainty model, and our processing framework (we elaborate on this point in Section 4). This separation of model details and query 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 to our requirements, with different specifications and implementations:

—*Independent tuples.* When all tuples events are independent, as in Figure 1, the model is simply the membership probabilities of base tuples. Using such a simple model, the joint probability of any combination of tuple events is computed by multiplying the probabilities of the corresponding tuple events.

—*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., Figure 3), 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 $\Pr(t1.e \wedge \neg t2.e) = 0$. Similar types of rules have been used in different uncertainty models [Sarma et al. 2006; Sen and Deshpande 2007], to represent tuples' correlations. Note that tuples' probabilities have to be *consistent* with rules semantics, otherwise the possible worlds semantics would be violated. For example, if the rule $(t1 \oplus t2)$ holds, then $\Pr(t1.e) + \Pr(t2.e)$ must be ≤ 1. Similarly, if the rule $(t1 \rightarrow t2)$ holds, then $\Pr(t1.e)$ must be $\leq \Pr(t2.e)$. 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. 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.

—*General inference model.* The two models above 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 distribution is factor graphs, proposed in Sen and Deshpande [2007], where arbitrary tuple correlations are maintained in the form of conditional probability tables.

We discuss the implementation details of our adopted uncertainty model in Section 4. However, we emphasize that model specifications, expressiveness, and implementation are not the main focus of our study.

3.2 Problem Definition

We assume the query model given in Eq. (1), which is applicable to arbitrary uncertainty models that treat tuples as probabilistic events (e.g., the models described in Section 3.1). Our goal is to compute query answers by conceptually computing query answers in each world and aggregating the probabilities of identical answers. In the following, we assume that $\mathcal{D}$ is a probabilistic database conforming to our uncertainty model, Q is a query to be processed on $\mathcal{D}$, and $\mathcal{F}$ is a scoring (ranking) function that ranks output tuples in $Q(\mathcal{D})$ (the convention adopted throughout this article is that we rank in the descending order of $\mathcal{F}$ values). A concrete example of Q is the probabilistic SPJ query described in Section 2. We assume $\mathcal{PW} = \{PW^1, \dots, PW^n\}$ is the set of possible worlds of $Q(\mathcal{D})$.

Definition 1. *Uncertain Top-k Query* (*U-Topk*). Let $\mathcal{T} = \{T^1, \dots, T^m\}$ be the set of all k-length tuple vectors in $Q(\mathcal{D})$ such that each $T^i \in \mathcal{T}$ is the top-k answer in a nonempty set of possible worlds $W(T^i) \subseteq \mathcal{PW}$ based on $\mathcal{F}$. We define the probability of a top-k vector $T^j \in \mathcal{T}$ as $\Pr(T^j) = \sum_{PW^i \in W(T^j)} \Pr(PW^i)$. A U-Top$k$ query based on $\mathcal{F}$, returns a k-length tuple vector $T^* \in \mathcal{T}$, with the highest probability among all tuple vectors in $\mathcal{T}$.

U-Top$k$ query returns the tuple vector with the highest probability of being top-k across all worlds. For example, consider Query 1 in Figure 1. Figure 3(b) shows the possible worlds of the `Readings` relation ordered on *Speed*. A U-Top2 query answer is the tuple vector $\langle t1, t2 \rangle$ with probability 0.28, since $\langle t1, t2 \rangle$ is the top-2 vector in PW^1 and PW^2 whose probability summation is 0.28, which is the maximum probability among all possible top-2 vectors. Tuple vectors reported by U-Top$k$ query are valid tuple combinations that conform to tuple dependencies (e.g., reported vectors cannot contain mutually-exclusive tuples).

Definition 2. *Uncertain k Ranks Query* (*U-kRanks*). For $i = 1 \dots k$, let $X_i = \{t_i^1, \dots, t_i^m\}$ be a set of tuples in $Q(\mathcal{D})$, where each tuple t_i^j appears at rank i in a nonempty set of possible worlds $W(t_i^j) \subseteq \mathcal{PW}$ based on $\mathcal{F}$. We define the probability of a tuple $t_i^j \in X_i$ to be at rank i as $\Pr(t_i^j) = \sum_{PW^l \in W(t_i^j)} \Pr(PW^l)$. A U-$k$Ranks query based on $\mathcal{F}$, returns a set $X^* = \{t_1^*, \dots, t_k^*\}$, where each tuple t_i^* has the highest probability among all tuples in X_i.

U-kRanks query answer is a set of tuples that together might not form the most probable top-k vector. However, each tuple is a clear winner at its rank over all worlds. Consider Figure 3. A U-2Ranks query answer is $\{t2 : 0.42, t6 : 0.324\}$, since $t2$ is at rank 1 in PW^5 and PW^6 with probability summation

0.42, while $t6$ is at rank 2 in PW^3, PW^5, and PW^8 with probability summation 0.324. Note that the set of worlds contributing to the probability of $t2$ being at rank 1 are considered independently from the set of worlds contributing to $t6$ being at rank 2. Hence, in contrast to U-Top$k$ query, a U-kRanks query relaxes the restriction of having dependency-compliant tuple combinations in a query answer.

Definition 2 does not restrict query answers at different ranks to be distinct. That is, a tuple can be the most probable tuple to appear at different ranks simultaneously. Since U-kRanks query concentrates on each rank independently of the others (e.g., finding the most probable athlete to win the silver medal), the query definition needs to be extended to address distinct query answers. We discuss such extensions in Section 8

U-Top$k$ and U-kRanks queries can be useful in a wide range of practical applications. For example, in Figure 3, U-Top$k$ may be useful in discovering groups of over-speeding cars in an accident investigation (e.g., the k cars that are consistently over-speeding in different possible worlds). On the other hand, U-kRanks may be useful in assessing insurance policies based on the makes of the most speeding cars over a time period, considered independently from each other. Other possible semantics of probabilistic top-k queries include finding the most probable top-k set, that is, the k tuples with the highest probabilities of appearing in the top-k of different worlds. We show in Section 8 that our query definitions can also cover these semantics.

We extend the above definitions to handle probabilistic top-k aggregate queries. In the following, let Q be a group-by query, $\mathcal{A}$ be a set of grouping attributes, and $\mathcal{F}$ be a group aggregate function, for example, *sum*, that ranks groups in $Q(\mathcal{D})$. The distinct combinations of the attributes' values in $\mathcal{A}$ act as *group identifiers*.

Definition 3. *Uncertain Top-k Aggregate Query* (*U-Topk-Agg*). Let $\mathcal{G}_i = \langle g_1, \ldots, g_k \rangle$ be the top-k-group vector in a nonempty set of possible worlds $W(\mathcal{G}_i) \subseteq \mathcal{PW}$, based on $\mathcal{A}$ and $\mathcal{F}$. We define the probability of a top-k-group vector $\mathcal{G}_i$ as $\Pr(\mathcal{G}_i) = \sum_{PW^l \in W(\mathcal{G}_i)} \Pr(PW^l)$. A U-Top$k$-Agg query, based on $\mathcal{A}$ and $\mathcal{F}$, returns a k-length vector $\mathcal{G}^*$ with the highest probability.

Definition 4. *Uncertain k Ranks Aggregate Query* (*U-kRanks-Agg*). For $i = 1 \ldots k$, let $X_i = \{g_i^1, \ldots, g_i^m\}$ be a set of group identifiers in $Q(\mathcal{D})$, where each group identifier g_i^j appears at rank i in a nonempty set of possible worlds $W(g_i^j) \subseteq \mathcal{PW}$ based on $\mathcal{A}$ and $\mathcal{F}$. We define the probability of group $g_i^j \in X_i$ to be at rank i as $\Pr(g_i^j) = \sum_{PW^l \in W(g_i^j)} \Pr(PW^l)$. A U-$k$Ranks-Agg query, based on $\mathcal{A}$ and $\mathcal{F}$, returns a set $X^* = \{g_1^*, \ldots, g_k^*\}$, where each group identifier g_i^* has the highest probability among all group identifiers in X_i.

A straightforward procedure to compute U-Top$k$-Agg, and U-kRanks-Agg queries is to materialize $\mathcal{PW}$, group each world based on the grouping attributes $\mathcal{A}$, apply aggregation function $\mathcal{F}$ to each group in each world, sort the results obtained from each world based on $\mathcal{F}$, and finally add up the probabilities of the worlds with the same top-k group vectors/group identifiers at different ranks,

	Time	Location	Make	PlateNo	Speed	Prob
t1	11:45	L1	Honda	X-123	130	0.4
t2	11:50	L1	Toyota	Y-245	120	0.7
t3	11:35	L2	Toyota	Y-245	80	0.3
t4	12:10	L3	Mazda	W-541	90	0.4
t5	12:25	L4	Mazda	W-541	110	0.6
t6	12:15	L4	Nissan	L-105	105	1.0

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

(a)

PW1	PW2	PW3	PW4
t1	t1	t1	t1
t2	t2	t6	t5
t6	t5	t4	t6
t4	t6	t3	t3
0.112	0.168	0.048	0.072

PW5	PW6	PW7	PW8
t2	t2	t6	t5
t6	t5	t4	t6
t4	t6	t3	t3
0.168	0.252	0.072	0.108

(b)

PW1	PW2	PW3	PW4	PW5	PW6	PW7	PW8
L1:125	L1:125	L1:130	L1:130	L1:120	L1:120	L4:105	L4:107.5
L4:105	L4:107.5	L4:105	L4:107.5	L4:105	L4:107.5	L3:90	L2:80
L3:90		L3:90	L2:80	L3:90		L2:80	
		L2:80					

(c)

Fig. 4. (a) Probabilistic relation and generation rules; (b) possible worlds space grouped on "Location"; (c) groups ranked on "Average(Speed)."

and report the answers with the maximum probability. This procedure is clearly impractical because of the huge space of possible worlds, and the potential complexity of probability computation.

We illustrate the semantics of our ranking-aggregates queries based on Query 2, described in Figure 1, where $\mathcal{A} = \{Location\}$ and $\mathcal{F} = Average(Speed)$. In Figure 4(b), tuples belonging to the same group in each world are enclosed in the same dotted box. Figure 4(c) shows the grouped and ranked possible worlds. A U-Top2-Agg query answer is the group vector $\langle L1, L4 \rangle$ with probability 0.82, while a U-3Ranks-Agg query answer is the group set $\{L1 : 0.82, L4 : 0.82, L3 : 0.329\}$.

In each of the previous queries, we report the most probable answer only. However, the query definitions can be generalized to formulate query answers as the l most probable answers for some input parameter l. We discuss this generalization in Section 8.

4. PROCESSING FRAMEWORK

In this section we describe our query processing framework to support top-*k* and top-*k* aggregate queries in probabilistic databases. The framework is based on two underlying design principles:

—**DP1**: To build on top of an RDBMS as our tuple access layer. We use an underlying RDBMS to store and process probabilistic data and uncertainty information. Our processing framework leverages RDBMS storage, indexing and query processing capabilities to compute probabilistic top-*k* (aggregate) queries. Similar arguments are made in the design of the TRIO system [Widom 2005; Benjelloun et al. 2006].

—**DP2**: To leverage current top-*k* and top-*k* aggregates query processing techniques in deterministic databases. In particular, our framework takes

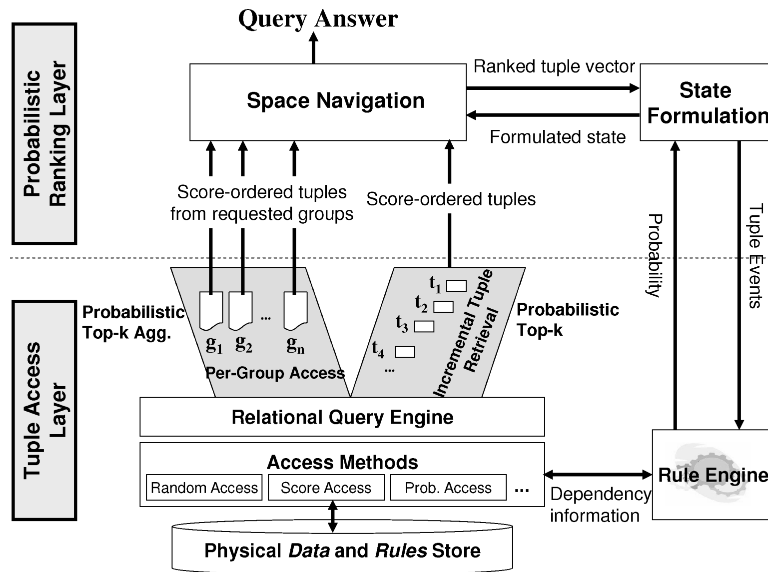

Fig. 5. Processing framework.

advantage of rank-aware/group-aware query processing (if supported by the underlying DBMS) to minimize the number of needed-to-access tuples/groups. The framework also adopts the upper-bounding principle used in several proposals [Ilyas et al. 2003; Li et al. 2006] to limit the size of the materialized space.

4.1 Framework Details

In this section we give a detailed description of our processing framework. Figure 5 shows the framework architecture.

Tuple access layer. Tuple retrieval, indexing, and query processing (including score-based ranking) are the main functionalities of this layer. Additionally, to support top-k aggregate queries, the *tuple access layer* provides an interface to allow the upper layer to retrieve tuples incrementally from specific groups. Different proposed techniques(e.g., index striding [Hellerstein et al. 1997]), can satisfy such requirement. We elaborate on these techniques in Section 11. The *tuple access layer* executes an incoming query, which acts as the tuple source of the upper layer. We show in Section 5.1 that sorted score access for output tuples from the tuple access layer is necessary for efficient processing.

While our techniques can benefit largely from efficient support of ranking and grouping in the *tuple access layer*, our framework is still valid if such support is limited or lacking. However in this case, extra tuples/groups need to be accessed to realize query answers. For example, a complete sorting of Boolean 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* are not the focus of our study, since they vary according to how sophisticated the underlying uncertainty model is, as discussed in Section 3.1. To illustrate, Dalvi and Suciu [2007] show how to generate *safe* plans for some class of SPJ queries, where the independence of tuple events is exploited to efficiently compute the probability of query output tuples. A simple rule engine that maintains the membership probabilities of base tuples can be sufficient in this case. Alternatively, Sen and Deshpande [2007] use factored representation of the conditional probability distribution of dependant tuples, allowing for encoding arbitrary dependencies. A much more sophisticated rule engine needs to be built in this case to compute the probabilities of arbitrary combinations of tuple events. Hence, treating the rule engine as a black box adds versatility to our framework, and does not restrict our techniques to a specific implementation of the underlying uncertainty model.

In our prototype [Soliman et al. 2007b], we experimented with three different implementations of the rule engine module: (1) a simple engine that supports probability computation over independent tuple events; (2) an engine compliant with the x-tuple model [Sarma et al. 2006; Benjelloun et al. 2006], where tuples are correlated with exclusiveness rules only; and (3) a more complex engine that implements and indexes a Bayesian network that is used during query processing to load relevant dependency information on demand and compute the probabilities of tuple combinations through Bayesian inference techniques. We do not discuss the details of these implementations in this article, rather, we abstract such details using our interface to the rule engine module.

Probabilistic ranking layer. This layer retrieves tuples from the *tuple access layer*, and efficiently navigates the space of possible worlds 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 possible worlds space while searching for query answers. We give the formal definitions of the problem space for probabilistic top-k queries, and probabilistic top-k aggregate queries in Sections 5.1, and 7.1, respectively.

4.2 Assumptions

We make the following assumptions in our framework design. We relax some of these assumptions in Sections 6 and 9 .

Tuple access: We assume tuples are consumed incrementally, that is, one by one, from the output of a query executed by the relational engine in the tuple access layer. That is, the probabilistic ranking layer does not have random access to some tuple t unless produced by the tuple access layer. This assumption is generally reasonable since random access to arbitrary query output tuples is usually not available, unless query output is fully computed. Full evaluation of

top-k queries should be avoided if possible, since only a small fraction of query output tuples suffices to get the query answer if tuples are consumed in the right order. In our framework, the relational engine incrementally computes the tuples of query answer and pipelines these tuples to the probabilistic ranking layer upon request through an *iterator* interface, which is widely used in RDBMSs. We relax the tuple access assumption in Section 6.2 by considering special cases that allow random access to the underlying database.

Group access: For top-k aggregate queries, we assume an interface that allows accessing tuples from specific groups incrementally. Group-by operators are usually blocking operators that see most of the data before producing tuple groups. The per-group tuple access can be enabled using the techniques described in Hellerstein et al. [1997]. For example, if the grouping operator hashes the input tuples on the grouping attributes, it can provide tuples from the requested groups incrementally to the probabilistic ranking layer. The index striding method, described in Section 11, realizes per-group tuple access efficiently. We use this method in the implementation of our framework.

