Ranked Retrieval in Uncertain and Probabilistic Databases

by

Mohamed A. Soliman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

Abstract

Ranking queries are widely used in data exploration, data analysis and decision making scenarios. While most of the currently proposed ranking techniques focus on deterministic data, several emerging applications involve data that are imprecise or uncertain. Ranking uncertain data raises new challenges in query semantics and processing, making conventional methods inapplicable. Furthermore, the interplay between ranking and uncertainty models introduces new dimensions for ordering query results that do not exist in the traditional settings.

This dissertation introduces new formulations and processing techniques for ranking queries on uncertain data. The formulations are based on marriage of traditional ranking semantics with possible worlds semantics under widely-adopted uncertainty models. In particular, we focus on studying the impact of tuple-level and attribute-level uncertainty on the semantics and processing techniques of ranking queries.

Under the tuple-level uncertainty model, we introduce a processing framework leveraging the capabilities of relational database systems to recognize and handle data uncertainty in score-based ranking. The framework encapsulates a state space model, and efficient search algorithms that compute query answers by lazily materializing the necessary parts of the space.

Under the attribute-level uncertainty model, we give a new probabilistic ranking model, based on partial orders, to encapsulate the space of possible rankings originating from uncertainty in attribute values. We present a set of efficient query evaluation algorithms, including sampling-based techniques based on the theory of Markov chains and Monte-Carlo method, to compute query answers.

We build on our techniques for ranking under attribute-level uncertainty to support rank join queries on uncertain data. We show how to extend current rank join methods to handle uncertainty in scoring attributes. We provide a pipelined query operator implementation of uncertainty-aware rank join algorithm integrated with sampling techniques to compute query answers.

Acknowledgements

First and foremost, my deepest gratitude to my advisor, Prof. Ihab Ilyas whose sincerity and encouragement I shall never forget. Prof. Ilyas has been my inspiration throughout the years of my graduate studies. His dedication, enthusiasm, and hard work have set the example for me on how to be a sincere, critical, and respectful researcher. Prof. Ilyas was always challenging me, encouraging me, and expecting more from me, in every research endeavor. He was always there for me whenever I needed help. I have learned a lot from him on both personal and academic levels, and I shall be forever indebted for everything he has given me. As I proceed in my career, Prof. Ilyas will always be my teacher, advisor, and dear friend.

I am deeply grateful to my thesis committee members, Prof. Shai Ben-David, Prof. R. Wayne Oldford, Prof. Ken Salem, and Prof. Gerhard Weikum. Being able to discuss my research with such outstanding group of researchers has been the greatest honor of my career. I would like to thank my thesis committee members for their invaluable feedback and suggestions that definitely made this dissertation better.

This dissertation would not have been possible without the support of my lovely wife, Hadir, who has been there for me every step on the way. Her love and continuous support contributed a lot to my success. I enjoyed sharing every happy moment in my life with her, and she was always my first resort whenever I am tired or frustrated.

I would like to thank my parents, Fathi and Nagat, for their ongoing and endless love. My parents sacrificed a lot to make me a successful person. They have given everything without expecting anything in return. I owe them a lot for what they have done for me, and I hope that I can be always a source of their happiness.

Above all, I thank God for all His blessings and mercies. I pray to God to guide me to the right path in this life, bless my family, and make me a useful member of the community.

Dedication

To Hadir, the love of my life. Without your love and support, this dissertation would not have been possible.

Table of Contents

List of Figures

Chapter 1

Introduction

Ranking queries are widely used in data exploration, data analysis and decision making scenarios. The objective of ranking queries (also referred to as top-k queries) is to report the top ranked query results based on scores computed by a given scoring function (e.g., a function defined on one or more database columns). A scoring function induces a unique total order on query results, where score ties are usually resolved using a deterministic tie-breaking criterion.

While most of the currently proposed ranking techniques focus on deterministic data, several emerging applications (e.g., Web mashups, location-based services, and sensor data management) involve data that are imprecise or uncertain. Dealing with data uncertainty by removing uncertain values is not desirable in many settings. For example, there could be too many uncertain values in the database (e.g., readings of sensing devices that become frequently unreliable under high temperature). Alternatively, there could be only few uncertain values in the database but they are involved in data entities that closely match query requirements. Dropping such uncertain values may lead to inaccurate or incomplete query answers. For these reasons, modeling and processing uncertain data have been the focus of many recent studies [56, 9, 20].

With data uncertainty, the semantics of ranking queries become unclear. For example, reporting a top ranked query result may not depend only on its computed score, but also on the potential uncertainty of that result as well as the scores and uncertainty of other query results. Ranking and uncertainty models interplay to decide on meaningful interpretation of ranking queries in this context.

Studying the interplay between ranking and uncertainty models is motivated by the need for ranking support in environments that generate or process uncertain data. For example, in the context of the Web, information extraction techniques are used to extract instances of entities, e.g., organization and person names. The imperfection of extraction tools and the inherent ambiguity of unstructured text introduce uncertainty in extracted information. Ranking objects with such uncertain 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 a probabilistic model, based on readings history and the dependencies among different sensors. Many applications can benefit from ranking sensor data. For example, aggregating and ranking sensor readings to find the top ranked locations based on temperature. Evaluating these queries needs to take into account both the uncertainty in the underlying data, and the ranking requirements to be imposed on query results.

Under the relational data model, we focus on studying the implications of two different types of data uncertainty on the semantics and processing techniques of ranking queries:

- *Tuple level uncertainty.* Tuples may belong to the database with less than absolute confidence. A widely-used model to capture this type of uncertainty is representing tuples as probabilistic events, and model the database as a joint distribution defined on these events.

- *Attriubute level uncertainty.* Tuple attributes may have uncertain values. A widely-used model to capture this type of uncertainty is representing tuple attributes as probability distributions defined on discrete or continuous domains.

This chapter starts by presenting real-world examples motivating the need for

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)

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

(b)

Figure 1.1: (a) A relation with tuple uncertainty (b) Example ranking query

supporting ranking queries under tuple-level uncertainty (Section 1.1), and attribute-level uncertainty (Section 1.2). We then list the challenges raised by the integration of ranking and uncertainty models in Section 1.3. We summarize our contributions and present the outline of this dissertation in Section 1.4.

1.1 Tuple Level Uncertainty

We use the following example to illustrate the challenges involved in formulating and computing ranking queries under tuple level uncertainty:

Example 1 *In a traffic-monitoring system, radars detect cars' speeds automatically, while car identification, e.g., 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, or plate number images that are not clear enough to identify the car precisely. Figure 1.1(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.*

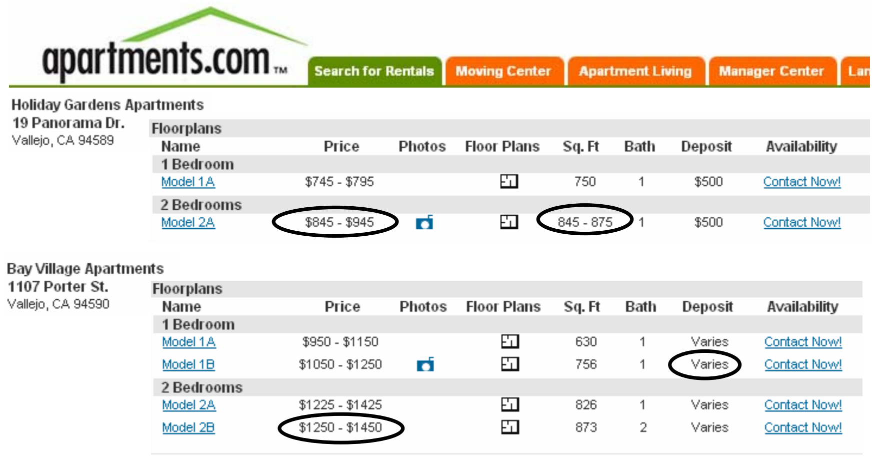
Figure 1.1(b) shows an example ranking query to be evaluated on the `Readings` relation in Example 1. The given query is a Top-k Selection Query, in which scores are computed for base tuples, and the k query results with the highest scores are reported. The query requests the top-k speeding cars in one hour interval, which can be used, e.g., in an accident investigation scenario.

Although tuple scores (the *Speed* values) are given as deterministic values, the tuples themselves are uncertain. 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, may eliminate uncertainty and lose valuable information that can be used to get more meaningful answers conforming with probabilistic query models (we elaborate on this point in Section 2.2).

1.2 Attribute Level Uncertainty

Uncertainty in attribute values induces uncertain scores when computing ranking queries. In contrast to conventional ranking settings, where a total order on query results is induced by the given scoring function, score uncertainty induces a *partial order* on the underlying tuples, where multiple rankings are valid (we formally define partial orders in Section 3.2).

To illustrate, consider Figure 1.2 which shows a snapshot of actual search results reported by `apartments.com` for a simple search for available apartments to rent. The shown search results include several uncertain pieces of information. For example,



Holiday Gardens Apartments
19 Panorama Dr.
Vallejo, CA 94589

Floorplans

Name	Price	Photos	Floor Plans	Sq. Ft	Bath	Deposit	Availability
1 Bedroom							
Model 1A	$745 - $795		▣	750	1	$500	Contact Now!
2 Bedrooms							
Model 2A	$845 - $945	📷	▣	845 - 875	1	$500	Contact Now!

Bay Village Apartments
1107 Porter St.
Vallejo, CA 94590

Floorplans

Name	Price	Photos	Floor Plans	Sq. Ft	Bath	Deposit	Availability
1 Bedroom							
Model 1A	$950 - $1150		▣	630	1	Varies	Contact Now!
Model 1B	$1050 - $1250	📷	▣	756	1	Varies	Contact Now!
2 Bedrooms							
Model 2A	$1225 - $1425		▣	826	1	Varies	Contact Now!
Model 2B	$1250 - $1450		▣	873	2	Varies	Contact Now!

Figure 1.2: Uncertain data in search results

some apartment listings do not explicitly specify the deposit amount. Other listings mention apartment rent and area as ranges rather than single values.

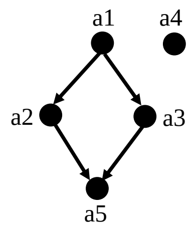
We illustrate the challenges involved in ranking with uncertain scores using the following simple example of the previous apartment search scenario.

Example 2 *Assume an apartment database. Figure 1.3(a) gives a snapshot of the results of some query posed against such database. Assume that we are interested in ranking query results using a function that scores apartment records based on rent (the cheaper the apartment, the higher the score). Since the rent of apartment* $a2$ *is specified as a range, and the rent of apartment* $a4$ *is unknown, the scoring function assigns a range of possible scores to* $a2$*, while the full score range* $[0 - 10]$ *is assigned to* $a4$*.*

Figure 1.3(b) depicts a diagram for the partial order induced by apartment scores. Disconnected nodes in the diagram indicate the incomparability of their corresponding records. Due to the intersection of score ranges, $a4$ is incomparable to all other records, and $a2$ is incomparable to $a3$.

AptID	**Rent**	**Score**
a1	$600	9
a2	[$650-$1100]	[5-8]
a3	$800	7
a4	*negotiable*	[0-10]
a5	$1200	4

(a)



(b)

Linear Extensions
<a1,a4,a2,a3,a5>
<a1,a2,a3,a5,a4>
<a1,a3,a2,a5,a4>
<a1,a4,a3,a2,a5>
<a1,a2,a3,a4,a5>
<a1,a2,a4,a3,a5>
<a1,a3,a2,a4,a5>
<a1,a3,a4,a2,a5>
<a4,a1,a2,a3,a5>
<a4,a1,a3,a2,a5>

(c)

Figure 1.3: Partial order for tuples with uncertain scores

A simple approach to compute a ranking based on the above partial order is to reduce it to a total order by replacing score ranges with their expected values. However, ranking based on expected values can result in unreliable ranking, as we show in Section 3.2. Moreover, expected values are known to be sensitive to the existence of outliers.

Another possible ranking query on partial orders is finding the skyline (i.e., the non-dominated objects [16]). An object is non-dominated if, in the partial order diagram, the object's node has no incoming edges. In Example 2, the skyline objects are $\{a1, a4\}$. The number of skyline objects can vary from a small number (e.g., Example 2) to the size of the whole database. Furthermore, skyline objects may not be equally good and, similarly, dominated objects may not be equally bad. For example in Figure 1.3(b), $a4$ dominates no objects, while $a1$ dominates all objects except $a4$. However, both $a1$ and $a4$ are skyline objects. This shows that there can be a considerable difference in the dominance power of skyline objects. A user may want to compare objects' relative orders in different data exploration scenarios. Current proposals [13, 69] have demonstrated that there is no unique way to distinguish or rank the skyline objects.

A different approach to rank the objects involved in a partial order is inspecting the space of possible rankings that conform to the relative order of objects. These rankings (or permutations) are called the *linear extensions* of the partial order. Figure 1.3(c) shows all linear extensions of the partial order in Figure 1.3(b). Inspecting the space of linear extensions allows ranking the objects in a way consistent with the partial order. For example, $a1$ may be preferred to $a4$ since $a1$ appears at rank 1 in 8 out of 10 linear extensions, even though both $a1$ and $a4$ are skyline objects. A crucial challenge for such approach is that the space of linear extensions grows exponentially in the number of objects [11].

Furthermore, in many scenarios uncertainty is quantified probabilistically. For example, a moving object's location can be described using a probability distribution defined on some region based on location history [15]. Similarly, a missing attribute can be filled in with a probability distribution of multiple possible imputations [74, 75]. Augmenting uncertain scores with such probabilistic quantifications generates a (possibly non-uniform) probability distribution of linear extensions that cannot be captured using a standard partial order or dominance relationship.

1.3 Challenges

There are multiple challenges associated with incorporating data uncertainty in the semantics and processing techniques of top-k queries. We summarize such challenges as follows:

- *Query Semantics.* Proper semantics of top-k queries on uncertain data need to integrate the semantics of querying uncertain data with the conventional semantics of top-k queries. Different possible query semantics arise from this integration. For example, conventional top-k query semantics assume that each tuple has a single score and a distinct rank (by resolving ties using a deterministic tie breaker). However, under uncertainty, query semantics allowing for a range of possible scores per tuple, and hence a set of possible ranks per tuple, need to be adopted. This is a clear departure from the conventional semantics of top-k queries, and is not also captured by current query semantics in uncertain databases.

- *Ranking Models.* Most current ranking models assume that computed tuple scores induce a total order on query results. Such total order model can be insufficient to capture uncertainty in the underlying data, and its impact on the computed ranking of query results. While partial order models can capture uncertainty in the relative order of tuples, incorporating probabilistic ranking quantifications in such models requires extending the definition of a partial order. We thus need to construct probabilistic ranking models different from the currently adopted models.

- *Query Processing.* While minimizing the number of accessed tuples is central to most conventional ranking techniques, uncertainty adds further processing complexity making existing methods inapplicable. Integrating ranking and uncertainty models yields a probability distribution over a huge space of possible rankings that is exponential in the database size. Hence, we need efficient algorithms to process such space in order to compute query answers. Under these settings, integrating tuple retrieval, ranking, and uncertainty management, within the same framework, is essential for efficient processing.

1.4 Contributions and Dissertation Outline

We present a summary of our contributions, and give the organization of the remainder of this dissertation.

1.4.1 Semantics of Top-k Queries on Uncertain Data

This dissertation presents the first proposed semantics of top-k queries on uncertain data. Our key contributions are the following:

- We study the impact of tuple level uncertainty on the semantics and formulations of ranking queries (Section 3.1).
- We introduce a novel probabilistic ranking model, based on partial orders, to represent the space of tuple orderings originating from attribute level uncertainty (Section 3.2).
- We formulate the problem of ranking uncertain data by introducing new semantics of top-k queries that can be adopted in different application scenarios (Section 3.4).

1.4.2 Top-k Selection Algorithms

We give novel query evaluation techniques to compute top-k query answers under both tuple level and attribute level uncertainty models.

For tuple level uncertainty model, our key contributions are the following:

- We introduce a processing framework that allows for early query termination by integrating tuple retrieval, uncertainty management, and ranking in a pipelined fashion (Section 4.1).
- We define top-k query evaluation as a space search problem. We introduce principled space navigation algorithms, with performance guarantees, to lazily and partially materialize the answer space while searching for top ranked query answers (Sections 4.2).

For attribute level uncertainty model, our key contributions are the following:

- We introduce a space pruning algorithm to cut down the answer space, allowing efficient query evaluation to be conducted subsequently (Sections 5.2).
- We give branch-and-bound search algorithms to compute exact query answers based on A* search. The search algorithms explore the space of possible answers, and early-prune search paths that do not lead to query answers (Section 5.5.1).
- We propose novel sampling techniques based on a Markov Chain Monte-Carlo (MCMC) method to compute approximate query answers (Section 5.5.2).
- We study the problem of optimal rank aggregation in partial orders induced by uncertain scores under both the *Spearman footrule* and *Kendall tau* distance metrics (Section 5.6):
  - We give a polynomial time algorithm to solve the problem under Spearman footrule distance (Section 5.6.1).
  - We identify classes of partial orders in which computing the optimal rank aggregation under Kendall tau distance has polynomial time cost. We give the corresponding query evaluation algorithms, and provide a detailed complexity analysis (Section 5.6.2).