Available dependency information: We assume the dependencies among query output tuples are only known when these tuples are consumed by the probabilistic ranking layer. That is, we do not know if the currently consumed tuples are correlated with other future tuples until these future tuples are actually consumed by the probabilistic ranking layer. This assumption is justified by the incremental tuple computation in the relational engine of the tuple access layer. Similarly, dependency information is built incrementally as tuples are produced and pipelined to the probabilistic ranking layer. Hence, no dependency information relating tuples currently consumed with other future tuples is available. We note, however, that in some cases complete dependency information can be directly available. For example, a query that involves a single relation with independent tuples is known to generate no dependencies among query output tuples. We address this case in Section 6.1, where we relax our assumption by exploiting more available information on tuple dependencies.

Group cardinality information: For ranking-aggregate queries, we assume the *tuple access layer* provides (bounds on) the size of each group. This information can be available from multiple sources including database catalog, prematerialized datacubes in OLAP environments, or by simply running a count query on each group. Previous studies [Li et al. 2006], have shown that limited optimization chances of top-k aggregate queries in deterministic databases are available if no group cardinality information is known. We discuss related works that can be adapted to obtain cardinality bounds in Section 11. Obtaining tight group cardinality bounds allows our framework to tightly bound possible group aggregates, leading to early query termination. If only loose cardinality bounds are available, our framework can still compute query answers—however, a larger number of tuples may be consumed, and hence a larger answer space needs to be explored. We elaborate on these points in Section 7.2.1.

Event probability computation: We assume that the rule engine responds with *exact* probabilities to the submitted questions (joint probabilities of tuple

combinations). However, since exact probability computation can be expensive for some query types/plans, particularly projections under set semantics [Dalvi and Suciu 2007], we discuss in Section 9 relaxing this assumption by dealing with approximate output from the rule engine in the form intervals enclosing the exact probability value.

5. PROBABILISTIC TOP-k QUERIES

We now discuss computing probabilistic top-k queries. The main idea of our approach is to model top-k query as a search problem over the space of all possible query answers (Sections 5.1 and 5.2). In order to navigate such space efficiently, we identify the tuple retrieval order that can be used to construct the space incrementally (Section 5.1). Based on the identified order, we design $\mathcal{A}^*$-like search mechanisms to compute query answers from partial space materialization by correctly bounding the probability of different search paths in the space. The search terminates early by finding an answer whose probability is not below the probability upper-bound of all other unexplored search paths (Section 5.3). We show that our algorithms minimize the number of consumed tuples and the size of the materialized space to evaluate probabilistic top-k queries (Section 5.4).

5.1 Problem Space

We start our space formulation by defining the search state:

Definition 5. *Top-l State.* A top-l state s_l is a *prefix* of *length* l of some possible world(s) of $\mathcal{D}$ that are ordered on a scoring function $\mathcal{F}$.

A top-l state s_l is *complete* if and only if $l = k$. Based on possible worlds semantics, the probability of state s_l is equal to the summation of the probabilities of all worlds having s_l as a prefix, that is, a top-l answer. Our search for query answers starts from an empty state (with *length* 0) and ends at a goal *complete* state with the maximum probability.

We next formulate state probability. We use the notation $\neg X$, where X is a tuple set/vector, to refer to the conjunction of the negation of tuple events in X. Let $\underline{\mathcal{F}}(s_l)$ be the minimum tuple score in s_l. Let I_{s_l} be the set of tuples not in s_l but have higher scores than $\underline{\mathcal{F}}(s_l)$, i.e., $I_{s_l} = \{t | t \in \mathcal{D}, t \notin s_l, \mathcal{F}(t) > \underline{\mathcal{F}}(s_l)\}$. The probability of state s_l, denoted $\mathcal{P}(s_l)$, is equal to the joint probability of the existence of tuples in s_l and the absence of tuples in I_{s_l}, that is, $\mathcal{P}(s_l) = \Pr(s_l \wedge \neg I_{s_l})$, which is the probability that s_l is the top-l query answer. For example in Figure 3, for a state $s_2 = \langle t1, t5 \rangle$, we have $\mathcal{P}(s_2) = \Pr((t1.e \wedge t5.e) \wedge (\neg t2.e)) = 0.072$. This result can be verified from PW^4. In general, state probability is computed by the rule engine module in our framework based on tuples' probabilities and their dependencies (Section 4).

Due to the overwhelming number of possible states, an efficient search mechanism needs to avoid visiting the states that do not lead to query answers, as we discuss next.

Tuple retrieval order. Based on our tuple access, and available dependency information assumptions (Section 4.2), we show that retrieving tuples in sorted

score order is *necessary* and *sufficient* to get nontrivial bounds on the probabilities of possible complete states. Such bounding is crucial for early query termination, that is, termination without checking every possible query answer. The next examples illustrate why tuple orders, different from sorted score order, fail in bounding state probabilities under our assumptions.

Example 2. *Arbitrary Order*. Consider Figure 3. Assume that we retrieved $t1$ and $t6$ from the tuple access layer, based on a random tuple retrieval order. Let $s_2 = \langle t1, t6 \rangle$ be a state in our search space (recall that a state is a score-ordered prefix of one or more worlds). At this point, $\mathcal{P}(s_2)$ cannot be computed precisely, since we are unaware of I_{s_2} (all other tuples with scores higher than the minimum score in s_2). Moreover, we cannot lower-bound $\mathcal{P}(s_2)$ by a value greater than 0, since I_{s_2} might contain a tuple t independent with s_2 tuples, and has a probability of 1, which would make $\mathcal{P}(s_2) = 0$. Clearly, we cannot conclude the search goal early, that is, without fully inspecting all space states, if we cannot compute nontrivial (greater than 0) lower-bounds on states' probabilities.

Example 3. *Probability Order*. Consider Figure 3. Assume that we retrieved $t6$ and $t2$ from the tuple access layer, based on tuple probability order. Let $s_2 = \langle t2, t6 \rangle$ be a state in our search space. Here, we have more information than Example 2, since we know that no tuple in I_{s_2} has a probability greater than 0.7 (the probability of $t2$). However, we still cannot compute a nontrivial lower-bound for $\mathcal{P}(s_2)$. The reason is that the set I_{s_2}, which is not completely known at this point, may contain a tuple t correlated with s_2 such that $\Pr(t2.e \wedge t6.e \wedge \neg t.e) = 0$, and hence $\mathcal{P}(s_2) = 0$. Based on our *available dependency information* assumption, we do not know whether such correlation exists or not, since t is not yet retrieved from the tuple access layer.

Example 4. *Score Order*. Consider Figure 3. Assume that we retrieved $t1$ and $t2$ from the tuple access layer, based on tuple score order (the *speed* attribute). Let $s_2 = \langle t1, t2 \rangle$ be a state in our search space. Since all nonretrieved tuples have scores less than $t2$, the set I_{s_2} is known to be empty, and we can precisely compute $\mathcal{P}(s_2) = 0.4 \times 0.7 = 0.28$. Hence, retrieving tuples in score order gives perfect information on each state s_l and its corresponding I_{s_l} set, allowing for computing precise state probabilities.

Based on the above examples, we conclude that under the assumptions discussed in Section 4.2, retrieval orders different from score order do not effectively bound state probabilities, necessitating fully materializing the state space to locate query answer. Hence, we adopt score order in retrieving tuples from the tuple access layer. Rank-aware query processing techniques can be used in the tuple access layer to incrementally produce tuples in score order using score indexes and rank-aware query operators [Ilyas et al. 2003; Li et al. 2005]. Theorem 1 formally proves the superiority of sorted score retrieval.

THEOREM 1. *Under the* tuple access*, and* available dependency information *assumptions, retrieving query output tuples in score order is* (1) *sufficient; and* (2) *necessary to compute nontrivial bounds on state probabilities.*

PROOF. Let X_{score} be a subset of query output tuples retrieved in score order. For any state s_l whose tuples form an arbitrary subset of X_{score}, we have $I_{s_l} \subseteq X_{score}$. Hence, X_{score} is *sufficient* to precisely compute $\mathcal{P}(s_l)$, and thus (1) follows.

We prove (2) by showing that tuple orders, different from sorted score order, fail to provide nontrivial bounds on state probabilities. We divide such orders into two groups:

(i) *Nonprobability orders*: Let $X_{arbitrary}$ be a subset of query output tuples retrieved in an arbitrary order following neither score nor probability. For any state s_l whose tuples are an arbitrary subset of $X_{arbitrary}$, assume a yet nonretrieved tuple $t \notin X_{arbitrary}$, such that $t \in I_{s_l}$, t has probability 1, and t is independent from tuples in s_l. Then, $\mathcal{P}(s_l) = 0$. Hence, based on $X_{arbitrary}$, the only *safe* lower-bound on $\mathcal{P}(s_l)$ is 0.

(ii) *Probability order*: Let X_{prob} be subset of query output tuples retrieved in probability order. For any state s_l whose tuples form an arbitrary subset of X_{prob}, assume a yet nonretrieved tuple $t \notin X_{prob}$, such that $t \in I_{s_l}$, and $\Pr(s_l \wedge \neg t) = 0$. Then, $\mathcal{P}(s_l) = 0$. Hence, based on X_{prob}, the only *safe* lower-bound on $\mathcal{P}(s_l)$ is 0.

Based on (i) and (ii), we conclude that (2) is correct. □

Relaxing the assumptions stated in Theorem 1 makes other tuple retrieval orders useful in evaluating our queries. We discuss such orders in Section 6.2.

5.2 Generating the Search Space

In this section we show how to use score-ordered tuple retrieval to build the search space.

A top-l state is a combination of tuple events. In possible worlds semantics, the probability of any combination of tuple events is the summation of the worlds' probabilities where this combination is satisfied. For example in Figure 3, the probability of the tuple event combination $(t1.e \wedge \neg t2.e)$ is the same as $\Pr(PW^3) + \Pr(PW^4) = 0.12$. We next explain an important property of space states.

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 follows from set theory, where a set cannot be larger than its intersection with another set. This holds in our uncertainty 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 $\subseteq$ the set of possible worlds where E_n is satisfied.

Search states are generated as follows. Assume a current state s_l. After retrieving a new tuple t from the *Tuple Access Layer*, we extend s_l into two states: (1) $\acute{s}_l$: a state with the same tuple vector as s_l, where we define $I_{\acute{s}_l} = I_{s_l} \cup \{t\}$; and (2) s_{l+1}: a state composed of the tuple vector of s_l appended by t, where

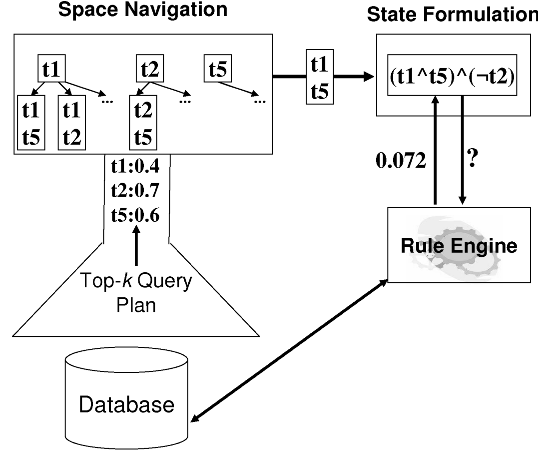

Fig. 6. Interaction of framework components.

$I_{s_{l+1}} = I_{s_l}$. For example, assume a state $s_2 = \langle t1, t2 \rangle$, where $I_{s_2} = \{t3\}$. Hence, $\mathcal{P}(s_2) = \Pr((t1.e \wedge t2.e) \wedge (\neg t3.e))$. Upon retrieving $t4$, the next tuple in score order, we extend s_2 into (1) $\acute{s}_2 = \langle t1, t2 \rangle$, where $I_{\acute{s}_2} = \{t3, t4\}$, and hence $\mathcal{P}(\acute{s}_2) = \Pr((t1.e \wedge t2.e) \wedge (\neg t3.e \wedge \neg t4.e))$; and (2) $s_3 = \langle t1, t2, t4 \rangle$, where $I_{s_3} = \{t3\}$, and hence $\mathcal{P}(s_3) = \Pr((t1.e \wedge t2.e \wedge t4.e) \wedge (\neg t3.e))$. Based on Property 1, both $\mathcal{P}(\acute{s}_l)$ and $\mathcal{P}(s_{l+1})$ cannot exceed $\mathcal{P}(s_l)$. In addition, $\mathcal{P}(\acute{s}_l) + \mathcal{P}(s_{l+1}) = \mathcal{P}(s_l)$.

We illustrate state generation using the interaction of our framework components depicted by Figure 6, which describes U-Top$k$ processing of Query 1 in Figure 1. Three tuples are produced by a (score-based) top-k query plan, running in the tuple access layer, and submitted to the space navigation module, which generates possible states based on the three *seen* tuples. Each state is extended by newly retrieved tuples to create new candidate top-l states. In order to compute the probability of each state, the state formulation module contacts the rule engine. For example, for state $s_2 = \langle t1, t5 \rangle$ with $I_{s_2} = \{t2\}$, the state formulation module formulates the event combination $((t1.e \wedge t5.e) \wedge (\neg t2.e))$, and requests its probability from the rule engine, which responds back with the value 0.072. In Section 5.3, we give efficient search algorithms that partially materialize the space by retrieving the least number of tuples and creating the least number of states.

Cost Metric. Based on our problem definition in Section 3.2, the number of possible top-k answers that can be obtained from n retrieved tuples is bounded by $\binom{n}{k}$, which is in $O(n^k/k!)$. Since k is a query parameter, 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.

5.3 Navigating the Search Space

In this section we give the details of our probability-guided search algorithms for U-Top$k$ (Section 5.3.1) and U-kRanks (Section 5.3.2) queries.

5.3.1 *Processing U-Topk Queries.* We describe Algorithm OPTU-Top$k$, our processing algorithm for U-Top$k$ queries. The details of OPTU-Top$k$ are given in Algorithm 1. The general idea is to buffer retrieved score-ordered tuples and adopt a *lazy* space materialization scheme to extend the state space, that is, a state might not be extended by all retrieved tuples. At each step, the algorithm extends only the state with the highest probability. State extension is performed using the next tuple drawn either from the buffer or from the underlying tuple access layer. The algorithm terminates when it reaches a complete state with the highest probability among all possible states.

Algorithm 1. OptU-Top$k$

U-TOPK (*source* : Score-ordered tuple stream, k : result size)
1 $\mathcal{Q} \leftarrow$ empty priority queue for states ordered on probabilities
2 $d \leftarrow 0$
3 Insert $s_{0,0}$ into $\mathcal{Q}$ {*initialize* $\mathcal{Q}$ *with an empty state*}
4 **while** (*source* is not exhausted AND $\mathcal{Q}$ is not empty)
5   **do**
6       $s_{l,i} \leftarrow$ dequeue ($\mathcal{Q}$) {*get the state with the highest probability*}
7   **if** ($l = k$)
8   **then return** $s_{l,i}$ {*highest probability state is complete, then terminate*}
9   **else**
10   **if** ($i = d$)
11   **then** {*need to retrieve a new tuple*}
12                   $t \leftarrow$ next tuple from *source*
13                   $d \leftarrow d + 1$
14   **else** {*can use an already retrieved tuple*}
15                   $t \leftarrow$ tuple at pos $i + 1$ in seen tuples buffer
16   Extend $s_{l,i}$ using t into $s_{l,i+1}$, $s_{l+1,i+1}$ {*See Section 5.2*}
17   Insert $s_{l,i+1}$, $s_{l+1,i+1}$ into $\mathcal{Q}$
18   **if** ($l + 1 = k$)
19   **then** {*a complete state is reached, prune inferior states*}
20   remove any state $s \in \mathcal{Q}$ where $\mathcal{P}(s) < \mathcal{P}(s_{l+1,i+1})$

We now discuss the details of Algorithm 1. We overload state definition s_l to be $s_{l,i}$, where i is the position of the last seen tuple by $s_{l,i}$ in the score-ordered tuple stream. Note that i can point to a buffered tuple or to the next tuple to be retrieved from the tuple access layer. We define $s_{0,0}$ as an initial *empty state* of *length* 0, where $\mathcal{P}(s_{0,0}) = 1$. The probability of the empty state upper-bounds the probability of any nonmaterialized state, since any nonmaterialized state is an extension of the empty state (cf., Property 1).

Let $\mathcal{Q}$ be a priority queue of states based on probability, where ties are broken using deterministic tie-breaking rules. We initialize $\mathcal{Q}$ with $s_{0,0}$. Let d be the number of retrieved tuples. OPTU-Top$k$ iteratively retrieves the top state in $\mathcal{Q}$, say $s_{l,i}$, extends it into the two next possible states (Section 5.2), and inserts the resulting two states back to $\mathcal{Q}$ according to their probabilities. Extending $s_{l,i}$ leads to consuming a new tuple from the tuple access layer

only if $i = d$, otherwise $s_{l,i}$ is extended using the buffered tuple pointed to by $i + 1$.

OPTU-Top$k$ terminates 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 nonmaterialized states (which are upper-bounded by the empty state) 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.

In addition to extending the space *lazily*, that is, only the top $\mathcal{Q}$ state is extended at each step, Algorithm 1 also applies a *pruning criterion* to significantly cut down the size of $\mathcal{Q}$ (line 18): If a complete state $s_{k,n}$ is reached, then all states in $\mathcal{Q}$ with probabilities less than $\mathcal{P}(s_{k,n})$ can be safely pruned, based on Property 1.

In Section 5.4, we analyze the complexity and performance guarantees of Algorithm OPTU-Top$k$.

5.3.2 *Processing U-kRanks Queries.* We describe OPTU-kRanks, our query processing algorithm for U-kRanks queries. Let t be the nth tuple in score-ordered tuple stream. Let $P_{t,i}$ be the probability that tuple t appears at rank i. 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. In other words, we can compute $P_{t,i}$, for $i = 1 \dots n$ as soon as we retrieve t from the tuple access layer. Algorithm OPTU-kRanks uses this observation by extending all maintained states on retrieving each new tuple t, causing all possible ranks of t to be identified. An upper-bound is maintained for the probability of an unseen tuple being at rank $i = 1 \dots k$. The algorithm reports an answer t^* at rank i when $P_{t^*,i}$ is greater than both the probability of any retrieved tuples being at rank i and the probability upper-bound of any nonretrieved tuple being at rank i.

Algorithm 2 describes the details of OPTU-kRanks. For each rank i, the algorithm remembers only the most probable answer obtained so far. This is because an unseen tuple u cannot change $P_{t,i}$ of a seen tuple t, since u can never appear before t in any possible world. In order to conclude an answer for rank i, the algorithm upper-bounds the probability of any unseen tuple to be at rank i as follows. Let ω_j be the current set of states with length j, and let $Z_j = \sum_{s_j \in \omega_j} \mathcal{P}(s_j)$. For any rank i, the value of $\sum_{j<i} Z_j$ can never increase when new tuples are consumed. Therefore, the maximum probability of an unseen tuple u being at rank i is $\sum_{j<i} Z_j$ (we formally prove this bound in Theorem 3). Let t^* be the current U-kRanks answer for rank i. The termination condition of Algorithm OPTU-kRanks, for rank i, is thus $P_{t^*,i} \geq \sum_{j<i} Z_j$.

Algorithm 2. OptU-kRanks

U-kRanks (*source* : Score-ordered tuple stream, k : Result size)
1 Initialize $\{answer_1 \dots answer_k\}$ as $\{null, \dots, null\}$
2 Initialize $(ubound_1, \dots, ubound_k)$ as $(1, \dots, 1)$ {*bounds of unseen tuples*}
3 *reported* $\leftarrow 0$
4 *depth* $\leftarrow 1$

```
5 space ← φ  {current set of materialized states }
6 {while ( source is not exhausted AND reported < k)
7        do
8          t ← next tuple from source
9          Extend all states in space based on t
10         for (i=1 to min(k, depth))
11            do
12                if(answer_i is previously reported)
13                   then Continue
14                Set ubound_i ← Σ_{j<i} Z_j based on space  {see Section 5.3.2}
15                Compute P_{t,i}
16                if (  (answer_i is null) OR
                      (answer_i is not null AND P_{t,i} > answer_i.prob)  )
17                   then  {found a better answer at rank i}
18                         answer_i ← t
19                         answer_i.prob ← P_{t,i}
20                         if (answer_i.prob ≥ ubound_i)
21                            then {termination condition reached at rank i}
22                            Report answer_i
23                            reported ← reported + 1
24            depth ← depth + 1
```

We analyze the complexity and performance guarantees of Algorithm OPTU-kRanks in Section 5.4.

5.4 Complexity of Search Algorithms

In this section we give optimality proofs and complexity analysis for our algorithms. We start by showing that Algorithm OptU-Top$k$ reduces to an instance of $\mathcal{A}^*$ search [Hart et al. 1968] over the space of possible top-k answers. Given an initial state s_0, $\mathcal{A}^*$ incrementally searches the space to find the path with the *least cost* to a goal state s_*. $\mathcal{A}^*$ applies a best-first search strategy, where visiting a state s is determined by a cost function $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the path from s_0 to s, while $h(s)$ is an estimate for the cost of the path from s to s_*. The state with the least $f(.)$ value is visited next, until the goal is reached. If the function f is *admissible*, that is, it does not *overestimate* the path cost, $\mathcal{A}^*$ algorithm is also *admissible*, which means that it is guaranteed to find the path with the least cost by correctly bounding other unexplored paths. Moreover, if $\mathcal{A}^*$ uses the tightest estimate of $h(.)$ to cost the search paths, then all admissible algorithms that are no more informed than $\mathcal{A}^*$ have to visit the same states visited by $\mathcal{A}^*$, which means that $\mathcal{A}^*$ is optimal in the number of visited states.

Similar to $\mathcal{A}^*$ search, OptU-Top$k$ starts from an empty (initial) state, and terminates at a complete (goal) state with the *maximum* probability. For a state s_l, let the cost function $f(s_l) = \mathcal{P}(s_l)$. Based on Property 1, $\mathcal{P}(s_l) \geq \mathcal{P}(s_k)$ for any complete state s_k derived from s_l. Hence, $\mathcal{P}(s_l)$ is admissible since it does not *underestimate* state probability. Since OptU-Top$k$ terminates when finding

a complete state whose probability is not below $\mathcal{P}(s_l)$ of any other state in the space, it follows that OptU-Top$k$ is admissible.

Further, $\mathcal{P}(s_l)$ is the tightest upper-bound of the probability of a complete state derived from s_l, under the tuple access and available dependency information assumptions. To see this, assume that there exist $k - l$ nonretrieved tuples, where each tuple has a probability 1, and is independent of all other tuples. In this case, the probability of a complete state derived from s_l is the same as $\mathcal{P}(s_l)$. It follows that OptU-Top$k$ is an instance of $\mathcal{A}^*$ with optimality guarantees on the number of visited states, as we rigorously prove next.

THEOREM 2. *Let $\mathcal{C}$ be the class of algorithms that search the problem space defined in Section* 5.1 *under the assumptions given in Section* 4.2 *to correctly find a complete state with the highest probability (i.e., members of $\mathcal{C}$ can use arbitrary state traversal orders not necessarily based on state probability). Then, any algorithm $\mathcal{A} \in \mathcal{C}$ must visit all the states visited by OptU-Top$k$.*

PROOF. The proof is similar to the optimality proof of $\mathcal{A}^*$ algorithm [Hart et al. 1968].

Case 1. *State probabilities have no ties.* Assume there exists some incomplete state s_l skipped by $\mathcal{A}$ but visited by OptU-Top$k$. Let the answer reported by $\mathcal{A}$ be x_k, and the answer reported by OptU-Top$k$ be y_k. Since both $\mathcal{A}$ and OptU-Top$k$ are correct, it follows that $x_k = y_k$. Since s_l is visited by OptU-Top$k$, then $\mathcal{P}(s_l) > \mathcal{P}(y_k)$ (based on Property 1 and the state traversal order of OptU-Top$k$). Hence, $\mathcal{P}(s_l) > \mathcal{P}(x_k) \quad \ldots^{(\dagger)}$. However, since $\mathcal{A}$ has skipped s_l, and $\mathcal{A}$ is correct, it follows that $\mathcal{P}(s_l) < \mathcal{P}(x_k) \quad \ldots^{(\ddagger)}$. By contradiction of (†) and (‡), Theorem 2 is proven for Case 1.

Case 2. *State probabilities can have ties.* Let OptU-Topk^* be the set of algorithms identical to OptU-Top$k$, but each algorithm resolves ties in a different way, such that members of OptU-Topk^* cover all possible tie-breaking rules. In the following, we show that there exists some member $Y^* \in$ OptU-Topk^* such that any algorithm $\mathcal{A} \in \mathcal{C}$ must visit all the states visited by Y^*. Assume $\mathcal{A}$ visits the same states as Y^* until some state s_l that is skipped by $\mathcal{A}$ but visited by Y^*. Let the next state visited by $\mathcal{A}$ be $\acute{s}_l$. Since, at this point, s_l is the most probable state in the space, we have two possibilities: (i) $\mathcal{P}(\acute{s}_l) < \mathcal{P}(s_l)$; and (ii) $\mathcal{P}(\acute{s}_l) = \mathcal{P}(s_l)$. However, possibility (i) contradicts with $\mathcal{A}$ being correct as proven in Case 1. Hence, we consider possibility (ii). Let $Y^* = Y_1^*$, where $Y_1^* \in$ OptU-Topk^*, and Y_1^* picks $\acute{s}_l$ as its next state. By repeating the above procedure at each state visited by Y^*, and skipped by $\mathcal{A}$, we can show that there exists a member $Y^* \in$ OptU-Topk^*, such that any state visited by Y^* must be visited by $\mathcal{A}$ as well, and hence Theorem 2 is proven for Case 2. □

It follows from Theorem 2 that $\mathcal{A}$ must consume at least the same number of tuples as OptU-Top$k$ (or a member of OptU-Topk^* in the case of probability ties) since $\mathcal{A}$ visits at least the same states as OptU-Top$k$.

The optimality of Algorithm OPTU-kRanks is defined in terms of the number of retrieved tuples, since OPTU-kRanks materializes all space states based on the retrieved tuples. We formulate our result in Theorem 3.

THEOREM 3. *Algorithm* OPTU-kRanks *retrieves the least number of tuples to compute U-kRanks query answers.*

PROOF. At any step during OPTU-kRanks processing, let ω_j be the set of materialized states with length j, and let $Z_j = \sum_{s_j \in \omega_j} \mathcal{P}(s_j)$. Assume OPTU-$k$Ranks reports t as the U-kRanks answer for rank i, while the termination condition is not reached, that is, $P_{t,i} < \sum_{j<i} Z_j$. Assume the next nonretrieved $i-1$ tuples are $\{u_1, \ldots, u_{i-1}\}$, such that u_j is *implied* by each state in ω_{j-1}, and *exclusive* with any other state s_l, for $l \neq j-1$. It follows that u_j extends all states in ω_{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 for $l \neq j-1$ will not change. By induction, it follows that $P_{u_{i-1},i} = \sum_{j<i} Z_j$. Then, $\sum_{j<i} Z_j$ upper-bounds $P_{u,i}$ for a nonretrieved tuple u. Further, assuming a higher upper-bound would be loose, based on Property 1 and the fact that $P_{u,i}$ depends only on states with length $< i$. Hence, $\sum_{j<i} Z_j$ is the tightest upper-bound of $P_{u,i}$. It follows that OPTU-kRanks consumes the least number of tuples to terminate. That is, if any other space search algorithm $\mathcal{A}$, where $\mathcal{A}$ is no more informed than OPTU-kRanks and works under the same assumptions and space model, assumes a looser bound than $\sum_{j<i} Z_j$, then $\mathcal{A}$ cannot terminate while retrieving less tuples than OPTU-kRanks. □

Complexity analysis. We next give time and space complexity analysis. We mean by space complexity the size of the materialized search space. In the following, we assume the cost of computing a state probability by the rule engine is bounded by some constant cost. We denote with n the number of tuples retrieved from the tuple access layer to compute query answer.

Algorithm OPTU-Top$k$. Since OPTU-Top$k$ is an instance of optimal $\mathcal{A}^*$ search, we expect its worst-case complexity bounds to be exponential similar to $\mathcal{A}^*$. We next analyze the complexity bounds. Based on the algorithm description in Section 5.3.1, the lower-bound time complexity of OPTU-Top$k$ is $\Omega(klogk)$, and the corresponding space complexity is $\Omega(k)$. These bounds are achieved when the algorithm always extends the state with the largest length at each step (iteration). This allows the algorithm to terminate in exactly k steps, where, at each step, OPTU-Top$k$ generates two new states that replace their parent state in the priority queue, which costs $O(logk)$ time. The upper-bound (worst case) time complexity of OPTU-Top$k$ is $O(\frac{n^k}{k!} log(\frac{n^k}{k!}))$, and the corresponding space complexity is $O(\frac{n^k}{k!})$. These bounds are achieved under the extreme case, where all possible score-ordered prefixes of the retrieved n tuples, that is, prefixes with lengths $0 \ldots k$, are fully materialized before concluding query answer (the number of possible score-ordered prefixes of length m out of n tuples is in $O(\frac{n^m}{m!})$).

We note, however, that on the average and for practical values of k, the algorithm efficiently computes query answers by exploiting its probability-guided search, and space-pruning criteria. We demonstrate this behavior through experiments conducted on different data distributions and tuple correlations (Section 10.1).

Algorithm OPTU-kRanks. The lower-bound time complexity of OPTU-kRanks is $\Omega(k2^k)$, and its corresponding space complexity is $\Omega(2^k)$. These bounds are achieved when the algorithm terminates in k steps, where in each step the whole space is extended by the new retrieved tuple. The upper-bound time complexity is $O(\frac{n^{k+1}}{k!})$ and the corresponding space complexity is $O(\frac{n^k}{k!})$, since all possible score-ordered prefixes of the retrieved n tuples are maintained and extended by each retrieved tuple.

Based on our experimental evaluation, the main factors that affect the average case performance of our algorithms are the following:

(1) *Score-probability correlation*: In general, positive score-probability correlation leads to finding highly-probable query answers early, which allows for early termination.
(2) *Complexity of tuple dependencies*: Heavy dependencies negatively affect the performance by increasing the probability computation cost in the rule engine.
(3) *Score and probability distributions*: Distributions of scores and probabilities affect the performance of the algorithms. For example, an exponential distribution of tuple probabilities with a fast rate of decay negatively affects the performance since many tuples will have small probabilities.

We empirically demonstrate the effect of these different factors in Section 10.1.

6. RELAXING FRAMEWORK ASSUMPTIONS

Our framework assumptions, discussed in Section 4.2, can be relaxed in some cases where additional information on the problem space is available. In this section we show how to exploit such information for more efficient processing. In Section 6.1, we discuss the effect of relaxing the *available dependency information* assumption, where we assume all tuples are known to be independent. In Section 6.2, we discuss the effect of relaxing our score-ordered *tuple access* assumption, where we use different tuple orders.

6.1 Exploiting Additional Information on Tuple Dependencies

Under general tuple dependencies, top-l states 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 could be reduced significantly if we have an ability to prune states from our search space early. In general, an incomplete state s_l can be pruned if there exists a *complete* state s_k with $\mathcal{P}(s_k) > \mathcal{P}(s_l)$. Hence, for an incomplete state s_l, if we can compute the maximum probability of a valid *complete* state generated from s_l, denoted $p_{max}(s_l)$, we can safely prune all states with probability less than $p_{max}(s_l)$. The rule engine may be able to compute $p_{max}(s_l)$ of a given state s_l. However, this operation is sensitive to the complexity of tuple dependencies, and the rule engine design. We show in the following how to conduct such pruning for the case of independent tuples.

Score-ranked stream

t1:0.2	t2:0.3	t3:0.8	t4:0.2	…

(a)

length	candid.	prob
0	¬t1	0.8
1	t1	0.2

(b)

length	candid.	prob
0	¬t1, ¬t2	0.56
✗1	t1, ¬t2	0.14
1	¬t1, t2	0.24
2	t1,t2	0.06

(c)

length	candid.	prob
✗0	¬t1,¬t2,¬t3	0.112
1	¬t1,¬t2,t3	0.448
✗1	¬t1,t2,¬t3	0.048
✗2	t1,t2,¬t3	0.012
2	¬t1,t2,t3	0.192
3	t1,t2,t3	0.048

(d)

length	candid.	prob
1	¬t1,¬t2,t3,¬t4	0.358
✗2	¬t1,¬t2,t3, t4	0.09
2	¬t1,t2,t3, ¬t4	0.15
3	t1,t2,t3	0.048
✗3	¬t1,t2,t3, t4	0.04

Fig. 7. `IndepU-Top`*k* processing steps.