1.4.3 Top-k Join Algorithms

We build on our techniques for computing top-k queries under attribute level uncertainty to support top-k join queries on uncertain data. In top-k join (rank join) queries, scores are computed for join results, rather than base tuples, and the top-k join results are reported. Our key contributions are summarized as follows:

- We formulate the problem of rank join under uncertainty, and give new query definitions that can be adopted in various application scenarios (Section 6.1).
- We extend rank join methods to handle uncertain scores, and provide a pipelined query operator implementation of uncertainty-aware rank join algorithm. The implementation can be integrated into relational query plans (Section 6.2).

- We present a new infrastructure for probabilistic rank join based on Monte-Carlo simulation. The infrastructure handles dependencies among the scores of join results using a novel join-aware sampling method, and incrementally reports ranked results under multiple probabilistic ranking semantics (Section 6.3).

1.4.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 gives an overview of related works from the literature of rank-aware query processing and probabilistic data management. Chapter 3 describes our adopted uncertainty models, and gives formal definitions for the problems we study in this dissertation. Chapter 4 presents our proposed processing techniques for top-k queries under tuple level uncertainty. Chapter 5 presents our proposed processing techniques for top-k queries under attribute level uncertainty. Chapter 6 describes our proposal to formulate and compute rank join queries under score uncertainty, where we also present the details of MashRank, a research prototype that applies our techniques in data mashup scenarios on the Web. Finally, Chapter 7 gives our conclusions, discusses the limitations of our proposal, and lists a number of directions for future work.

Chapter 2

Background and Related Work

In this chapter, we review related work from the literature of top-k processing (Section 2.1), and probabilistic data management (Section 2.2). We also give background on the technical tools we make use of in our methods, namely Monte-Carlo method and the theory of Markov chains (Section 2.3). We finally describe a set of recent proposals for supporting ranking queries on uncertain data (Section 2.4).

2.1 Top-k Processing

Formulating and processing top-k queries have been addressed from different perspectives in the current literature. We have conducted an extensive literature survey on this topic in [36].

A key component in any top-k processing technique is the scoring (ranking) function. The properties of the scoring function largely influence the design of top-k processing techniques. One important property is the ability to determine upper-bounds on scores. This property allows early pruning of certain query results without exactly knowing their scores. A *monotone* scoring function facilitates upper-bound computation. A function F, defined on predicates $p_1, \dots, p_n$, is monotone if $F(p_1, \dots, p_n) \leq F(\acute{p_1}, \dots, \acute{p_n})$ whenever $p_i \leq \acute{p_i}$ for every i. Most of the current top-k

processing techniques assume monotone ranking functions since they fit in many practical scenarios, and have appealing properties allowing for efficient top-k processing.

In the following, we describe the ranking query models adopted by current proposals. We discuss two different models: (1) top-k selection, and (2) top-k join.

Top-k Selection. In this model, scores are attached to base tuples, and the query reports the k tuples with the highest scores, similar to the query in Figure 1.1 (b). Scores may not be readily available since they could be the outcome of a user-defined scoring function that aggregates information coming from different tuple attributes. Consider a relation R, where each tuple in R is associated with some score that gives it a rank within R. The score of a tuple $t \in R$ is given by user-defined scoring function $F(t)$ defined on a set of scoring predicates $\{p_1(t), \ldots, p_m(t)\}$ in R. For example, a scoring predicate can simply be a column in R. A top-k selection query produces the top-k ranked tuples in R. A possible SQL-like formulation for expressing a top-k selection query is the following:

```
SELECT *
FROM R
WHERE selection condition
ORDER BY $F(p_1(t), \ldots, p_m(t))$
STOP AFTER  $k$
```

The NRA algorithm [24] is one example of top-k techniques that adopt the top-k selection model. The input to the NRA algorithm is a set of sorted lists, each ranks the "same" set of objects based on one scoring predicate. The output is a ranked list of these objects ordered on the aggregate input scores. The NRA algorithm finds the top-k answers by adopting sequential access retrieval from each list. The NRA algorithm may not report the exact object scores, as it produces the top-k answers using bounds computed on their exact scores. The score lower-bound of some object t is obtained by applying the score aggregation function on t's known scores and the minimum possible values of t's unknown scores. On the other hand, the score upper-bound of t is obtained by applying the score aggregation function on t's known scores and the maximum possible values of t's unknown scores, which are the same as the last seen scores in the corresponding ranked lists. This allows the algorithm to report a top-k object even if its score is not precisely known. Specifically, if the score lower-bound of an object t is not below the score upper-bounds of all other objects (including unseen objects), then t can be safely reported as the next top-k object.

Top-k Join. In this model, scores are assumed to be attached to join results rather than base tuples. A top-k join query joins a set of relations based on a given join condition, assigns scores to join results based on some scoring function, and reports the top-k join results. Consider a set of relations $\{R_1, \ldots, R_m\}$. Each tuple in R_i is associated with some score that gives it a rank within R_i . The top-k join query joins R_1 to R_m and produces the results ranked on a total score. The total score is computed according to a user-defined function F that combines the individual scores. A possible SQL-like formulation of a top-k join query is the following:

```
SELECT *
FROM $R_1, R_2, \ldots, R_m$
WHERE join condition($R_1, R_2, \ldots, R_m$)
ORDER BY $F(R_1.score, R_2.score, \ldots, R_m.score)$
STOP AFTER $k$
```

Many top-k join techniques address the interaction between computing the join results and producing the top-k answers. One example is the Rank-Join algorithm [35], which efficiently integrates the joining and ranking tasks. Similar to the NRA algorithm, the Rank-Join algorithm adopts no random access, however, the main difference is that the Rank-Join algorithm maintains the scores of the completely seen join combinations only, not partially seen objects in each list as in the NRA algorithm. The Rank-Join algorithm scans input lists (the joined relations) in the order of their scoring predicates. Join results are discovered incrementally as the algorithm moves down the ranked input relations. For each join result j, the algorithm computes a score for j using a score aggregation function F. The algorithm maintains a threshold T bounding the scores of join results that are not discovered yet. The top-k join results are obtained when the minimum score of the k join results with the maximum $F(\cdot)$ values is not below T.

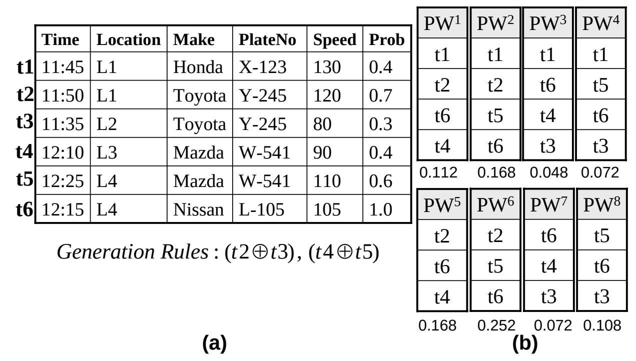
Current proposals in top-k processing provide a good basis for ranking probabilistic data in general. Specifically, these techniques reduce the cost of top-k queries by exploiting two main optimizations: (1) most database tuples are not part of the query answer, and hence many tuple retrieval operations can be avoided; and (2) the scores of retrieved tuples can be bounded, i.e., not fully computed, while still being able to rank query answers. Our problem formulation, described in Section 3.4, is based on integrating tuples' scores and probabilities as two interacting ranking dimensions. Current top-k processing proposals assume deterministic data, and hence they are not explicitly designed to treat probability as an additional ranking dimension. We extend the optimization opportunities of top-k queries in the context of probabilistic databases, and analyze the implication of incorporating probability as an additional ranking dimension.

2.2 Managing Probabilistic Data

Uncertain and incomplete data are common in real life. Managing such data is currently receiving increasing attention in many application domains, e.g., sensor networks, data cleaning, data integration, information extraction, and location-based services. These domains exhibit uncertainty in their underlying data, coupled with increasing demand from users to efficiently derive high-quality answers for the queries posed on such data. We next discuss multiple current proposals for modeling and querying uncertain data.

2.2.1 Data Models

We discuss two important constructs for building uncertain data models. The first construct is *possible worlds semantics*, where data uncertainty is captured by viewing the database as a set of possible instances that correspond to the different possible instantiations of the uncertain data items. The second construct is *data dependencies*, where the dependencies among data items are represented by some dependency model.