6.1.1 *U-Topk Queries Under Independence.* Under tuple independence, we can aggressively prune incomplete states, early in our search, to keep only the states that could lead to query answer. This property is formulated in Lemma 1.

LEMMA 1. *Under tuple independence, a state y_m is pruned if there exists another state x_n with $\mathcal{P}(x_n) > \mathcal{P}(y_m)$, such that x_n and y_m are both maintained after seeing the same set of score-ranked tuples and $n \geq m$.*

Lemma 1 formulates a setting where an incomplete state y_m can be pruned early from the search space. The reason is that, based on the conditions in Lemma 1, the most probable complete states derived from each of the states x_n and y_m are obtained using the same set of unseen tuples. Due to tuple independence, x_n and y_m would use exactly the same set of existence/absence events of unseen tuples to reach complete states. However, since $n \geq m$, the state x_n will reach a complete state at most in the same number of steps as y_m. Since $\mathcal{P}(x_n) > \mathcal{P}(y_m)$, and based on Property 1, we conclude that the complete state derived from x_n would have a higher probability than the one derived from y_m, and we can thus prune y_m early from our search space.

Algorithm `IndepU-Top`*k* uses Lemma 1 by grouping states based on state *length*. The algorithm keeps *at most one* state for each *length* $0 \ldots k$ in a candidate set. The candidate set is extended upon retrieving a new tuple. `IndepU-Top`*k* terminates when (1) at least k tuples have been retrieved, and (2) the probability of any current state is not above the probability of the current *complete* candidate.

Consider for example the score-ranked tuple stream in Figure 7 (fractions indicate probabilities, 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 in I_{s_l} (tuples seen but not included in s_l) with the ¬ symbol. In step (a), after retrieving the first tuple $t1$, we construct two states $\langle \neg t1 \rangle$ and $\langle t1 \rangle$ with lengths 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, $\langle t1, \neg t2 \rangle$ and $\langle \neg t1, t2 \rangle$, are generated. However, we keep only the candidate with the highest probability, since the other candidate is pruned based on Lemma 1.

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

Score-ranked stream	t1:0.3	t2:0.9	t3:0.6	t4:0.25	t5:0.8	...

	t1	t2	t3	t4	t5
Rank 1	0.3	0.63	0.042	0.007	0.0168
Rank 2	0	0.27	0.396	0.0765	0.1892
Rank 3	0	0	0.162	0.126	0.3636

Fig. 8. IndepU-kRanks processing steps.

class. Note that the candidate $\langle \neg t1, \neg t2, \neg t3\rangle$ is pruned because there is another candidate $\langle \neg t1, \neg t2, t3\rangle$ with a larger length and higher probability. In step (c) we reach the first *complete* candidate, $\langle t1, t2, t3\rangle$, 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 $\langle t1, t2, t3\rangle$ will be beaten. Applying the space pruning criterion given in Lemma 1 results in significant performance improvements as we illustrate in Section 10.1.

Complexity analysis. Let n be the total number of consumed tuples. For each tuple, Algorithm IndepU-Top$k$ extends at most k states. The time complexity is thus in $O(nk)$, while space complexity is in $O(k)$.

6.1.2 *U-kRanks Queries Under Independence.* Under tuple independence, a U-kRanks query exhibits the *optimal substructure* property, that is, 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-kRanks, a dynamic programming algorithm to answer U-kRanks queries under the independence of tuple events.

Consider the example depicted by Figure 8, where we are interested in U-3Ranks query answer. In the table in Figure 8, a cell at row i and column x contains $P_{x,i}$ (the probability of tuple x to be at rank i). Hence, $P_{x,1} = \Pr(x.e) \times \prod_{z:\mathcal{F}(x)<\mathcal{F}(z)}(1 - \Pr(z.e))$, which is the probability that $x.e$ is true and all tuple events with higher scores are false. The computation of the probabilities in the remaining rows is based on the following property:

Property 2. *Under tuple independence and for $i > 1$,*

$$P_{x,i} = \Pr(x.e) \times \sum_{y:\mathcal{F}(y)>\mathcal{F}(x)} P_{y,i-1} \times \prod_{z:\mathcal{F}(x)<\mathcal{F}(z)<\mathcal{F}(y)} (1 - \Pr(z.e))$$

The rationale of 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 as the probability that x exists, each tuple z that appears at an intermediate rank between x and y does not exist, and y appears at rank $i - 1$.

For example, in Figure 8, $P_{t2,2} = 0.9 \times 0.3 = 0.27$, 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 ranked possible worlds. This means that we can report an answer from any row whenever the maximum probability in that row is greater than $(1 - \textit{sum(row probabilities)})$. Notice also that the computation in each row depends solely on the row above.

The description above gives rise to the following dynamic programming formulation. We construct a matrix M with k rows, where 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 as follows:

$$M[i,t] = \begin{cases} \Pr(t.e) \times \prod_{z:\mathcal{F}(t)<\mathcal{F}(z)} (1 - \Pr(z.e)) & \textit{if } i = 1 \\ \Pr(t.e) \times \sum_{y:\mathcal{F}(y)>\mathcal{F}(t)} M[i-1, y] \times \prod_{z:\mathcal{F}(t)<\mathcal{F}(z)<\mathcal{F}(y)} (1 - \Pr(z.e)) & \textit{if } i > 1 \end{cases} \quad (5)$$

For example in Figure 8, $M[2, 3] = \Pr(t3.e) \times (M[1, 2] + (1 - \Pr(t2.e)) \times M[1, 1])$. Algorithm `IndepU-kRanks` returns a set of k tuples $\{t_1 \ldots t_k\}$, where $t_i = argmax_x\ M[i, x]$.

Complexity analysis. The size of the matrix M is in $O(nk)$, where n is the number of consumed tuple. For each consumed tuple, the algorithm scans the matrix rows to compute tuple probability at each rank. The time complexity is thus in $O(n^2k)$.

6.2 Using Other Tuple Retrieval Orders

Our previous algorithms retrieve tuples in score order, which is the necessary order to compute nontrivial lower-bounds on state probabilities for general tuple dependencies, as we show in Section 5.1. We show in this section two settings where probability order can be used to bound states probabilities, when tuples are independent.

Combining score and probability orders. We describe an adaptation of our algorithms in Section 6.1 to combine score and probability orders in a *threshold algorithm*-like fashion. Threshold algorithm (TA) [Fagin et al. 2003] aggregates multiple ranked lists for the same set of objects based on a monotone score aggregation function. TA finds the top-k objects by scanning the ranked lists in parallel, computing the aggregated score of seen objects, and bounding the scores of unseen objects.

We assume two ranked lists sorting tuples in descending score and probability orders. The aggregation function is not explicit in our settings. However, based on Property 1, state probability is monotone in the number of state tuple events, that is, adding a tuple event to some state does not increase its probability. We thus use the score list to construct states, and the probability list to further shrink state probabilities by upper-bounding the probabilities of nonretrieved tuples in the score list. We give a high-level description of our TA-adaptation of Algorithm `IndepU-Topk` in the following:

(1) Retrieve a tuple t from the score list, use random access to the probability list to get the probability of t. Update search states based on t, as described in Algorithm `IndepU-Topk`.

(2) Retrieve a tuple $\acute{t}$ from the probability list. If $\acute{t}$ is not retrieved before in the score list, let $\overline{p}$ be the probability of $\acute{t}$. Hence, $\overline{p}$ upper-bounds the probability of any nonretrieved tuple in the score list. We do not need to probe the score list with $\acute{t}$ since it comes out of score order.

(3) For each state materialized s_l, the maximum probability of a complete state reached from s_l is $\mathcal{U}(s_l) = \mathcal{P}(s_l) \times \overline{p}^{(k-l)}$. We use $\mathcal{U}(s_l)$, instead of $\mathcal{P}(s_l)$, to decide on state pruning in `IndepU-Topk` using the same pruning criteria in Lemma 1.

(4) Repeat from step (1) until the termination conditions of `IndepU-Topk` are reached.

In step (3), $\mathcal{U}(s_l)$ is computed by optimistically assuming that s_l extends to a complete state by appending $k - l$ tuples to s_l, each has a probability $\overline{p}$. This bound is correct since it assumes the maximum probability that can be achieved by a complete state derived from s_l, based on the largest possible probability of the nonretrieved tuples. Since $\mathcal{U}(s_l) \leq \mathcal{P}(s_l)$, using probability order allows further reduction in the probabilities of incomplete states, which can lead to faster termination of `IndepU-Topk`.

If *arbitrary tuple dependencies* exist, probability order can give a tighter upper-bound for the probability of any complete state derived from a state s_l as $\mathcal{U}(s_l) = min(\mathcal{P}(s_l), \overline{p})$, since the probability of an event conjunction is bounded by the minimum probability of the conjuncted events. However, incomplete states cannot be pruned early by comparing their bounds, since a lower-bound on state probability is still trivially 0 (e.g., consider the case of a nonretrieved tuple in the score list whose event is mutually exclusive with one event in the incomplete state).

Using probability order only. When further information is available, probability order can be used by itself to compute probabilistic top-k queries, while possibly providing early query termination. Specifically, knowing N, the total number of tuples in query output, and that all tuples are independent, we can compute nontrivial lower bounds on state probabilities. For example, assume 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, a state $s_2 = \langle t2, t1 \rangle$ has a probability lower-bound of $0.1 \times 1.0 \times (1 - 0.1)^{(N-2)}$, which is computed pessimistically by assuming all nonretrieved tuples have the highest possible probability (0.1), and higher scores than $t1$. Hence, s_2 is pessimistically the top-2 in all worlds that exclude all the remaining tuples. Note that the knowledge of N is essential to compute this bound, since otherwise the absence probability of nonretrieved tuples cannot be bounded. Nontrivial lower-bounds of state probabilities clearly allow for early query termination.

We note, however, that the lower-bounds computed in the above manner can be very loose, that is, they heavily underestimate actual probabilities, since the lower-bound is polynomial in the number of nonretrieved tuples, which is usually a large number. Hence, in practice, it is hard to argue for the value of using probability order by itself.

7. PROBABILISTIC TOP-k AGGREGATE QUERIES

We now describe our approach to compute probabilistic top-k aggregate queries. The aggregate functions we study are in the class of *bounded functions*, defined as follows: an aggregate function $\mathcal{F}$ is *bounded* if the upper-bound (lower-bound) of $\mathcal{F}$ on a group g can be obtained by assuming all members of g have the maximum (minimum) scores in g. Bounded aggregate functions are monotone, since increasing (decreasing) the maximum scores in g results in increasing (decreasing) the value of $\mathcal{F}$. Typical aggregate functions such as *sum* and *average* are bounded functions. The properties of bounded functions allow bounding group aggregates, based on a partial retrieval of group tuples.

The main idea of our approach to compute probabilistic top-k aggregate queries is to conduct space search, staged on two levels: (1) *intragroup* search to provide ranked retrieval of group aggregates that is necessary to incrementally construct the space; and (2) *intergroup* search to find the query answer in the space of group aggregates' combinations. In the first stage, we compute ordered nonoverlapping aggregate ranges by lazily consuming group tuples to materialize only the necessary parts in the aggregate distribution within each group (Section 7.2.1). In the second stage, we use probability-guided search techniques to navigate the space of possible answers. We allow early query termination by bounding the probabilities of explored and unexplored search paths (Section 7.2.2).

The techniques in this section are similar to the techniques in Section 5 in adopting score-ordered retrieval. However, while the retrieved units in Section 5 are the tuples, the retrieved units in this section are ranked partitions of group aggregate distribution.

7.1 Problem Space

We define the space of top-k aggregate query answers as a space of possible top-k group vectors aggregated across ranked possible worlds. In the following, we use the term "tuple instances" to refer to the existence and absence events, $t.e$ and $\neg t.e$, of tuple t. We start by giving formal definitions for space components.

Definition 6. *Group World.* Let $g_i = \{t_1, \ldots, t_n\} \subseteq \mathcal{D}$ be a group in $\mathcal{D}$. A group world $gw_i^j = \{\acute{t}_1, \ldots, \acute{t}_n\}$ of g_i is obtained by picking an instance $\acute{t}$ of each tuple $t \in g_i$. $\Pr(gw_i^j) = \Pr(\acute{t}_1, \ldots, \acute{t}_n)$, and the aggregate value of gw_i^j results from applying $\mathcal{F}$ to the tuple instances in gw_i^j corresponding to existence events.

Group worlds project the space of possible worlds on the grouping attributes. For example, Figure 9(b) shows the group worlds, for the aggregate function *sum*(*Score*), for the probabilistic relation in Figure 9(a). The group world $gw_1^1 = \{t1, \neg t4\}$ of g_1 is obtained by picking a possible instance for each of $t1$ and $t4$. In this example we assume independence among instances of different tuples, which means that the joint probability of tuple instances is obtained by multiplying their probabilities. Therefore, $\Pr(gw_1^1) = \Pr(t1) \times \Pr(\neg t4) = 0.6 \times 0.9 = 0.54$. The aggregate value of a group world is obtained by applying the aggregate function to tuple instances corresponding to

Grouping Attribute → gid; Scoring Attribute → Score; Membership probability → Prob.

tID	gid	Score	Prob.
t1	1	100	0.6
t2	2	120	0.7
t3	3	90	0.4
t4	1	80	0.1

(a)

g1

Group World	Sum	Prob.
$gw_1^1 = \{t1, \neg t4\}$	100	0.54
$gw_1^2 = \{\neg t1, t4\}$	80	0.04
$gw_1^3 = \{t1, t4\}$	180	0.06
$gw_1^4 = \{\neg t1, \neg t4\}$	0	0.36

g2

Group World	Sum	Prob.
$gw_2^1 = \{t2\}$	120	0.7
$gw_2^2 = \{\neg t2\}$	0	0.3

g3

Group World	Sum	Prob.
$gw_3^1 = \{t3\}$	90	0.4
$gw_3^2 = \{\neg t3\}$	0	0.6

(b)

Global World	Group Ranking	Prob.
$GLW^1 = \{gw_2^1, gw_1^1, gw_3^1\}$	g2,g1,g3	0.15
$GLW^2 = \{gw_2^1, gw_1^1, gw_3^2\}$	g2,g1	0.226
$GLW^3 = \{gw_1^1, gw_3^1, gw_2^2\}$	g1,g3	0.064
$GLW^4 = \{gw_1^1, gw_2^2, gw_3^2\}$	g1	0.097
$GLW^5 = \{gw_2^1, gw_3^1, gw_1^2\}$	g2,g3,g1	0.011
$GLW^6 = \{gw_2^1, gw_1^2, gw_3^2\}$	g2,g1	0.017
$GLW^7 = \{gw_3^1, gw_1^2, gw_2^2\}$	g3,g1	0.005
$GLW^8 = \{gw_1^2, gw_2^2, gw_3^2\}$	g1	0.007
$GLW^9 = \{gw_1^3, gw_2^1, gw_3^1\}$	g1,g2,g3	0.017
$GLW^{10} = \{gw_1^3, gw_2^1, gw_3^2\}$	g1,g2	0.025
$GLW^{11} = \{gw_1^3, gw_3^1, gw_2^2\}$	g1,g3	0.007
$GLW^{12} = \{gw_1^3, gw_2^2, gw_3^2\}$	g1	0.01
$GLW^{13} = \{gw_2^1, gw_3^1, gw_1^4\}$	g2,g3	0.1
$GLW^{14} = \{gw_2^1, gw_3^2, gw_1^4\}$	g2	0.15
$GLW^{15} = \{gw_3^1, gw_2^2, gw_1^4\}$	g3	0.043
$GLW^{16} = \{gw_1^4, gw_2^2, gw_3^2\}$		0.064

(c)

Fig. 9. (a) Probabilistic relation with grouping attribute; (b) group worlds; (c) global worlds.

existence events in that group world. For example, $sum(gw_1^3) = sum(100, 80) = 180$. The aggregate of each group is a probability distribution on the aggregates of group worlds, as opposed to a single value as in deterministic databases.

Definition 7. *Global World.* Let $\mathcal{D}$ be a probabilistic database with m different groups, and $\mathcal{F}$ be a group aggregate function. Let $gw(g_i) = \{gw_i^1 \dots gw_i^{n_i}\}$ be the set of group worlds of group g_i in $\mathcal{D}$, for $i = 1 \dots m$. A global world $GLW^j = Sorted_{\mathcal{F}}(\{g\acute{w}_1 \dots g\acute{w}_m\})$ is obtained by picking exactly one group world $g\acute{w}_i$ from each group g_i, and sorting all picked group worlds based on $\mathcal{F}$. The probability of GLW^j is computed as $\Pr(GLW^j) = \Pr(g\acute{w}_1, \dots, g\acute{w}_m)$.