	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)

PW^1	PW^2	PW^3	PW^4
t1	t1	t1	t1
t2	t2	t6	t5
t6	t5	t4	t6
t4	t6	t3	t3
0.112	0.168	0.048	0.072

PW^5	PW^6	PW^7	PW^8
t2	t2	t6	t5
t6	t5	t4	t6
t4	t6	t3	t3
0.168	0.252	0.072	0.108

(b)

Figure 2.1: Probabilistic database (a) Probabilistic relation and generation rules (b) Possible worlds space

We then describe several proposed implementations of uncertain data models.

Possible Worlds Semantics

Many proposed uncertainty models, e.g., [38, 4, 56], adopt *possible worlds* semantics, where a probabilistic database $\mathcal{D}$ is viewed as a set of possible instances (worlds) $\{PW^1, \ldots, PW^n\}$. The possible worlds space represents an enumeration of all possible views of the database resulting from the uncertainty or incompleteness in the underlying data.

Possible worlds probabilities are determined based on the probabilistic dependencies among tuples, e.g., mutual exclusion of tuples that map to the same real world entity [56]. We call such dependencies *generation rules*, since they control how the possible worlds space is generated. Such rules could naturally arise with unclean data [5], or could be enforced to satisfy application requirements or reflect domain semantics [73, 56, 9]. Moreover, the relational processing of probabilistic tuples induces dependencies among intermediate query results, even when base tuples are independent [20]. We elaborate on this point in Section 2.2.2.

To illustrate, Figure 2.1(a) shows the `Readings` relation, from Example 1 in Chapter 1, augmented with *generation rules* that enforce the following constraint:

"based on radar locations, the same car cannot be detected at two different locations within 1 hour interval." Figure 2.1(b) shows the possible worlds, and their probabilities. Each world can be seen as a *joint* event of the *existence* of 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 apply to these tuples. The given xor rules in Figure 2.1(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 independent. Consequently, $\Pr(PW^1) = \Pr(t1 \wedge t2 \wedge t6 \wedge t4 \wedge \neg t3 \wedge \neg t5) = 0.4 \times 0.7 \times 1.0 \times 0.4 = 0.112$. The probabilities of other worlds are computed similarly. Any possible world, other than $PW^1 \dots PW^8$, has zero probability based on tuple probabilities and generation rules.

Data Dependencies

Dependency models give a foundation for capturing and dealing with noise and uncertainty of real-world data. A Bayesian network is an example probabilistic model that compactly encodes complex dependencies among random variables. The key insights exploited in Bayesian networks are the locality of dependence and conditional independence, i.e., a variable is directly dependent on only a few others (called parent variables), and conditionally independent of other variables given its parents. A Bayesian network captures this insight by representing the variables joint distribution as a directed acyclic graph whose nodes are the variables and whose edges are the direct dependencies. Each node is associated with a conditional probability table specifying the node's probability distribution given each combination of the values of its parents.

One limitation of Bayesian networks is that they can only encode dependencies among a fixed set of variables. The concept of an entity that encodes the properties of a set of similar instances is missing in such model. This makes the model incapable of reasoning about entities with dependent properties, or represent domains where the set of entities and their relations are not fixed in advance. Probabilistic Relational Models (PRMs) [26, 10] augment probabilistic models with relational constructs to model domains as entities, properties, and relations among them. Hence, the uncertainty of the properties of an entity and its dependence on other properties in the same entity or other entities can be captured.

The basic components in a PRM are *entities* and *relations*. Entities define a set of disjoint classes $X_1, \ldots, X_n$. Each class has a set of attributes $\mathcal{A}(X_i)$. Each attribute $A_j \in \mathcal{A}(X_i)$ has some fixed domain of values. Relations in a PRM encode dependencies among entities. The PRM induces a probability distribution whose random variables are the entity attributes. The PRM defines for each attribute a set of parents, and a local probabilistic model that specifies the dependence on these parents. The PRM defines the dependency model at the class level, allowing it to be used for any instance in the class. Further, the PRM explicitly uses the relational structure of the model, in that it allows the probabilistic model of an entity attribute to depend on the attributes of related entities.

The semantics of the PRM is that given a dependency structure, we have a set of random variables of interest: the set of entity attributes in the structure. The PRM specifies a probability distribution over the possible assignments of values to these random variables, such that each possible value assignment is an instance whose probability is computed by multiplying the conditional probability distributions of the attributes under consideration. We give an example highlighting the details of the PRM when discussing model implementations later in this section.

Model Implementations

One of the classical models that adopt possible worlds semantics to capture uncertainty and incompleteness in attribute values is the c-tables model [38]. c-tables are relational tables whose attributes are represented using variables, and each tuple is associated with a Boolean condition on the attribute variables. A tuple belongs to the database if and only if its associated condition is satisfied. Figure 2.2 gives a simple example of a c-table, where some attributes are represented as variables, and each tuple is associated with a Boolean condition, which can be empty. Assuming the integer domain for all attribute variables, Figure 2.2 shows some of the possible

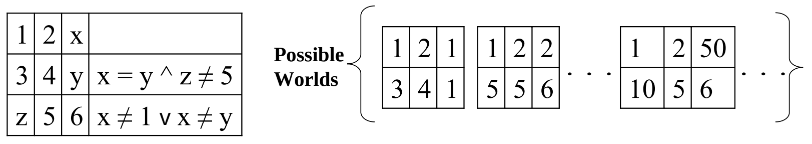


Figure 2.2: c-table and possible worlds space

worlds that can be enumerated based on possible assignment of attribute values. The possible worlds space has an infinite size since the variables' domains are infinite. By restricting the domains of the variables, e.g., variables only take Boolean values, the number of possible worlds becomes finite.

The conditions associated with tuples in the c-tables capture the uncertainty on the tuple level, while attribute variables capture the uncertainty on the attribute level. The c-tables model assumes that these two types of uncertainties are somehow related, i.e., a tuple belongs to the database if the assignment of attribute variables grounds tuple's condition to *true*. Many of the proposed models afterwards, e.g., [4, 56, 20, 14, 27], treat tuples' uncertainty and attributes' uncertainty separately by introducing two basic types of uncertainty quantified with probability values. The first type, usually referred to as "membership uncertainty" [20, 56], treats tuples as probabilistic events capturing the belief that they belong to the database. 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. The probabilities of tuples' events originate from different sources, e.g., reliability of data source in data integration environments [32], or similarity measures in approximate matching [20]. The second uncertainty type, referred to as "value uncertainty" [14, 56, 49] represents attributes as probability distributions on continuous or discrete domains of possible values, e.g., modeling readings of sensing devices, or data entry errors in dirty databases.

The proposal in [8] addressed efficient materialization of the possible worlds space based on the concept of world-set decompositions (WSDs). The WSDs encode the

TID	SSN	Name	Marital Status
t1	{185,785}	Smith	{1,2}
t2	{185,186}	Brown	{1,2,3,4}

(a)

t1.S	t1.N	t1.M	t2.S	t2.N	t2.M
185	Smith	1	186	Brown	1
185	Smith	1	186	Brown	2
185	Smith	1	186	Brown	3
185	Smith	1	186	Brown	4
185	Smith	2	186	Brown	2
..	..	..	..	..	..
785	Smith	2	186	Brown	4

(b)

t1.S	t2.S
185	186
785	185
785	186

x

t1.N
Smith

x

t1.M
1
2

x

t2.N
Brown

x

t2.M
1
2
3
4

(c)

Figure 2.3: (a) Relation with value uncertainty (b) Worlds-set relation (c) World-set decomposition

world-set relation whose tuples are the possible worlds of a relation with value uncertainty. Each tuple in the world-set relation is assumed to be padded with a special symbol $\perp$ such that all tuples (worlds) have the same arity. The world-set relation is impossible to maintain explicitly even for probabilistic relations with reasonable size, due to the explosion of the possible worlds space. This problem is addressed by representing the world-set relation as a set of relations (components) such that the Cartesian product of the components gives the world-set relation. Each component in the WSDs enumerates possible values of one attribute in one tuple in the original relation. WSDs are based on the independence among attribute values. That is, each set of attribute values in different tuples involving dependencies is separated as a single component in the WSDs. For example, for a relation with value uncertainty in the SSN attribute, in order to enforce the constraint that the SSN value must be unique, a component enumerating only the unique combinations of SSN values in different tuples is maintained in the WSDs. The WSDs are stored in a relational schema that maps attribute values in each component to tuples in the original relation.

Figure 2.3 depicts the previous example for a relation that holds survey data with value uncertainty. The world-set relation has a total of 32 records corresponding to all possible combinations of attribute values. Enforcing the constraint that SSN is unique reduces the number of possible worlds to 24, which are represented using WSDs by grouping dependent SSN values in one component.

The above concepts were the bases of several research projects that address modeling uncertain data. The TRIO system [73, 56, 9] introduced *working models* to capture uncertainty at different levels by relating uncertainty with lineage and leveraging existing DBMSs capabilities for uncertain data management. Lineage is a mechanism to track query results to their source tuples, allowing for encoding tuple dependencies and computing the probabilities of query answers. The ORION project [14] deals with constantly evolving data in the form of continuous intervals, and presents query processing and indexing techniques for managing uncertain data in such representation. The CONQUER project [5, 27] introduced query rewriting algorithms to extract clean and consistent answers from unclean databases under possible worlds semantics, and proposed methods to derive probabilistic quantifications of data uncertainty. The MystiQ project [20, 54] analyzes the complexity of structurally rich queries (e.g., joins, subqueries, aggregate and group-by) in uncertain databases. It was shown that, in general, exact evaluation of many query types in uncertain databases is intractable. However, there exist some query plans under which the evaluation of some query instances can be done in polynomial time. The MayBMS project [7, 8] proposes a query language and algebra for processing uncertain data, coupled with a space-efficient materialization of the possible worlds space supporting efficient query evaluation.

Graphical models are the primary representations of probabilistic dependencies. Graphical models compactly encode the joint distribution of a set of variables using a graph. The graph nodes are the variables, while the graph edges encode direct variables dependencies. Disconnected variables are conditionally independent given a combination of some other variables. These conditional independencies represent

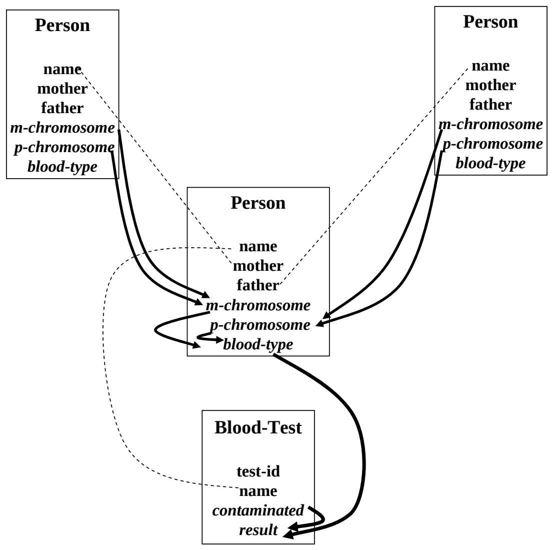


Figure 2.4: Probabilistic Relational Model

the full joint distribution as a product of conditional probability distributions each involving a smaller subset of variables.

The PRM graphical model in [10, 26] focuses on capturing the dependencies among attributes in the underlying relations, which can be used for several tasks such as filling in missing attribute values, computing high-level data summaries, and detecting different anomalies in the underlying data. The relational schema consists of a set of tables where each table may contain descriptive (uncertain) attributes whose values are drawn from a finite domain of possible values. The probabilistic schema models the conditional dependence of descriptive attributes on other attribute values.

We illustrate the previous model using an example from [26] depicted in Figure 2.4. The example describes the inheritance of a gene that determines person's blood type. Each person has two copies of the chromosome containing this gene, one inherited from mother, and one inherited from father. There is also a possibly contaminated test that attempts to recognize the person's blood type. The relational schema contains two relations Person and Blood-Test. Conventional attributes are shown in regular font and probabilistic attributes are shown in italic. Dotted lines indicate foreign keys, while solid lines indicate attribute dependencies. Consider the attribute $BloodTest.Result$. Since the result of a blood test depends on whether it was contaminated, it has $BloodTest.Contaminated$ as a parent. The result also depends on the genetic material of the person tested, which is encoded in the $Person.BloodType$ attribute, which in turn depends on $Person.MChromosome$ and $Person.PChromosome$. Given a set of parents for each attribute, we can define a conditional probability distribution for the attribute values given the possible parents' values. The conditional distributions are learned from training data in the form of different complete instances of the database. Specifically, the counts of value combinations for an attribute and its parents can be used to estimate the conditional probability distributions.

Factorization is a widely-adopted technique to decompose a complex joint distribution into a product of set of independent factors. A graphical dependency model based on factor graphs was proposed in [58]. The proposed model assumes Boolean variables associated with database tuples to capture their uncertain existence in the database. The modeled dependencies are defined over tuples rather than attributes. These dependencies are represented as factors defined on the tuple random variables, where each factor enumerates the possible assignments of tuples' variables dependent on a certain tuple t, as well as the conditional probability of t under each assignment. The factor tables capture different kinds of tuple dependencies such as mutual exclusiveness, implication, and mutual co-existence. Each complete assignment of the tuple random variables gives one possible world (instance) of the database. The probability of an instance is computed by multiplying all factors defined on the tuple random variables in the database.

Factor graphs have been also adopted by the uncertainty model given in [72] to capture the dependencies among random variables corresponding to database objects such as tuples and attributes. In this model, the underlying relational database always represents a single world, and an external factor graph encodes a distribution over possible worlds. The given techniques combine the use of factor graphs with the MCMC method (discussed in Section 2.3) in order to provide scalable query

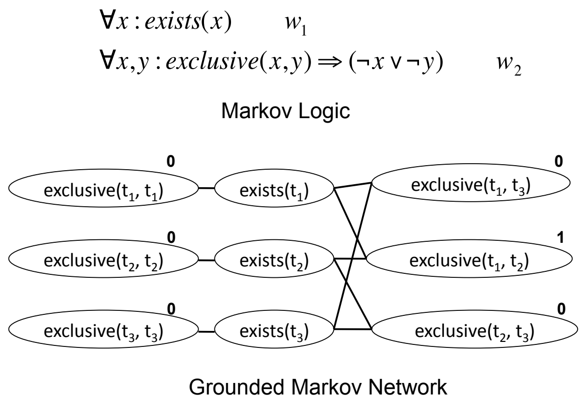


Figure 2.5: Markov Logic Network

evaluation. The MCMC method generates samples from the possible worlds space. The main idea is to evaluate the given query on deltas between consecutive samples generated by the MCMC method, rather than evaluate the query from scratch on each sample. The probability of a tuple t to be in the query answer is then approximated as the relative frequency of samples containing t.

Markov Logic Network (MLN) [22, 55] is a related graphical dependency model that is based on integrating first order logic and Markov networks. Uncertainty in MLN is modeled as a set of first order logic formulas, each associated with a weight. Each formula expresses a constraint on the underlying data, while formula's weight expresses the strength of that constraint. Assigning a weight of ∞ to a formula F means that F is a hard constraint that should always hold, while assigning a finite weight to F means that F is a soft constraint that may be violated. We illustrate the previous model using Figure 2.5, which shows a simple example of Markov logic defined using two weighted first order formulas. The first formula models the uncertainty of the existence of data entities (for example, this can capture tuple uncertainty in an uncertain database), while the second formula models the potential exclusiveness of entities' existence (for example, this can capture generation rules discussed in Section 2.2.1).

Let 'grounding' of a predicate p mean the assignment of variables in p to values from the corresponding domains. The grounding of a formula F means the grounding of all predicates involved in F. A Markov logic L can be represented graphically using a grounded MLN created by assigning each possible grounding of a predicate in L to a binary node, such that the value of that node is 1 if the ground predicate is true, and 0 otherwise. An edge is created between two nodes if the corresponding ground predicates appear together in at least one grounding of one formula in L. A possible world is given by assigning truth values to each grounded predicate in the graph.

We illustrate the previous graphical representation using Figure 2.5, where we assume the variables x and y range over a domain of three possible constants $\{t_1, t_2, t_3\}$. We further assume that only t_1 and t_2 are exclusive. Hence, we assign constant truth values to all the 'exclusive' nodes in the MLN, while the remaining 'exists' nodes are binary variables. Hence, the set of possible worlds contains 8 worlds created by taking all possible truth assignments of the three 'exists' variables. Given a Markov logic, where the weight of formula F_i is given by w_i, the probability of a possible world X is computed using the following log-linear model:

$$\Pr(X) = \frac{1}{Z} \cdot e^{\sum_{F_i} w_i \cdot n_i(X)}$$

where $n_i(X)$ is the number of true groundings of F_i in X, and Z is a normalizing constant computed as follows:

$$Z = \sum_{X \in \mathcal{X}} \prod_{F_i} e^{(w_i \cdot n_i(X))}$$

where $\mathcal{X}$ is the set of all possible worlds.