Figure 9(c) shows the global worlds for the given probabilistic relation and the group ranking corresponding to each world. For example, $GLW^9 = \{gw_1^3, gw_2^1, gw_3^1\} = \{t1, t4, t2, t3\}$ corresponds to the group ranking $\langle g1, g2, g3\rangle$, where $\Pr(GLW^9) = 0.06 \times 0.7 \times 0.4 = 0.017$, assuming tuple independence.

Based on Definition 7, a global world is a grouped and ranked version of one possible world of the database. We show in Section 7.2 that treating possible worlds as global worlds makes it easier to understand the semantics of probabilistic top-*k* aggregate queries.

Let $\mathcal{D}$ be a probabilistic database. Let $\mathcal{PW} = \{GLW^1, \dots, GLW^n\}$ be the set of global worlds of $\mathcal{D}$ based on grouping attributes $\mathcal{A}$ and a group aggregate function $\mathcal{F}$. Let $P_{g,i}$ be the probability of group identifier g to appear at rank i. We now redefine our problem based on our space components.

Definition 8. *U-Topk-Agg Query* (*revised*). Let $\mathcal{G}_{i,k}$ be a *prefix* of length k in GLW^i, where $\Pr(\acute{\mathcal{G}_k}) = \sum_{GLW^i : \mathcal{G}_{i,k} = \acute{\mathcal{G}_k}} \Pr(GLW^i)$. A U-Top$k$-Agg query returns a k-length global world prefix $\mathcal{G}_k^*$ with the highest probability.

For example, in Figure 9(c), U-Top2-Agg query answer is $\langle g2, g1 \rangle$ with probability 0.39, which is the summation of the probabilities of GLW^1, GLW^2, and GLW^6.

Definition 9. *U-kRanks-Agg Query* (*revised*). Let $\gamma_{i,j}$ be the group identifier that appears at rank i in the global world GLW^j. Then, $P_{g,i} = \sum_{GLW^j : \gamma_{i,j} = g} \Pr(GLW^j)$. A U-$k$Ranks-Agg query returns a set of k group identifiers $\{g^*_{(1)}, \ldots, g^*_{(k)}\}$, where for each $g^*_{(i)}$, the value of $P_{g^*_{(i)},i}$ is the maximum among all other group identifiers.

In Figure 9(c), U-2Ranks-Agg answer is $\{g2 : 0.65, g1 : 0.4\}$, since $g2$ is ranked 1st in $\{GLW^1, GLW^2, GLW^5, GLW^6, GLW^{13}, GLW^{14}\}$, and $g1$ is ranked 2nd in $\{GLW^1, GLW^2, GLW^6, GLW^7\}$ with probability summations 0.65 and 0.4, respectively.

7.2 Space Navigation

In this section we describe how to explore the space of probabilistic ranking aggregates. In Section 7.2.1 we show how to provide aggregate-ordered retrieval, while in Section 7.2.2 we show how to combine group aggregates to compute query answer.

7.2.1 *Intragroup Space Navigation.* Our intragroup aggregation layer produces an aggregate-ranked retrieval of (partial) group worlds, which is necessary to minimize the materialized space size and the number of retrieved tuples to evaluate our queries (Theorem 5 formalizes the necessity of this retrieval order). We start by formulating intragroup space, followed by describing the space navigation techniques.

Definition 10. *Group State.* A group state s_g of group g is a set of tuple instances $T_{s_g} = \{\acute{t}_1 \ldots \acute{t}_m\}$ derived from (a subset of) g tuples. The group state s_g is described by (1) its probability: $\Pr(s_g) = Pr(T_{s_g}) = \Pr(\acute{t}_1 \wedge \ldots \wedge \acute{t}_m)$; and (2) its aggregate range: let $\underline{\mathcal{R}}^{\mathcal{F}}(s_g)$ be the lowest aggregate value of g given T_{s_g}, and $\overline{\mathcal{R}}^{\mathcal{F}}(s_g)$ be the highest aggregate value of g given T_{s_g}. The aggregate range of s_g is defined as $\mathcal{R}^{\mathcal{F}}(s_g) = [\underline{\mathcal{R}}^{\mathcal{F}}(s_g), \overline{\mathcal{R}}^{\mathcal{F}}(s_g)]$.

Each state s_g aggregates all group worlds of g subsuming T_{s_g}. We use group states to incrementally navigate the space of group worlds by refining group states, as needed, using an incremental retrieval of group tuples. The aggregate ranges of group states allows ordering the groups, based on a partial space materialization.

Computing group state probability. Consider Figure 10. Assume a group $g = \{t1, t2, \ldots, tn\}$, where all tuples are independent. Assume that $t1$ has score 5 and probability 0.4, while $t2$ has score 4 and probability 0.8. Let *sum* be the aggregate function. Figure 10(a) shows g's world space after consuming $t1$. The space is *partitioned* into two *group states*: $s_g^1 = \{t1\}$, and $s_g^2 = \{\neg t1\}$. $\Pr(s_g^1) = 0.4$

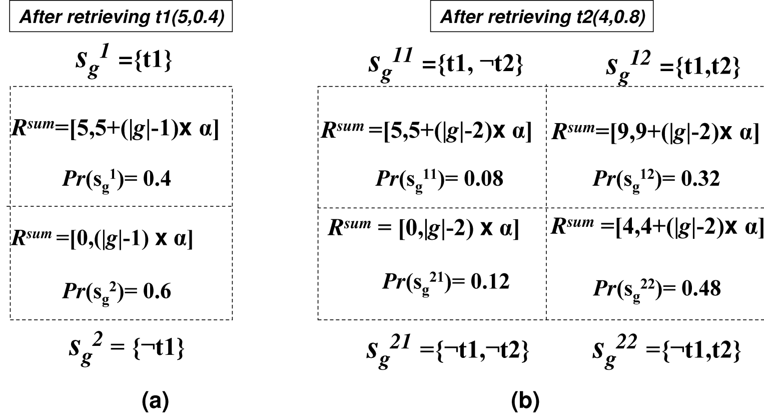

Fig. 10. Group worlds space navigation.

is the probability summation of all g's worlds that include $t1$, while $\Pr(s_g^2) = 0.6$ is the probability summation of all g's worlds that do not include $t1$. Each group state thus aggregates a part of g's space. Figure 10(b) shows how g's space is further partitioned into four finer states s_g^{11}, s_g^{12}, s_g^{21}, and s_g^{22}, after consuming $t2$. State probabilities are computed by multiplying the corresponding tuple events, assuming independence. In general, probability computation is performed by the rule engine component (Section 4), which maintains tuple dependencies. An important property that applies to group states is the following:

PROPERTY 3. *Any two states s_g^i and s_g^j of the same group g are mutually exclusive.*

Property 3 holds because no group world is subsumed by two group states at the same time, since each state is conditioned on an event that conflicts with all other states. For example, in Figure 10(b), s_g^{11} is conditioned on $(t1 \wedge \neg t2)$, while s_g^{12} is conditioned on $(t1 \wedge t2)$.

Computing the aggregate range of group state. Computing group aggregate bounds, from a partial retrieval of group tuples, is possible if we have knowledge about (1) group size; and (2) bounds over tuple score. Li et al. [2006] show that for a *bounded* aggregate function $\mathcal{F}$ applied to a group $g = \{t_1, \ldots, t_n\}$, where α is the maximum tuple score in g, an algorithm that retrieves g tuples one by one can upper-bound $\mathcal{F}(g)$ as follows:

$$\overline{\mathcal{F}(g)} = \mathcal{F}(x_1, \ldots, x_n),\ where\ \ x_i = \begin{cases} t_i.score & if\ \ t_i\ is\ seen \\ \alpha & if\ \ t_i\ is\ unseen \end{cases} \tag{6}$$

$\mathcal{F}(g)$ can be lower-bounded similarly, if we know the minimum score of g tuples. We build on these results to bound the aggregates of group states.

As described in Section 4, we assume the tuple access layer, is *rank-aware* and *group-aware*. By rank-awareness we mean that tuples are pipelined in score order, which can be realized efficiently using available rank-aware query operators [Ilyas et al. 2004], and rank-aware relational algebra [Li et al. 2005]. By group-awareness we mean the knowledge of (bounds of) the size of each group

(the *group cardinality information* assumption), and the existence of an interface allowing consuming tuples from specific groups incrementally. Proposed techniques can satisfy group-awareness in different forms including the use of materialized views [Goldstein and Larson 2001] and index striding [Hellerstein et al. 1997]. We discuss these techniques in Section 11.

Consider Figure 10(a). We show how to compute $\mathcal{R}^{sum}(s_g^1)$. Assume that the minimum tuple score in g is 0. Then, $\underline{\mathcal{R}}^{sum}(s_g^1) = 5$, since any group world subsuming s_g^1 must include $t1$ which has score 5. In addition, since only $t1$ is included in s_g^1, there are $|g| - 1$ "unknown" tuples that can further partition s_g^1. Assume that the maximum score of such tuples is α, then $\overline{\mathcal{R}}^{sum}(s_g^1) = 5 + (|g| - 1) \times \alpha$. Similarly, since $t1$ does not belong to s_g^2, then $\underline{\mathcal{R}}^{sum}(s_g^2) = 0$ while $\overline{\mathcal{R}}^{sum}(s_g^2) = (|g| - 1) \times \alpha$. Retrieving tuples in score order allows reducing α, the maximum score of unseen tuples. The aggregate range of group states, for aggregate functions other than *sum*, can be computed similarly. Group space can thus be navigated *lazily* by incrementally retrieving tuples in score-order, and using these tuples to refine the aggregate bounds of group states. The goal is to partition group space into aggregate-ordered group states.

Algorithm 3. `Intra-Group Aggregation`

INITIALIZE (g : Tuple Group)
1 $\mathcal{Q} \leftarrow$ priority queue for states of g ordered on $\overline{\mathcal{R}}^{\mathcal{F}}(.)$
2 $s_g^0 \leftarrow$ an empty state, where $T_{s_g^0} = \phi$, $\Pr(s_g^0) = 1$, and $\mathcal{R}^{\mathcal{F}}(s_g^0) = [\underline{\mathcal{F}}(g), \overline{\mathcal{F}}(g)]$
3 Insert s_g^0 into $\mathcal{Q}$
GETNEXTSTATE (g : Tuple Group)
1 **while** ($\mathcal{Q}$ is not empty)
2 **do**
3 **if** ($\mathcal{Q}.size > 1$)
4 **then**
5 $s_g^* \leftarrow$ top state in $\mathcal{Q}$; $s_g^{**} \leftarrow$ second top state in $\mathcal{Q}$
6 **if** ($\underline{\mathcal{R}}^{\mathcal{F}}(s_g^*) \geq \overline{\mathcal{R}}^{\mathcal{F}}(s_g^{**})$)
7 **then return** s_g^*
8 $s_g^* \leftarrow$ dequeue ($\mathcal{Q}$)
9 $t \leftarrow$ next g tuple in score order not used yet in partitioning s_g^*
10 Partition s_g^* using t into s_g^1 and s_g^2
11 Insert s_g^1 and s_g^2 into $\mathcal{Q}$
12 **return** null

Algorithm 3 gives the details of our intragroup aggregation procedure. On each invocation of procedure GETNEXTSTATE, the algorithm returns the next state of group g in aggregate order. This is done by maintaining a priority queue (initialized in procedure INITIALIZE) of visited group states ordered on aggregate upper-bound. Group tuples are retrieved, when needed, in score order. At each step, the state s_g^* at queue top is partitioned using the next tuple in score order that is not used yet in partitioning s_g^*. Partitioning a state results in two new states that replace the old state in the queue. State partitioning ends when the

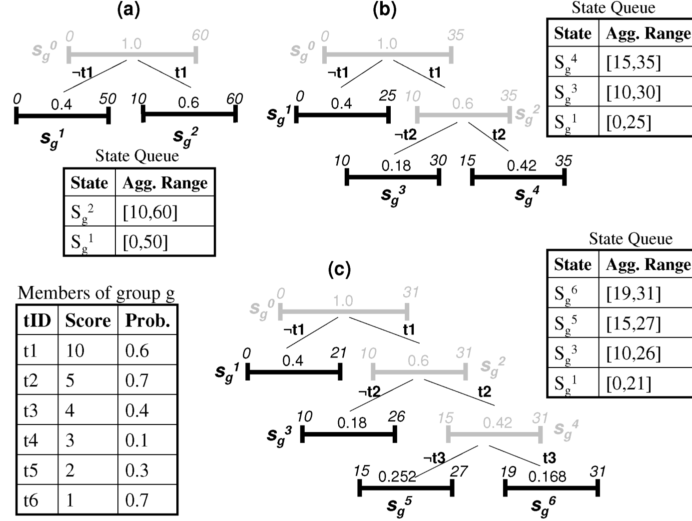

Fig. 11. Partitioning group worlds space.

aggregate lower-bound of queue top state is higher than the aggregate upper-bound of all other states, and the algorithm reports the state at queue top. We prove the necessity of visiting group states in aggregate upper-bound order in Theorem 4.

We illustrate Algorithm 3 using Figure 11. For the shown group g, tuple score is assumed to be in [0,10], $|g| = 6$, and the aggregate function is *sum*. The algorithm starts from state s_g^0, where $\mathcal{R}^{sum}(s_g^0) = [0, 60]$ and $\Pr(s_g^0) = 1$. In Figure 11(a), $t1$ partitions s_g^0 into s_g^1 and s_g^2. The probabilities and aggregate ranges of s_g^1 and s_g^2 are computed, and they are stored in the state priority queue. In Figure 11(b), the top queue state s_g^2 is partitioned into s_g^3 and s_g^4, using $t2$. At this point, we update our information about the scores of unseen g tuples, which is now bounded by 5. Moreover, $\overline{\mathcal{R}}^{sum}(s_g^1)$ is also updated to $sum(0, |g-1| \times 5) = 25$, without actually partitioning s_g^1. In Figure 11(c), s_g^4 is partitioned using t_3, and the upper-bounds of s_g^1 and s_g^3 are updated. Note that the state queue maintains only the leaves of the state partitioning tree.

THEOREM 4. *The group state with the highest aggregate upper-bound cannot be skipped while partitioning group space using incrementally retrieved score-ordered tuples.*

PROOF. Assume an algorithm $\mathcal{A}$ that incrementally retrieves score-ordered group tuples to find the next group state in aggregate order. Assume that $\mathcal{A}$ skipped partitioning s_g^*, the state with the highest aggregate upper-bound, but could correctly report $\acute{s}_g$, the next state in aggregate order. Hence, $\underline{\mathcal{R}}^{\mathcal{F}}(\acute{s}_g) \geq \overline{\mathcal{R}}^{\mathcal{F}}(s_g^*) \ldots (\dagger)$.

However, since s_g^* is not partitioned by $\mathcal{A}$, the value of $\overline{\mathcal{R}}^{\mathcal{F}}(s_g^*)$ remains the maximum among the upper-bounds of all states including $\acute{s}_g$. Hence,

the condition (†) cannot be satisfied, and $\mathcal{A}$ has reported an incorrect state. □

It follows from Theorem 4 that any algorithm that works under our framework assumptions by incrementally consuming score-ordered tuples to report nonoverlapping aggregate ranges cannot visit less states than our intragroup aggregation algorithm. The worst-case scenario of our algorithm is to fully materialize the exponential space of group worlds, which can be unavoidable in some cases under the assumptions of our processing framework. In practice, our experiments (Section 10.2.1) show that materialized group space is usually manageable for practical values of k.

We conclude this section by describing how to produce aggregate-ranked states coming from different groups. The intragroup aggregation layer prioritizes groups based on their aggregate upper-bounds. That is, the group with the highest aggregate upper-bound is requested for its next state in aggregate-order. A reported group state $s^l_{g_i}$, from the intragroup aggregation layer, satisfies the following condition: $\underline{\mathcal{R}}^{\mathcal{F}}(s^l_{g_i}) \geq \overline{\mathcal{R}}^{\mathcal{F}}(s^m_{g_j})$, for any unreported state $s^m_{g_j}$ in any group g_j. The output of the intragroup aggregation layer is reported incrementally (on demand) to the intergroup layer, which is described next.