For example in Figure 2.5, let the world $X = \{exists(t_1), exists(t_2), exists(t_3)\}$. Then, $\Pr(X) = e^{3w_1}/(e^{3w_1} + 3e^{(2w_1+w_2)} + 3e^{(w_1+w_2)} + e^{w_2})$, where the denominator is the normalizing constant Z. Hence, if we set $w_2 = \infty$, we get $\Pr(X) = 0$, since X violates the hard exclusiveness constraint between t_1 and t_2 in X. If, alternatively, we set w_2 to a finite value, the value of $\Pr(X)$ will be inversely proportional to w_2, since X violates a soft constraint in this case.

2.2.2 Query Processing

In possible worlds semantics, each 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.

In the following, we mean by a "probabilistic relation" a relation with uncertain tuple membership. Let Q be a query to be executed over a probabilistic database $\mathcal{D}$, 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:

$$\Pr(t_q.e) = \sum_{PW^i : t_q \in Q(PW^i)} \Pr(PW^i) \tag{2.1}$$

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:

$$(\sigma^p_c(r)).e = \begin{cases} r.e & if\ c(r) = true \\ undefined & if\ c(r) = false \end{cases} \ ;\ where\ c(r)\ is\ the\ selection\ condition \tag{2.2}$$

$$(\pi^p_{\{A_1 \ldots A_n\}}(r)).e = \bigvee_{\acute{r} \in R : \pi_{\{A_1 \ldots A_n\}}(\acute{r}) = \pi_{\{A_1 \ldots A_n\}}(r)} \acute{r}.e \tag{2.3}$$

$$(r \bowtie^p_{c(r,s)} s).e = \begin{cases} r.e \wedge s.e & if\ c(r,s) = true \\ undefined & if\ c(r,s) = false \end{cases} \quad ;\ where\ c(r,s)\ is\ the\ join\ condition \tag{2.4}$$

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 Equation 2.1 [20]. Such computation is done using the formulation of $t_q.e$ (as given above), and the dependencies that bind the tuple events involved 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 [20]. 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 [70].

Relational query processing on graphical dependency models is based on the concepts of probabilistic inference. For example, the model in [58] uses a factored representation of tuples dependencies to compute the probabilities of output tuples. The computational cost is reduced using variable elimination and factor decomposability, however, the problem remains generally intractable. The query evaluation procedure in [58] encodes dependencies among (intermediate) query output tuples by introducing new factors. The selection operator applied to a tuple t, satisfying the selection predicate, gives a new tuple r dependent on t through a new factor, where the existence event of r is true if and only if the existence event of t is true. Similarly, the join operator gives a new tuple r dependent on the joined tuples $t1$ and $t2$ through a new factor, where the existence event of r is true if and only if both the existence events of $t1$ and $t2$ are true. The projection operator is defined similarly as disjunction of the existence events of the involved tuples.

Probabilistic graphical models can be also used to support queries on tuple frequencies [29]. This frequency information can be used to efficiently answer questions about the expected number of tuples in query answer, which can be used by the query optimizer to choose the appropriate query plan, or it can be used to approximate aggregate query answers. In the following example, we illustrate the approach of [29] to use tuple frequencies for selectivity estimation. Consider two tables R and S such that R has a foreign key, $R.F$, that points to $S.K$, which is the key of S. The joint probability of R and S is defined using a sampling process that randomly samples a tuple r from R and independently samples a tuple s from S. The two tuples may or may not join with each other. A join indicator variable is used to model this event. This variable, J, is binary valued; it is true when $r.F = s.K$ and false otherwise. This sampling process induces a distribution $\Pr(J, \mathrm{A}, \mathrm{B})$, where $\mathrm{A} = \{A_1, \ldots, A_n\}$ is the set of descriptive, i.e., uncertain, attributes in R, while $\mathrm{B} = \{B_1, \ldots, B_m\}$ is the set of descriptive attributes in S. For any query Q over R and S with the predicates $R.\mathrm{A} = a, S.\mathrm{B} = b, R.F = S.K$, the probability that Q is satisfied is found by computing $\Pr(J = true, \mathrm{A} = a, \mathrm{B} = b)$.

Most of the discussed uncertainty models can be extended to allow computing preference scores based on some user-defined scoring functions. However, the interpretation of tuple scores in this context is challenging due to their interaction with probabilities. Probabilistic query models need thus to be extended with score-based ranking semantics. Further, from query processing perspective, current proposals mainly assume Boolean queries, and so they are insufficient to handle probabilistic top-k queries. For example, current probabilistic query processing techniques are primarily tuple-based, i.e., they compute probabilities for individual tuples in query output, while, as we show in Section 3.4, probabilistic top-k queries can be after computing the probabilities of top-k tuple vectors/sets.

2.3 Monte-Carlo Integration and Markov Chains

In this section, we give background on the method of Monte-Carlo integration and the theory of Markov chains. These technical tools are used in our proposed solutions for formulating a probability space of tuple orderings under attribute-uncertainty model, and designing sampling-based ranking techniques.

Monte-Carlo Integration. The method of Monte-Carlo integration [52] computes accurate estimate of the integral $\int_{\acute{\Gamma}} f(x)dx$, where $\acute{\Gamma}$ is an arbitrary volume, by sampling from another volume $\Gamma \supseteq \acute{\Gamma}$ in which uniform sampling and volume computation are easy. The volume $\acute{\Gamma}$ is estimated as the proportion of samples from Γ that are inside $\acute{\Gamma}$ multiplied by the volume of Γ. The average $f(x)$ over such samples is used to compute the integral. Specifically, let v be the volume of Γ, s be the total number of samples, and $x_1 \ldots x_m$ be the samples that are inside $\acute{\Gamma}$. Then, the value of the integral can be estimated as follows:

$$\int_{\acute{\Gamma}} f(x)dx \approx \frac{m}{s} \cdot v \cdot \frac{1}{m} \sum_{i=1}^{m} f(x_i) \tag{2.5}$$

In general, let Γ be a sample space in which uniform independent sampling can be done. Assume that we would like to estimate ρ, the volume of some subspace embedded in Γ relative to the volume of Γ. Given two real numbers $\epsilon \in (0, 1]$ and $\delta \in (0, 1]$, the Monte-Carlo method computes an estimate of ρ, denoted $\hat{\rho}$, such that $\Pr\ (\ |\rho - \hat{\rho}| \ \leq\ \epsilon\ \cdot\ \rho)\ \geq\ (1 - \delta)$, provided that the number of drawn samples from Γ is in $\Omega(\frac{1}{\rho \cdot \epsilon^2} ln(\frac{1}{\delta}))$ ([50], Theorem 11.1).

Several variations of Monte-Carlo methods have been used to solve different database problems. The closest work to our study is the proposal given in [54], where the objective is to find the top-k probable records in the answer of conjunctive queries that do not have score-based ranking aspect, which is the main focus of this dissertation. Hence, the data model, problem definition, and processing techniques are quite different in both works. For example, the proposed Monte-Carlo method in [54] is mainly used to estimate the satisfiability ratios of DNF formulae corresponding to the membership probabilities of individual records, while we study computing the probabilities of possible tuple orderings.

Markov Chains. We give a brief description for the theory of Markov chains. We refer the reader to [40, 43] for more detailed coverage of the subject. Let X be a random variable, where X_t denotes the value of X at time t. Let $S = \{s_1, \ldots, s_n\}$

be the set of possible X values, denoted the *state space* of X. We say that X follows a Markov process if the probability that X moves from the current state to a next state depends only on its current state; X's previous history contains no further information. That is, $\forall s_i, s_j : \Pr(X_{t+1} = s_i | X_0 = s_m, \ldots, X_t = s_j) = \Pr(X_{t+1} = s_i | X_t = s_j)$. A Markov chain is a state sequence generated by a Markov process. The transition probability between a pair of states s_i and s_j, denoted $\Pr(s_i \rightarrow s_j)$, is the probability that the process at state s_i moves to state s_j in one step. A Markov process of n states is usually described using a stochastic $n \times n$ matrix P, where $P[i,j] = \Pr(s_i \rightarrow s_j)$ and $\sum_j P[i,j] = 1$.

A Markov chain may reach a stationary distribution π over its state space, where the probability of being at a particular state is independent from the initial state of the chain. A stationary distribution π satisfies $\pi = P\pi$. Any distribution satisfying the detailed balance equation (Equ. 2.6) is a stationary distribution of the Markov chain [43].

$$\Pr(s_i \rightarrow s_j)\pi(s_i) = \Pr(s_j \rightarrow s_i)\pi(s_j) \tag{2.6}$$

The conditions of reaching a unique stationary distribution are *irreducibility* (i.e., any state is reachable from any other state), and *aperiodicity* (i.e., the chain does not cycle between states in a deterministic number of steps).

Markov Chain Monte-Carlo (MCMC) Method. The concepts of Monte-Carlo method and Markov chains are combined in the MCMC method [40] to simulate a complex distribution using a Markovian sampling process, where each sample depends only on the previous sample.

A standard MCMC algorithm is the Metropolis-Hastings (M-H) sampling algorithm [31]. Suppose that we are interested in drawing samples from a target distribution $\pi(x)$. The (M-H) algorithm generates a sequence of random draws of sample values that follow $\pi(x)$ as follows:

1. Start from an initial sample x_0.

2. Generate a candidate sample x_1 from an *arbitrary* proposal distribution $q(x_1|x_0)$.

3. Accept the new sample x_1 with probability
   $\alpha = min(\frac{\pi(x_1).q(x_0|x_1)}{\pi(x_0).q(x_1|x_0)}, 1)$.

4. If x_1 is accepted, then set $x_0 = x_1$.

5. Repeat from step (2).

The (M-H) algorithm draws sample values biased by their probabilities under π. At each step, a candidate sample x_1 is generated based on the current sample x_0. For example, if samples are simply real values, x_1 can be computed as $x_0 + \epsilon$, where ϵ is a random positive/negative real value. The ratio α compares $\pi(x_1)$ and $\pi(x_0)$ to decide on accepting x_1. The main idea is that we always accept a new sample that has a higher probability than the probability of the current sample according to the target distribution. On the other hand, when the new sample has a smaller probability than the probability of the current sample, the new sample is accepted with some probability. As a result, values that are highly probable according to the target distribution are more likely to appear in the generated sequence of samples.

The (M-H) algorithm satisfies the balance condition (Equation 2.6) with arbitrary proposal distributions [31]. Hence, the algorithm converges to the target distribution π. The (M-H) algorithm is typically used to compute distribution summaries (e.g., average) or estimate a function of interest on π.

2.4 Ranking Queries on Uncertain Data

The techniques we present in this dissertation are the first to address formulating and computing score-based ranking queries in the presence of data uncertainty. Recently, multiple other works that build on our proposed query semantics have been proposed. In addition, other formulations of ranking queries on uncertain data have also been identified. We discuss some of the recent proposals in this area.

Supporting ranking queries on uncertain data has been first proposed in our work in [64, 65], while [76, 33, 17] proposed other query semantics and efficient processing algorithms. The proposal in [48] uses the notion of *generating functions* to construct a unified ranking function that can be instantiated to multiple ranking functions proposed in the current literature. The given algorithms use an and/xor tree model (discussed in more detail in Section 3.1), where leaf nodes are tuple instances that can be possibly exclusive. The uncertainty model in all these works assumes that tuples have deterministic single-valued scores, and they are associated with membership probabilities.

In [17], a number of plausible properties (borrowed from conventional ranking queries) of adopted semantics for ranking queries on uncertain data are given:

- Exact-k: A top-k answer contains exactly k tuples.
- Containment: The top-$(k + 1)$ answer contains the top-k answer.
- Unique ranking: Each tuple appears in a unique position in the top-k answer.
- Value invariance: Changing the scores of tuples without changing their relative order should not alter the top-k answer.
- Stability: If tuple t is part of the top-k answer, then t should remain in the top-k answer if its score is increased.

It was shown in [17] that proposed semantics for ranking queries do not satisfy all of these properties together. The main reason is that the existence of dependencies among tuples may introduce irregularities in the space of possible tuple orderings. The proposed semantics in [17] is based on computing the *expected rank* of each tuple in the space of possible worlds. While satisfying the previous properties, *expected ranks* can be easily thrown off with the existence of outliers. For example, we may not be able to report tuples with very high scores and probabilities, if the majority of other tuples have low scores and/or probabilities. In addition, a ranking based on expected ranks may not be consistent with the modeled dependencies. For example, the top-2 tuples based on expected ranks may be exclusive tuples that do not appear together in any possible world. This is not the case with other query semantics that do not rely on computing expected values.

In [33], computing probabilistic ranking queries with a given probability threshold is addressed. Given a threshold τ, the objective is to report each tuple whose probability to appear in the top-k answer is at least τ. The given techniques are based on dynamic programming formulation under tuple level uncertainty.

The uncertainty model we propose in [62] (discussed in Section 3.2) is the first to study the impact of attribute level uncertainty on score-based ranking. The model is based on assigning a score range to each tuple; a representation that fits many real data sources particularly on the Web (cf. Section 1). In contrast to tuple uncertainty models (e.g., [48, 33, 76]), our score uncertainty model enforces all tuples to belong to every possible world, where the difference among worlds is the relative order of tuples, which captures a new types of uncertainty in the context of score-based ranking.

Our proposal in [67] (discussed in Chapter 6) adopts the attribute level uncertainty model for formulating uncertain rank join queries, where we study the integration of rank join processing with probabilistic ranking. The methods proposed in [47] assume a similar model, where generating functions are used to formulate and compute ranking queries efficiently on continuous score distributions. We mainly address in [67] the consequences of assuming such model in the case of joins.

Some recent works have addressed the problem of computing a consensus ranking from a space of possible worlds. The algorithms given in [46] for computing a consensus ranking return a consensus top-k answer, while the methods we propose in [63] (discussed in Section 5.6) return a consensus full ranking. In addition, while [46] gives an approximate algorithm for computing a consensus ranking under the Kendall tau distance, we identify in [63] different classes of uncertainty models, where there exist exact polynomial time algorithms for computing a consensus ranking under the Kendall tau distance.

Impact of Data Uncertainty on Ranking Queries. The impact of tuple-level and attribute-level uncertainty on ranking queries has been modeled and addressed by current proposals from different perspectives. In most proposals, the two uncertainty types are handled in isolation by assuming that the underlying uncertainty type is either tuple-level (e.g., [64, 33]) or attribute-level (e.g., [62]). An important distinction among proposals that handle attribute-level uncertainty is their ability to support discrete and/or continuous domains of the uncertain attributes. For discrete uncertain attributes, a mapping can be constructed to model attribute-level uncertainty as tuple-level uncertainty, and hence leverage the ranking techniques developed for tuple-level uncertainty. We discuss this mapping in Section 3.1. Such mapping is not possible (without loss in accuracy) when uncertain attributes have continuous domains. Consequently, specialized processing techniques that handle uncertainty in attributes with continuous domains have also been proposed [62, 63, 47].

The expressiveness of the underlying dependency model also impacts query evaluation. For example, simple models that assume restricted types of dependencies (e.g., exclusiveness rules) usually allow for efficient evaluation techniques with polynomial time complexity (e.g., [17, 33]). In Section 4.3.1, we discuss using dynamic programming techniques for efficient evaluation of ranking queries under a simplistic dependency model. On the other hand, more expressive models (e.g., a graphical model that captures a complicated joint distribution) usually impose non-trivial computational overhead (e.g., [58, 64]). The main reason is that computing the probability of a query answer under such general models usually entails probabilistic inference; a problem that is known to be generally intractable. We discuss using different dependency models in the context of evaluating ranking queries in Chapter 3.

Chapter 3

Query Semantics

The material of this chapter presents our proposed query semantics in [62, 63, 64, 65]. We start by describing the impact of tuple level (Section 3.1) and attribute level (Section 3.2) uncertainty models on the semantics of ranking queries. We then give the formal definition of our proposed query semantics (Section 3.4).

3.1 Ranking under Tuple Uncertainty

Under tuple level uncertainty, tuples are represented as probabilistic events, and the database is modeled as a joint distribution defined on these events. Tuple events can be dependent as we show in Section 2.2. Each tuple is assigned a determinstic score given by a user-defined scoring function.

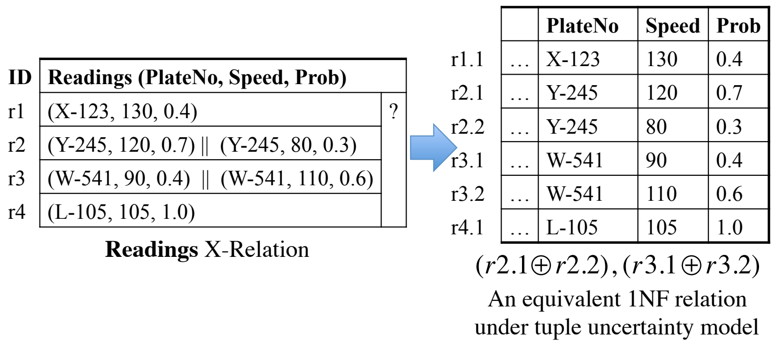
An intuitive and simple choice of ranking query semantics under the tuple uncertainty model is to rank the tuples based on their expected scores. We show in the next example that such a simple approach can result in unreliable ranking.