7.2.2 *Intergroup Space Navigation.* We discuss in this section how to use aggregate-ordered group states to search the answer space of ranking-aggregate queries. We start by formulating the search space, then we demonstrate the necessity of consuming aggregate-ordered group states to efficiently evaluate our queries.

Definition 11. *Intergroups State.* An intergroups state S_l, with $l \leq k$, is a vector of l different group identifiers appearing as a prefix in one or more global worlds. $\Pr(S_l) = \sum_{GLW^j:\mathcal{G}_{j,l}=S_l} \Pr(GLW^j)$. An intergroups state is *complete* if $l = k$.

Definition 12. *Global State.* A global state G_l, with $l \leq k$, is vector of l group states ordered on aggregate ranges. $\Pr(G_l) = \Pr(G_l \wedge \neg I_{G_l})$, where I_{G_l} is the set of group states *seen* by G_l in aggregate order, but not included in G_l.

Each intergroups state aggregates global states with a *matching* group identifer vector. We illustrate the above definitions using the next example.

Example 5. *Matching Intergroups State and Global State*. Recall that $s^j_{g_i}$ denotes group state j in group g_i. Assume an intergroups state $S_3 = \langle g_1, g_3, g_4\rangle$. The global state $G_3 = \langle s^1_{g_1}, s^5_{g_3}, s^2_{g_4}\rangle$ matches S_3, since the group identifiers in G_3 match those in S_3 in the same order. Let $I_{G_3} = \{s^1_{g_2}\}$. Then, $\Pr(G_3) = \Pr(s^1_{g_1} \wedge s^5_{g_3} \wedge s^2_{g_4} \wedge \neg s^1_{g_2})$.

Notice that global states do not contain group states with the same group identifier, since group states coming from the same group are mutually exclusive (Property 3). Intergroups states comprise the candidate answers of our ranking-aggregates queries.

Algorithm 4. Inter-Group Aggregation

U-TOPK-AGG (*source*: Aggregate-ordered stream of group states, k : Result Size)
1 $\mathcal{Q} \leftarrow$ priority queue of intergroups states ordered on probability upper-bounds
2 $S_0 \leftarrow$ an empty state, with probability 1
3 Insert S_0 into $\mathcal{Q}$
4 {**while** ($\mathcal{Q}$ is not empty AND *source* not exhausted)
5 **do**
6 $S_l^* \leftarrow$ dequeue ($\mathcal{Q}$)
7 $G_l \leftarrow$ remove the most probable matching global state to S_l^*
8 Extend G_l into G_{l+1} and $\acute{G}_l$, using next state from *source* not seen yet by G_l
9 Insert G_{l+1} and $\acute{G}_l$ in their matching sets of global states
10 Update probability bounds of affected intergroups states
11 $S_l^* \leftarrow$ peek at top state in $\mathcal{Q}$
12 **if** ($l = k$ AND $\Pr(S_l^*) \geq$ probability upper-bounds of other states in $\mathcal{Q}$)
13 **then return** S_l^*
14 **return** null

Similar to our discussion of tuple orders in Section 5.1, retrieving group states in aggregate order is necessary and sufficient to compute nontrivial bounds on the probabilities of intergroups states.

THEOREM 5. *Retrieving group states in aggregate order is necessary and sufficient to compute nontrivial bounds on the probabilities of intergroups states.*

PROOF. Follows directly from Theorem 1's proof. □

Computing U-Topk-Agg queries. Based on Definition 11, the answer to U-Top$k$-Agg query is a *complete* intergroups state with the maximum probability. Our strategy to find such a state is to grow intergroups state space until we have guarantees that a complete state has the maximum probability. Intergroups states are constructed by aggregating global states, which are constructed from the ranked stream of group states consumed from the intragroup aggregation layer.

Let G_l be a global state. Let s_g be the next group state, in aggregate order, not seen yet by G_l. We *extend* G_l using s_g into G_{l+1} (by appending s_g to G_l), and $\acute{G}_l$ (by appending s_g to I_{G_l}). Note that $\Pr(G_l) = \Pr(G_{l+1}) + \Pr(\acute{G}_l)$. An intergroups state is extended by extending one of its matching global states.

Algorithm 4 describes our intergroup aggregation technique for U-Top$k$-Agg query. The algorithm retrieves aggregate-ordered group states from the intragroup aggregation layer. The algorithm visits intergroups states in the order of their probability upper-bounds (we show how to compute these bounds next). Each visited state is extended by extending its matching global state with the highest probability. The search ends when finding a complete intergroups state whose probability is not below the probability upper-bounds of other states. The details of the algorithm are illustrated through the next example.

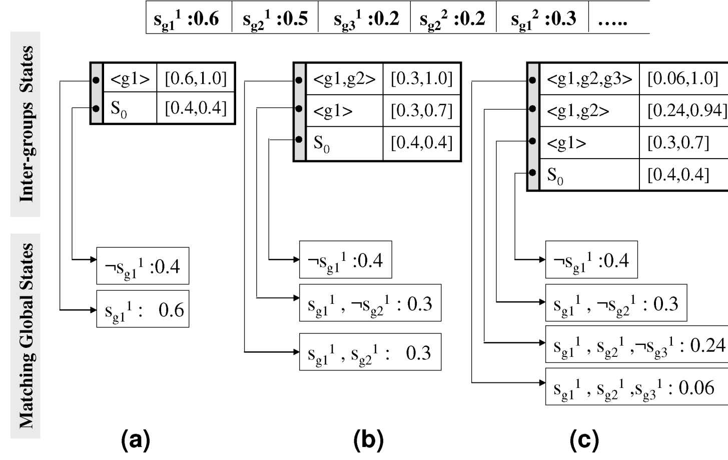

Fig. 12. Inter-groups states and matching global states.

Consider Figure 12, where aggregate-ordered group states produced by the intragroup aggregation layer is shown at the top. Space search starts from the empty state S_0. In Figure 12(a), the first group state $s^1_{g_1}$ extends S_0 into $\langle g_1 \rangle$ and S_0. Each intergroups state is linked to matching global states, and each global state remembers its last seen group state. The current probability of an intergroups state S_l is the summation of the probabilities of its currently matching global states. The actual probability of S_l may be larger because of nonmaterialized global state that match S_l. We thus upper-bound the probability of S_l by the summation of S_l's current probability, and the current probabilities of other intergroups states sharing a prefix with S_l. For example, the state $S_1 = \langle g1 \rangle$ has a current probability of 0.6, and is upper-bounded by $0.6 + 0.4 = 1.0$, since it shares the *empty* prefix with S_0. The rationale is that a nonmaterialized intergroups state, identical to S_l, could only be produced by extending a state sharing a prefix with S_l. For example, another $\langle g_1 \rangle$ state could only be produced from S_0. Hence, the summation of the current probabilities of S_0 and $\langle g_1 \rangle$ is the maximum probability $\langle g_1 \rangle$ could eventually have.

We buffer intergroups states in a priority queue ordered on their upper-bound probabilities. At each step, the top queue state is extended by extending its current matching global state G_l with the highest probability. In Figure 12(b), $\langle g_1 \rangle$ is extended by extending the global state $\langle s^1_{g_1} \rangle$ using the next group state $s^1_{g_2}$. This results in two global states $\langle s^1_{g_1}, s^1_{g_2} \rangle$ and $\langle s^1_{g_1}, \neg s^1_{g_2} \rangle$. The first state creates a new entry, $\langle g_1, g_2 \rangle$, in the intergroups state queue, while the second state updates the probability of $\langle g_1 \rangle$. The probability upper-bounds of the states are updated accordingly. In Figure 12(c), the state $\langle g_1, g_2 \rangle$ is extended by extending its most probable global state $\langle s^1_{g_1}, s^1_{g_2} \rangle$.

Computing U-kRanks-Agg queries. Our processing algorithm for U-kRanks-Agg queries uses the same intergroup aggregation machinery discussed above. However, in this case, each input group state, coming from the intragroup aggregation layer, is used to extend all maintained global states. We keep a list of group identifiers that appear at each rank $1 \ldots k$ in one or more global states.

For group identifier g_i, we bound $P_{g_i,j}$ as follows. The lower-bound of $P_{g_i,j}$ is computed by adding up the probabilities of all global states having g_i at rank j, while the upper-bound of $P_{g_i,j}$ is computed by adding its lower-bound to the probability summation of all global states G_l, where $l < j$ (as discussed in Section 5.3.2). Similarly, an upper-bound for $P_{g_u,j}$, for any unseen group identifier g_u, is computed by summing the probabilities of all global states G_l, where $l < j$. We report an answer g_i for rank j when the lower-bound of $P_{g_i,j}$ is not below the upper-bound of $P_{g_r,j}$ for all other group identifiers g_r, including the unseen groups.

8. OTHER QUERY TYPES

In this section we discuss other query types that can be computed using our processing framework.

Beyond the most probable answers. We extend our search algorithms to find the l most probable answers, as opposed to terminating on finding the most probable answer, where l is an input parameter. We illustrate this extension using our probabilistic top-k search algorithms. The extension of probabilistic top-k aggregates algorithms is similar.

Algorithm 1 can be extended to return the l most probable U-Top$k$ answers by changing its termination condition as follows: "terminate upon returning l complete states." Since each reported state is guaranteed to be the most probable among all unreported states, the algorithm correctly computes the l most probable U-Top$k$ answers. Similarly, Algorithm 2 can be extended to return the l most probable U-kRanks answers at each rank $i = 1 \ldots k$ by maintaining, for each rank i, a set T_i of l retrieved tuples with the largest $P_{t,i}$ values among all retrieved tuples. The termination condition at rank i changes to "terminate at rank i when the minimum $P_{t,i}$ value in T_i is not below $\sum_{j<i} Z_j$."

Another generalization of Algorithm 2 is to find distinct U-kRanks query answers. Based on U-kRanks query definition (Definition 2), the same tuple can be the most probable to appear at multiple ranks. Algorithm 2 can be generalized to report distinct answers at different ranks by changing the termination as follows: "terminate at rank i with tuple t if (1) t is not reported as an answer at rank $j < i$; and (2) $P_{t,i} \geq \sum_{j<i} Z_j$."

Top-k sets. Another possible query semantics that arise from the integration of the scoring and probability dimensions is to find the most probable top-k set, that is, the k tuples with the highest probabilities of appearing in the top-k prefix of different worlds. We next show that our U-kRanks algorithm (Algorithm 2) can be used to compute this query. The probability of tuple t to appear in the top-k is $P_{topk}(t) = \sum_{i=1}^{k} P_{t,i}$. Hence, the value of $P_{topk}(t)$ can be computed by summing up the $P_{t,i}$ values (computed at line (15) in Algorithm 2) over all ranks. Let $\mathcal{T}_{topk}$ be the set of k retrieved tuples with the maximum $P_{topk}(t)$ values, and let $U = min(1, \sum_{i=1}^{k} u_i)$, where $u_i = 1 - \sum P_{t,i}$ for any retrieved tuple t. The algorithm terminates when $(min_{\ t \in \mathcal{T}_{topk}}\ P_{topk}(t)) \geq U$. We note, however, that the above extension may not be the most efficient solution for this problem due to the extra overhead incurred in computing $P_{t,i}$ at individual ranks.

Reviews

	Make	Rating
t1	Honda	{7:0.6, 5:0.4}
t2	Toyota	{9:0.3, 8:0.7}
t3	Chevy	{10:1.0}
t4	Nissan	{6:1.0}

(a)

	Make	Rating	Prob
t1.1	Honda	7	0.6
t1.2	Honda	5	0.4
t2.1	Toyota	9	0.3
t2.2	Toyota	8	0.7
t3.1	Chevy	10	1.0
t4.1	Nissan	6	1.0

t1.1 ⊕ t1.2 t2.1 ⊕ t2.2

(b)

Fig. 13. (a) Relation with probabilistic tuple scores (b) Equivalent normalized relation.

Top-k queries with probabilistic scores. Our uncertainty model assumes probabilistic tuples with deterministic scores. An interesting extension is to allow each tuple to have a probability distribution over its possible scores. For example, assume a database of car makes, where each car make is given a set of ratings (scores), and each rating is associated with a probability values reflecting the fraction in a sample of users' population who agree on that rating. Figure 13(a) shows an example of such `Reviews` relation, where each tuple maintains a probability distribution of the ratings given to some car make.

Consider a query that finds the top-*k* rated car makes in the `Reviews` relation. This query can be viewed as an implicit "top-*k* groups" query, where group identifier is the same as tuple id (the "Make" attribute). That is, each tuple is a distinct group whose members are tuple duplicates, with each duplicate assigned one possible score. The uncertainty in tuple scores is thus mapped to tuple membership uncertainty by transforming the original relation to its 1NF, as shown in Figure 13(b). The added exclusiveness rules enforce that no two instances of the same tuple co-exist in the same possible world ($ti.j$ is instance j of tuple ti). Note that this transformation is only possible for discrete score distribution.

Under the above settings, each group has at most one instance in any possible world. Conceptually, this "*top-k aggregate query*" aggregates all possible top-*k* vectors with the same ranking of car makes. For example both $\langle t3.1, t2.1\rangle$ and $\langle t3.1, t2.2\rangle$ map to the same ranking $\langle t3, t2\rangle$, and so their probabilities need to be aggregated. The techniques described in Section 7.2 can be applied to the normalized relation to compute such ranking-aggregate query under our the semantics: U-Top$k$-Agg and U-kRanks-Agg.

9. HANDLING EXPENSIVE PROBABILITY COMPUTATION

In this section we address relaxing the *event probability computation* assumption of our framework. Our previous discussion assumes that the probabilities of query output tuples (or tuple combinations), as computed by the *Rule Engine*, are exact. However, computing the exact probabilities is expensive for some query types [Dalvi and Suciu 2007]. To illustrate, assume an SPJ query plan $\pi^p_x(R \bowtie^p S)$ (cf, Section 2). Assume that the two join results $(r1 \bowtie^p s1)$, and $(r1 \bowtie^p s2)$ have the same value in attribute x. Then, the event associated with a query output tuple t_q is $t_q.e = (r1.e \wedge s1.e) \vee (r1.e \wedge s2.e)$. Computing $\Pr(t_q.e)$

is equivalent to finding the satisfiability ratio of a DNF formula, which is #P-Complete [Valiant 1979]. Specifically, to find the exact value of $\Pr(t_q.e)$, we need to enumerate all possible truth assignments of the events $r1.e$, $s1.e$, and $s2.e$, and sum up the probabilities of the assignments that reduce the DNF formula to *true*.

Ré et al. [2007] describes a simulation algorithm, based on Monte-Carlo method [Karp and Luby 1983], that computes intervals enclosing the precise probabilities with high confidence. These intervals are tightened, when needed, by simulating *random* truth assignments of the involved events. We next describe an extension of our techniques, based on Monte-Carlo simulation, to handle expensive probability computation. We focus our discussion on Algorithm OPTU-Top$k$. The extension of other algorithms is similar.

As described in Section 5.3.1, Algorithm OPTU-Top$k$ maintains a priority queue based on states' probabilities. The algorithm iteratively picks the state at queue top, extends it, and inserts the resulting two states into the queue. The algorithm terminates when the state at queue top is complete. All states are assumed to have exact probabilities as computed by the rule engine. We now consider the case where the probabilities of one or more output tuple events are expensive to compute, and hence they are represented using intervals enclosing the exact probabilities. Since our states are formulated as conjunctions (i.e., CNF formulas) of tuple events, state probability in this case is represented as an interval as well. That is, for a state s_l, an interval $[\underline{\mathcal{P}}(s_l), \overline{\mathcal{P}}(s_l)]$ enclosing the exact $\mathcal{P}(s_l)$ is computed by simulating random truth assignments of the events involved in the CNF formula representing s_l.

Following our strategy of growing a state space guided by probability, two operations can now be applied to each state: (1) tightening state probability intervals by simulating further truth assignments of its tuple events; and (2) extending state (if not complete) with its next unused tuple. An important property that allows early query termination is the following:

PROPERTY 4. *$\overline{\mathcal{P}}(s_l)$ does not increase when s_l is further simulated, or extended.*

Property 4 is true, since simulating further truth assignments in s_l would decrease $\overline{\mathcal{P}}(s_l)$, or keep it the same. Furthermore, extending s_l yields a new CNF formula that is unsatisfiable under any simulated truth assignment that did not satisfy the original CNF formula of s_l. This means that the value of $\overline{\mathcal{P}}(.)$ for extended states from s_l is always bounded by $\overline{\mathcal{P}}(s_l)$. We can thus report query answers by bounding the probabilities of both complete and incomplete states, which allows avoiding the costly computation of exact probabilities.

Moreover, we can further tighten the upper-bounds of states' probabilities by exploiting their correlation under possible worlds semantics. Specifically, at any point, the summation of the $\underline{\mathcal{P}}(.)$ of all maintained states is the total probability that is already assigned to some (partial) answers in the space. The remainder, that is, $(1-\sum \underline{\mathcal{P}}(s_l))$, is a probability margin that any state can further obtain. A tighter probability upper-bound for state s_l, denoted $TU(s_l)$, is thus computed as follows: $TU(s_l) = min(\overline{\mathcal{P}}(s_l), 1 - \sum_{\acute{s} \neq s_l} \underline{\mathcal{P}}(\acute{s}))$.

Based on the above discussion, a general U-Top$k$ query processing algorithm that handles expensive probability computation is described in the following:

(1) Keep a priority queue for all visited states based on $TU(.)$.
(2) At any step, if there exists a *complete* state s_k^* at queue top, where $\underline{\mathcal{P}}(s_k^*) > TU(s)$ for any other state s, terminate with s_k^*.
(3) Simulate the top queue state using a new random assignment of its variables, while updating the queue at each simulation step, until the state s_l^* at queue top satisfies the condition $\underline{\mathcal{P}}(s_l^*) > TU(s)$ for any other state s.
(4) Extend s_l^* using the next tuple event, and replace it in the queue with the two resulting states. Goto step (3).

Based on our previous discussion in Section 5.3.1, and Theorem 2, it can be shown that the above algorithm minimizes the number of visited states, since it does not extend a state unless it is indeed the state with the highest probability.

10. EXPERIMENTS

Our experiments are conducted on a 3GHz Pentium IV PC with 1 GB of memory, running Debian GNU/Linux3.1. We built our framework on top of RankSQL [Li et al. 2005], which provides rank-awareness and group-awareness support. We used synthesized datasets generated by R [R-project]. Space navigation algorithms, and a rule engine prototype were implemented in C. We created a customized set of rules for each dataset to control the generation of possible worlds. Section 10.1 describes the experiments we conducted to evaluate our techniques for computing probabilistic top-k queries, while Section 10.2 is our experimental evaluation for probabilistic top-k aggregates techniques.

10.1 Probabilistic Top-k Queries

The main performance metrics to evaluate probabilistic top-k query processing techniques are (1) query execution time, and (2) tuple scan depth (the number of consumed tuples from $\mathcal{D}$). In all experiments we used rank-aware plans to efficiently report tuples in score order. We emphasize, however, that our techniques are transparent from the underlying source of score-ranked tuples.

10.1.1 *The Naïve Approach.* We illustrate the infeasibility of applying the naive 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 the average, one 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 space.

10.1.2 *Effect of the Distribution of Tuples Probabilities.* We evaluate here the effect of the distribution of tuples probabilities on execution time and

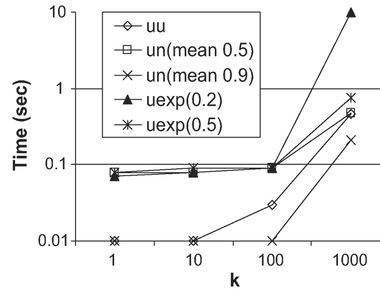

Fig. 14. `IndepU-Top`k time (different distributions).

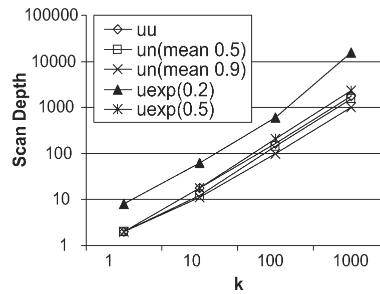

Fig. 15. `IndepU-Top`k depth (different distributions).

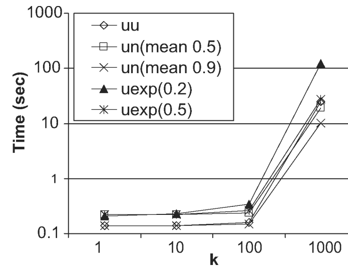

Fig. 16. `IndepU-`k`Ranks` time (different distributions).

scan depth. We used datasets with the following (score,probability) distribution pairs: (1) *uu*: score and probability are uniformly distributed; (2) *un (mean x)*: score is uniformly distributed and probability is normally distributed with mean x, where $x = 0.5$ or 0.9, and standard deviation 0.2, and (3) *uexp (x)*: score is uniformly distributed and probability is exponentially distributed with mean x, where $x = 0.2$ or 0.5.

Figures 14 and 15 show the time and scan depth of `IndepU-Top`k, respectively, while Figures 16 and 17 show the time and scan depth of `IndepU-`k`Ranks`, respectively, for k values up to 1000. The best case for both algorithms is to frequently find highly probable tuples 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) where the exponential rate of decay in

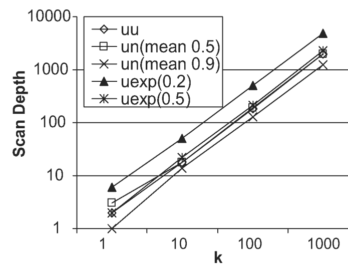

Fig. 17. `IndepU-`k`Ranks` depth (different distributions).

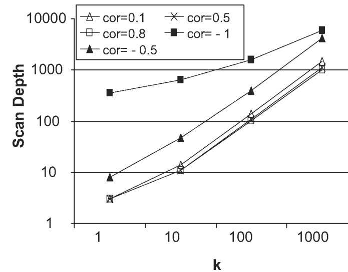

Fig. 18. `IndepU-Top`k (correlations).

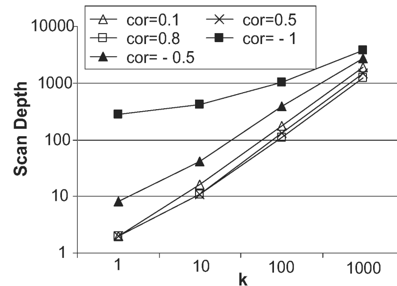

Fig. 19. `IndepU-`k`Ranks` (correlations).

probabilities results in a 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 are more expensive operations in `IndepU-`k`Ranks` than `IndepU-`k`Ranks`.

10.1.3 *Score-Probability Correlations.* We evaluate the effect of score-probability correlations. We generated bivariate Gaussian data on score and probability dimensions, and controlled the correlation coefficient by adjusting the bivariate covariance matrix. Positive correlations result in large savings, since in this case highly-scored tuples have high probabilities, which allows reducing the number of needed-to-see tuples to compute query answers. Figures 18 and 19 show the effect of the correlation coefficient on the scan

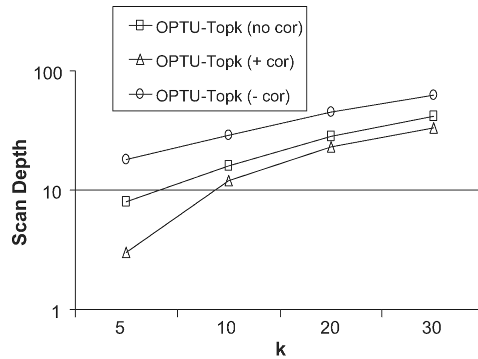

Fig. 20. OPTU-Top$k$ (depth).

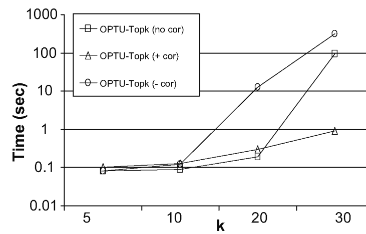

Fig. 21. OPTU-Top$k$ (time).

depth of IndepU-Top$k$ and IndepU-kRanks, respectively. Increasing the correlation coefficient from 0.1 to 0.8 reduces the scan depth of IndepU-Top$k$ and IndepU-kRanks by an average of 20% and 26%, respectively. Negative correlation degrades the performance since it leads to consuming more tuples. For example, decreasing the correlation coefficient from -0.5 to -1 results in an increase, with an average of 1.5 orders of magnitude, in scan depth for IndepU-Top$k$, and 1 order of magnitude for IndepU-kRanks. The effect on execution time is similar.

10.1.4 *Evaluating Algorithms Performance.* We evaluate the performance of OPTU-Top$k$. We used databases of exclusive tuples with uncorrelated, positively correlated, and negatively correlated score and probability values. Figures 20 and 21 show the scan depth and execution time of the OPTU-Top$k$ algorithm. The execution time is under 100 seconds for values of k reaching 30. The time is consumed 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$k$ increases by an average of 1 order of magnitude when going from positively to negatively correlated data. This can be explained based on the fact that for positively correlated data, highly-probable states are reached quickly after retrieving a small number of tuples, while for negatively correlated data more tuples need to be retrieved before concluding an answer, which leads to materializing more states.

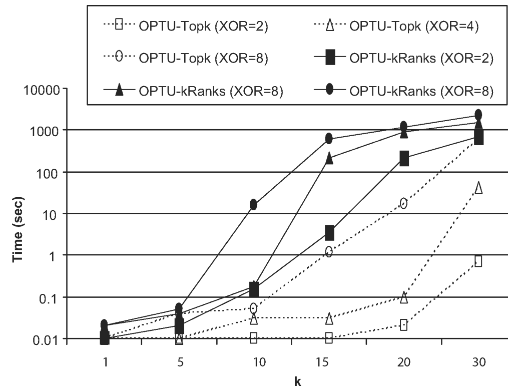

Fig. 22. Rule set complexity.

10.1.5 *Rule Set Complexity.* We evaluate the effect of potential complexity of model rules on the performance. Since the study of efficient rule evaluation techniques is beyond the scope of this article, we implemented a rule engine prototype that computes the probabilities of partial states under tuple exclusiveness. We experimented with different rule sets with different *XOR* degrees; which is the number of tuples that are exclusive with a given tuple. Figure 22 shows the execution times of `OPTU-Top`k and `OPTU-`k`Ranks` with different *XOR* degrees. Increasing the *XOR* degree generally results in increasing the execution time, with an average of one order of magnitude when going from $XOR = 2$ to $XOR = 4$, or from $XOR = 4$ to $XOR = 8$ at the same value of k. Increasing the *XOR* degree 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 currently processed states, which leads to larger computational overhead.

10.2 Probabilistic Top-k Aggregate Queries

In this section we give the experimental evaluation of our processing techniques for top-k aggregate queries and top-k queries with probabilistic tuple scores. Probabilistic tuple scores are represented as multiple exclusive tuple instances associated with the same tuple identifier. We assume the independence of tuples coming from different groups to simplify computing the probabilities of global states. The intra- and intergroup aggregation layers are implemented on top of a rank-aware and group-aware RDBMS [Li et al. 2005, 2006]. Our experimental study does not take into consideration tuple retrieval cost, which is transparent to our techniques.

We study the effect of k, the number of groups, and the group tuple count, on the performance of our techniques. The performance metrics are query execution time, the number of retrieved tuples, the number of generated group states, and the pruning power of our space navigation techniques, where the pruning power is estimated using the nonmaterialized portion of the possible worlds space.

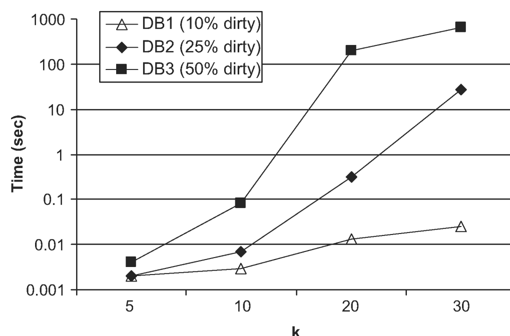

Fig. 23. Effect of k on execution time.

10.2.1 *Effect of Changing k Values.* We evaluate the performance of our techniques with different k values. The problem space grows exponentially when increasing k. Specifically, for n groups the total number of possible U-Top$k$-Agg query answers is all the possible group permutations of length k, which is in $O(n^k)$. Increasing the value of k results in exponential increases in query execution time and the number of processed tuples and groups. However, our space pruning and efficient navigation algorithms allow for graceful performance for typical values of k, as we show in this experiment.

To study the effect of increasing k, we generated three different datasets DB1, DB2, and DB3 with the following configurations:

—the size of each database is 1 million record;
—group tuple count within each database is uniformly distributed on [1,50] tuples;
—the *dirtiness* ratio of DB1, DB2, and DB3 are 10%, 25%, 50%, respectively, where a tuple is considered *dirty* if its membership probability is less than 1.

Figure 23 shows query execution time when increasing the value of k from 5 to 30 for the different databases. Query execution time increases by an average of 3 orders of magnitude when the value of k increases from 5 to 30. This behavior is expected because of the nature of our problem space. The maximum query execution time is around 600 seconds in the worst case (DB3 at $k = 30$).

The dirtiness ratio contributes to the increase in query execution times, as shown in Figure 23. Query execution time increases by an average of 1.5 orders of magnitude between DB1 and DB2, and by an average of 2.2 orders of magnitude between DB2 and DB3. Smaller execution times in cleaner datasets are attributed to how space is navigated. When a state is extended with a probabilistic tuple, a new state needs to be maintained. On the other hand, when a state is extended with a certain tuple, no new states are generated. Encountering certain tuples frequently while processing U-Top$k$-Agg queries reduces the materialized space size, and consequently contributes to smaller query execution time.

Figure 24 shows the effect of increasing k on the number of retrieved tuples. In general, increasing k does not contribute to large increases in the number

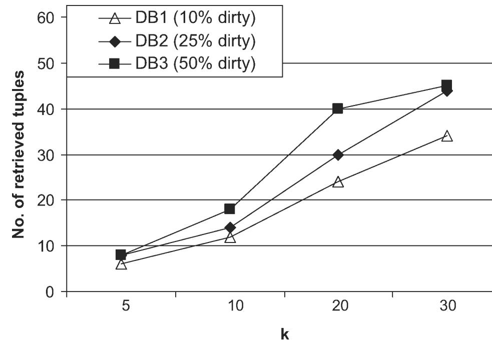

Fig. 24. Effect of k on retrieved tuples.

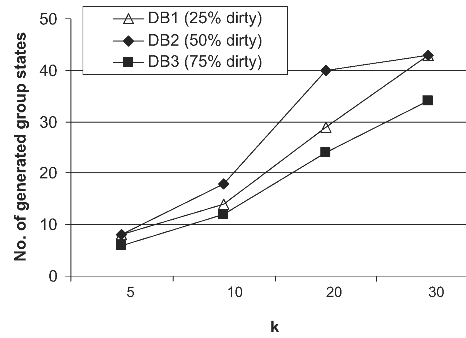

Fig. 25. Effect of k on generated group states.

of retrieved tuples. The number of retrieved tuples for the three datasets was comparable for the same k values. The interpretation is that a small number of tuples generates a huge number of possible top-k vectors. Tuples that are located deep inside the score-ordered tuple stream have negligible chances of contributing to the top-k answer, since their existence in an answer is conditioned on the nonexistence of most of the preceding tuples.

Figure 25 shows the number of group states generated from the intragroup aggregation layer for each dataset at different k values. Figure 25 shows that the number of consumed states is small. Specifically, the number of generated states doubled by 4 times, in the worst case of DB3, when the value of k doubled by 6 times. Our score-ordered state traversal enables the aggregation techniques to generate highly-probable query answers based on the seen states, and hence no useless states are generated.

10.2.2 *Varying the Number of Groups.* We evaluate the effect of varying the number of groups on the performance of our techniques. Figure 26 shows the execution time for U-Top5-Agg queries with a number of groups ranging from 100,000 to 500,000 and a fixed group size of 5 exclusive tuples. Increasing the number of groups does not dramatically degrade the performance. The maximum query execution time is 80 seconds for the case of 500,000 groups. This can be interpreted based on how our processing framework picks groups/group

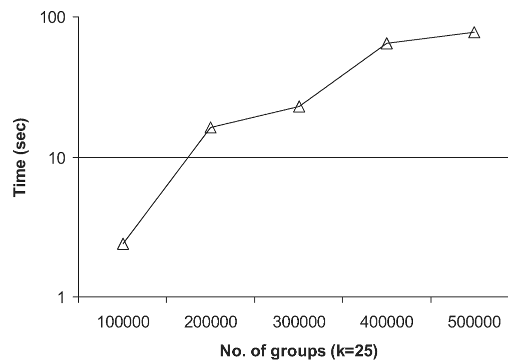

Fig. 26. Effect of number of groups on execution time.

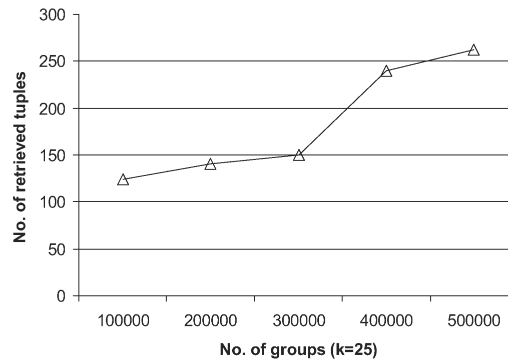

Fig. 27. Effect of number of groups on retrieved tuples.

states to process based on both score and probability. Useless groups/group states that do not contribute to query answer are not processed based on the upper bounds maintained on scores and probabilities. Our framework always explores the groups/states with the highest chances of contributing to query answer. Increasing the number of groups, however, involves extra processing in the intragroup aggregation layer to produce the next group state in score order across all groups, which results in relatively small increase in query execution time, as shown in Figure 28.

The number of retrieved tuples is not largely affected by the increase in the number of groups, as shown in Figure 27. The number of retrieved tuples increased by 129% when the number of groups increased from 100,000 to 500,000. This is due to the fact that a limited number of groups contribute to query answer. Our space search methods probe a small number of groups based on probability guidance, which results in retrieving a small number of tuples.

10.2.3 *Varying the Group Size.* Varying the group size results in changing the number of group worlds, which influences our techniques as follows. For groups of small number of worlds, the probability of each group world is considerable. This allows the intragroup aggregation layer to generate group states with high probability early during its processing. Obtaining highly-probable

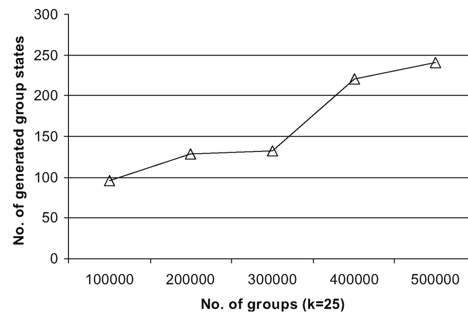

Fig. 28. Effect of number of groups on group states.

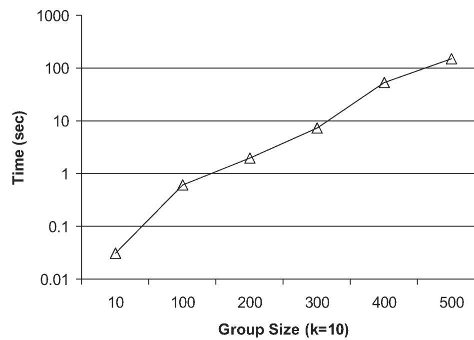

Fig. 29. Effect of group size on execution time.

states early from the intragroup aggregation layer enables the processing framework to conclude answers quickly. On the other hand, when the number of group worlds increases, the probability of each world is small. The generated group states, which aggregate the probabilities of many group worlds, would therefore have small probabilities as well. This forces the intergroup aggregation layer to consume more group states before concluding an answer.

Figure 29 shows query execution time for datasets of 1000 groups, and the number of group tuples ranges from 10 to 500. The variation in the number of group worlds is simulated by customizing the rule set to control the number of possible group worlds. The sensitivity of our techniques to group size leads to increasing query execution time from 0.05 seconds to 150 seconds when group size increases from 10 to 500 worlds. Figures 30 and 31 show similar behavior for the number of retrieved tuples, and the number of generated states, respectively, which both increased by almost one order of magnitude when group size increases from 10 to 500 worlds.

10.2.4 *Evaluating the Pruning Power of Space Navigation.* We evaluate the performance of our techniques with respect to the savings in the materialized space. Table I shows the results obtained with different parameter configurations. Each result is obtained from the average of 10 independent runs with the same parameter configurations. In all runs, datasets with groups of exclusive tuples are used to allow computing the exact size of the answer space.

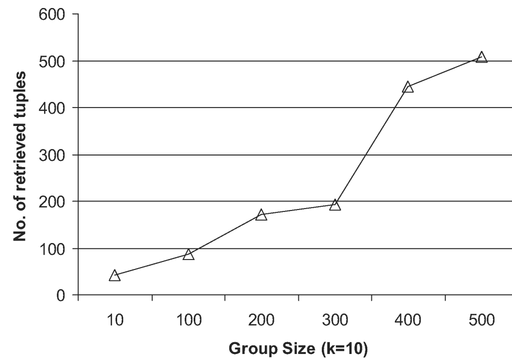

Fig. 30. Effect of group size on retrieved tuples.

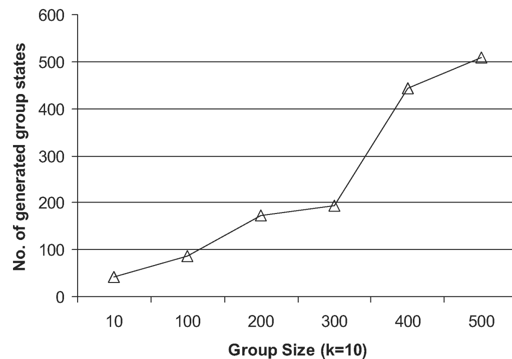

Fig. 31. Effect of group size on generated states.

For a database of n groups with m exclusive tuples within each group, the number of possible worlds is m^n, while the number of possible U-Top$k$-Agg query answers is in $O(n^k)$, which corresponds to all possible k-length permutations of n groups. Although the problem space is exponentially large, the number of materialized intergroups states is manageable for typical k values. The ratio between the number of intergroups states to the total number of possible answers is negligible, which reflects the efficiency of our techniques in pruning the answer space and the applicability of our methods in practical settings.

11. RELATED WORK

Our study is mainly related to proposals in two large areas:

(1) *Probabilistic query processing*. Several recent research projects address query processing aspects in uncertain databases. The Trio system [Widom 2005; Sarma et al. 2006; Benjelloun et al. 2006] introduced *working models* to capture uncertainty at different levels by relating uncertainty with lineage and leveraging existing DBMSs capabilities for uncertain data management. The ORION project [Cheng et al. 2007] 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

Table I. Materialized Intergroups States for Different Configurations

Group Size	Groups	k	Space Size	Inter-groups States	Inter-groups States / Space Size
1000	500	5	500^5	14,223	4.6E-10
2000	500	5	500^5	14,267	4.5E-10
5000	500	5	500^5	16,319	5.2E-10

possible worlds semantics under membership uncertainty and generation rules. The CONQUER project [Fuxman et al. 2005; Andritsos et al. 2006] introduced query rewriting algorithms to extract clean and consistent answers from unclean databases under possible worlds semantics and proposed methods to derive the probabilities of uncertain data items.

Most of the proposed probabilistic query processing methods mainly assume Boolean queries, and so they are insufficient to compute probabilistic top-k queries efficiently. The reason is twofold: (1) the proposed algorithms are primarily tuple-based, that is, they compute probabilities for individual output tuples, while probabilistic top-k queries are after tuple vectors; and (2) conventional score-based ranking criteria are not considered in any of these algorithms. Ré et al. [2007] address top-k queries in probabilistic databases based on the probability dimension only, where a top-k query reports the k most probable tuples in query output. The proposed technique computes 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, an integrated solution that combines both scores and probabilities as two interacting ranking dimensions does not exist.

Probabilistic aggregate operators have been addressed in Ross et al. [2005], with the objective of computing the probability distribution of the aggregate value over the whole probabilistic relation. The authors proposed a method to group the relation's possible worlds into buckets with the same aggregate value and solve a linear program over each bucket to derive its probability bounds. The algorithm has a general exponential complexity. Polynomial approximation algorithms are proposed to only process the worlds specified by a selection strategy, for example, filtering out worlds whose probabilities are below a certain threshold. Jayram et al. [2007] study the semantics and processing of aggregate queries in probabilistic data streams. The adopted semantics is reporting the expected aggregate value, based on aggregate probability distribution. The authors proposed efficient data stream algorithms that estimate aggregate values within accuracy bounds. None of these works address formulating and processing ranking-aggregate queries.

The work in Yi et al. [2008] uses our query definitions, and presents efficient algorithms to handle special cases of our general search algorithms, where only the most probable U-Top$k$ or U-kRanks query answer is required, and tuples are either independent or mutually exclusive.

(2) *Rank/group-aware query processing*. Recent proposals in rank-aware query processing [Ilyas et al. 2004; Li et al. 2005], focus on embedding rank-awareness in query operators to allow for early-out of top-k queries. While these approaches provide a good basis for ranking probabilistic data in general,

they are not explicitly designed to treat probability as an additional ranking dimension.

Group-awareness can be supported by several existing methods. In Goldstein and Larson [2001], view-matching algorithms are proposed for group-by queries to determine whether a query can be fully or partially answered using precomputed materialized views. Matching views can provide statistics on different groups, for example, group tuple count. Additionally, frequently-used grouping columns, based on query workloads, can be indexed in the views to provide efficient means to incrementally access group tuples.

Datacubes [Gray et al. 1997] are widely-adopted in OLAP environment. A datacube aggregates a measure attribute based on different subsets of dimensional attributes. Given the potentially huge size of datacubes, different materialized views, summaries, and index structures are usually used to make extracting information from datacubes more efficient. Such methods can be directly applied to materialize statistics, including group tuple count, for arbitrary groups; or at least making obtaining such statistics relatively cheap.

In Hellerstein et al. [1997], an index striding method is proposed to draw random tuples from each group, allowing progressive improvement in the accuracy of the computed aggregates. When grouping attributes are indexed, the index striding method opens multiple cursors on the index, one per group, such that a cursor is advanced whenever a tuple is retrieved. When multikey indexes are available, such that the grouping attribute is the primary key and the scoring attribute is the secondary key, index striding can provide incremental tuple access from each group in score order [Li et al. 2006].

The previous methods can be used to provide an interface that allows us to incrementally access tuples in specific groups based on score order, in addition to giving bounds on group tuple count. This allows scheduling group access to minimize the number of needed-to-see tuples from each group, while computing the top-k groups [Li et al. 2006] (we build on these results in our intragroup aggregation layer in Section 7.2.1). However, these methods cannot be used solely when group aggregates are probabilistically defined.

12. CONCLUSIONS

To the best of our knowledge, this article presents the first work that studies formulating and processing probabilistic top-k and top-k aggregate queries. Our work provides insights on the interplay between the score and probability dimensions in these query types and presents efficient mechanisms to aggregate probabilities across possible worlds.

We introduced new probabilistic formulations for top-k and ranking-aggregate queries under possible worlds semantics. We modeled our problem as a state space search, and described several query processing algorithms with guarantees on the number of accessed tuples and the size of materialized search space. Our processing methods integrate tuple retrieval, ranking, grouping, and uncertainty management in the same framework, while leveraging existing indexing and query processing techniques in RDBMSs. We introduced several extensions to our techniques to handle other problem variants and query types.

Our experimental study illustrates the efficiency of our techniques in different practical settings.

REFERENCES

ABITEBOUL, S., KANELLAKIS, P., AND GRAHNE, G. 1987. On the representation and querying of sets of possible worlds. *SIGMOD Rec. 16,* 3, 34–48.

ANDRITSOS, P., FUXMAN, A., AND MILLER, R. J. 2006. Clean answers over dirty databases: A probabilistic approach. In *Proceedings of the 22nd ICDE International Conference on Data Engineering*. 30.

BENJELLOUN, O., SARMA, A. D., HALEVY, A., AND WIDOM, J. 2006. ULDBs: Databases with uncertainty and lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*. 953–964.

BHATTACHARYA, I., GETOOR, L., AND LICAMELE, L. 2006. Query-time entity resolution. In *Proceedings of the 12th ACM SIGKDD International Conference*. ACM, New York, 529–534.

CHENG, R., KALASHNIKOV, D. V., AND PRABHAKAR, S. 2007. Evaluation of probabilistic queries over imprecise data in constantly-evolving environments. *Inform. Syst. 32,* 1, 104–130.

DALVI, N. AND SUCIU, D. 2007. Efficient query evaluation on probabilistic databases. *VLDB J. 16,* 4, 523–544.

FAGIN, R., LOTEM, A., AND NAOR, M. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci. 66,* 4, 614–656.

FUXMAN, A., FAZLI, E., AND MILLER, R. J. 2005. ConQuer: Efficient management of inconsistent databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. ACM, New York, 155–166.

GOLDSTEIN, J. AND LARSON, P.-A. 2001. Optimizing queries using materialized views: A practical, scalable solution. *SIGMOD Rec. 30,* 2.

GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. 1997. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining Knowl. Disc. 1,* 1, 29–53.

HART, P. E., NILSSON, N. J., AND RAPHAEL, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. 4,* 2, 100–107.

HE, B. AND CHANG, K. C.-C. 2006. Automatic complex schema matching across Web query interfaces: A correlation mining approach. *ACM Trans. Datab. Syst. 31,* 1, 346–395.

HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. 1997. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 171–182.

HERNANDEZ, M. A. AND STOLFO, S. J. 1998. Real-world data is dirty: Data cleansing and the merge/purge problem. *J. Data Mining Knowl. Disc. 2,* 1, 9–37.

HRISTIDIS, V., KOUDAS, N., AND PAPAKONSTANTINOU, Y. 2001. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*. ACM, New York, 259–270.

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2003. Supporting top-k join queries in relational databases. In *Proceedings of the 29th International Conference on Very Large Data Base (VLDB)*. 754–765.

ILYAS, I. F., SHAH, R., AREF, W. G., VITTER, J. S., AND ELMAGARMID, A. K. 2004. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, 203–214.

IMIELIŃSKI, T. AND WITOLD LIPSKI, J. 1984. Incomplete information in relational databases. *J. ACM 31,* 4, 761–791.

JAYRAM, T. S., KALE, S., AND VEE, E. 2007. Efficient aggregation algorithms for probabilistic data. In *Proceedings of the 18th Annual ACM-SIAM SODA Conference*. 346–355.

KARP, R. AND LUBY, M. 1983. Monte-carlo algorithms for enumeration and reliability problems. In *Proceedings of the 24th STOC Conference*. 56–64.

LAKSHMANAN, L. V. S., LEONE, N., ROSS, R., AND SUBRAHMANIAN, V. S. 1997. ProbView: A flexible probabilistic database system. *ACM Trans. Datab. Syst. 22,* 3, 419–469.

LI, C., CHANG, K. C., AND ILYAS, I. F. 2006. Supporting ad-hoc ranking aggregates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. ACM, New York, 61–72.

LI, C., CHANG, K. C., ILYAS, I. F., AND SONG, S. 2005. RankSQL: Query algebra and optimization for relational top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Data (SIGMOD'05)*. ACM, New York, 131–142.

MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2003. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York, 491–502.

NATSEV, A., CHANG, Y., SMITH, J. R., LI, C., AND VITTER, J. S. 2001. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th Conference on Very Large Data Bases (VLDB)*. 281–290.

R-PROJECT. The R project for statistical computing: www.r-project.org.

RÉ, C., DALVI, N. N., AND SUCIU, D. 2007. Efficient top-k query evaluation on probabilistic Data. In *Proceedings of the 23rd ICDE Conference*. 886–895.

ROSS, R., SUBRAHMANIAN, V. S., AND GRANT, J. 2005. Aggregate operators in probabilistic databases. *J. ACM 52,* 1, 54–101.

SARMA, A. D., BENJELLOUN, O., HALEVY, A., AND WIDOM, J. 2006. Working models for uncertain data. In *Proceedings of the 22nd ICDE Conference*. 7.

SEN, P. AND DESHPANDE, A. 2007. Representing and querying correlated tuples in probabilistic databases. In *Proceedings of the 23rd ICDE Conference*. 596–605.

SOLIMAN, M. A., ILYAS, I. F., AND CHANG, K. C. 2007a. Top-k query processing in uncertain databases. In *Proceedings of the 23rd ICDE Conference*. 896–905.

SOLIMAN, M. A., ILYAS, I. F., AND CHANG, K. C.-C. 2007b. URank: Formulation and efficient evaluation of top-k queries in uncertain databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 1082–1084.

VALIANT, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput. 8,* 3, 410–421.

WIDOM, J. 2005. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the 2nd Biennial CIDR Conference*. 262–276.

YI, K., LI, F., SRIVASTAVA, D., AND KOLLIOS, G. 2008. Efficient processing of top-k queries on uncertain databases. In *Proceedings of the 24th ICDE Conference*. 1406–1408.

Received July 2007; revised January 2008; accepted March 2008