Example 3 *Expected Scores under Tuple Uncertainty. Assume 2 tuples t_1 and t_2, where $score(t_1) = 1000$, $score(t_2) = 1$, $\Pr(t_1.e) = 0.01$, and $\Pr(t_2.e) = 1$. Then, the expected scores of t_1 and t_2 are 10 and 1, respectively. The ranking based on expected* *scores is thus $\langle t_1, t_2 \rangle$. That is , t_1 is ranked first even though* $\Pr(t_1.e)$ *is very low. Moreover, if $score(t_1)$ drops to 10, the ranking becomes $\langle t_2, t_1 \rangle$. That is, the ranks of t_1 and t_2 are reversed even though $score(t_1) > score(t_2)$ in both cases.*

The previous example shows that we may have two database instances with identical tuple probabilities and identical ranking based on tuple scores. Yet, the ranking based on expected scores is different in the two instances. Another problem with expected scores is that dependencies among tuple events are not considered at all, which means that we can obtain a tuple ranking that is against the modeled dependencies. A third, and more fundamental, problem is that when computing expected scores, we assume that the score of a tuple t equals $score(t)$ with probability $\Pr(t.e)$, and equals 0 with probability $\Pr(\neg t.e)$. This assumption is unjustified, since the non-existence event of t should not be interpreted the same as having 0 score of t.

In order to study the impact of tuple uncertainty on the semantics and evaluation of ranking queries, we assume a general tuple uncertainty model that allows for computing the joint probability of tuple events. Computing this probability is the only interface between the uncertainty model, and our proposed processing framework, as we discuss in Section 4.1. 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 proposed in the literature, conforming to our requirements with different specifications and implementations.

Independence. The simplest possible model is when all tuple events are independent. The model needs only to maintain the membership probabilities of all tuples. Using such model, the joint probability of any combination of tuple events is computed by multiplying the probabilities of the corresponding tuple events. Note that this simple model cannot be adopted when relational operations (e.g., joins) take place, since these operations induce dependencies among intermediate query results.



ID	Readings (PlateNo, Speed, Prob)	
r1	(X-123, 130, 0.4)	?
r2	(Y-245, 120, 0.7) || (Y-245, 80, 0.3)	
r3	(W-541, 90, 0.4) || (W-541, 110, 0.6)	
r4	(L-105, 105, 1.0)	

Readings X-Relation

		PlateNo	Speed	Prob
r1.1	…	X-123	130	0.4
r2.1	…	Y-245	120	0.7
r2.2	…	Y-245	80	0.3
r3.1	…	W-541	90	0.4
r3.2	…	W-541	110	0.6
r4.1	…	L-105	105	1.0

$(r2.1 \oplus r2.2), (r3.1 \oplus r3.2)$

An equivalent 1NF relation under tuple uncertainty model

Figure 3.1: X-Relation Model

X-Relation. The model adopted in [9, 56] captures tuple uncertainty when each entity in the database is represented by one or more tuples called alternatives. Each set of alternatives corresponding to the same entity is denoted as X-Tuple. When the presence of an entity is uncertain, the corresponding x-tuple is denoted as 'maybe x-tuple.' A set of x-tuples is referred to as x-relation, defined as follows:

Definition 1 [X-Relation] *An x-relation R is a set of x-tuples that compactly represents a finite set of possible worlds. Each world is created by choosing exactly one alternative from each x-tuple in R that is not a maybe x-tuple, and choosing zero or one alternatives from each maybe x-tuple in R.*

For example, assume that PlateNo is a key in the `Readings` relation in Figure 1.1. Figure 3.1 shows the x-relation version of the `Readings` relation that complies with such key constraint. Tuples with identical keys are alternatives of the same x-tuple. The x-tuple $r1$ is a maybe x-tuple since it belongs to the relation with less than absolute confidence. X-tuples can be mapped to uncertain tuples by transforming the X-relation into its 1NF, as shown in Figure 3.1. Exclusiveness rules enforce that no two alternatives of the same x-tuple co-exist in the same possible world.

Under the previous mapping, the joint probability of a 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 $t2$, we have $\Pr(t1.e \wedge t2.e) = 0$, while $\Pr(t1.e \wedge \neg t2.e) = \Pr(t1.e)$.

Mutual Exclusion and Co-existence. The and/xor tree model proposed in [46] captures mutual exclusion/co-existence dependencies in a tree structure allowing for combining tuple uncertainty and discrete attribute uncertainty under the same model.

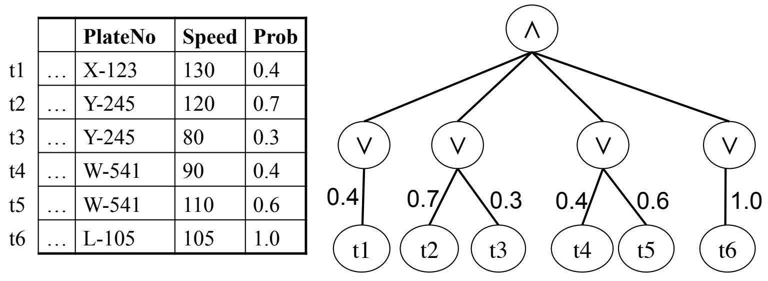
Definition 2 **[And/Xor Tree]** *A tree where each leaf node is a tuple instance, and each internal node is labeled with either* $(\vee)$ *or* $(\wedge)$*. The edge connecting a* $(\vee)$ *node* u *to one of its children* v *is labeled with a non-negative value* $P_{u,v}$ *such that* $\sum_{v \in children(u)} P_{u,v} \leq 1$*. Moreover, the least common ancestor of any two leaf nodes, corresponding to the same source tuple, must be a* $(\vee)$ *node.* □

Given a node u, the and/xor tree model inductively defines a possible world W_u (a subset of the leaves of the subtree rooted by u) by the following recursive process:

- If u is a leaf, $W_u = \{u\}$.
- If u is a $(\vee)$ node, $W_u = \begin{cases} W_v & \textit{with probability } P_{u,v} \\ \phi & \textit{with probability } 1 - \sum_{v \in children(u)} P_{u,v} \end{cases}$
- If u is a $(\wedge)$ node, $W_u = \bigcup_{v \in children(u)} W_v$.

For example, Figure 3.2 shows the and/xor tree corresponding to the relation shown in Figure 2.1.

Under the and/xor tree model, the joint probability of a combination of tuple events is computed based on the labels of the and/xor tree nodes. For example, for two tuples $t1$ and $t2$ whose parent is a $(\vee)$ node, we have $\Pr(t1.e \wedge t2.e) = 0$. On the other hand, for two tuples $t1$ and $t2$ whose parent is a $(\wedge)$ node, we know that $t1.e \equiv t2.e$, and hence $\Pr(t1.e \wedge t2.e) = \Pr(t1.e) = \Pr(t2.e)$.



		PlateNo	Speed	Prob
t1	…	X-123	130	0.4
t2	…	Y-245	120	0.7
t3	…	Y-245	80	0.3
t4	…	W-541	90	0.4
t5	…	W-541	110	0.6
t6	…	L-105	105	1.0

Figure 3.2: And/Xor tree model

General Dependency Models. The previous models are limited in their scope to special cases. Hence, such models may be insufficient to represent and reason about rankings of uncertain data in more general scenarios. A more general model, subsuming the above simple models, is to maintain the explicit *joint probability distribution* of all database tuples. One compact representation of such huge joint distribution is factor graphs (discussed in Section 2.2.1), which are adopted by the proposals in [58, 72]. In this model, tuple dependencies are maintained in the form of conditional probability tables exploiting the concept of conditional independence, and allowing representing arbitrary dependencies.

We illustrate the model of [58] using Figure 3.3 which shows the `Readings` relation, augmented with exclusiveness and implication dependencies. The dependency $(t1 \rightarrow t2)$ means that the existence of $t1$ in any world must be accompanied with the existence of $t2$ in the same world. Figure 3.4 shows the corresponding factor graph that describes these dependencies. In the shown graph, connected tuples are conditionally dependent, while disconnected tuples are independent. Each tuple maintains a conditional probability table representing its conditional probability distribution given its parents. For example, the third row in the table of tuple $t1$ maintains the two conditional probabilities $Pr(t1.e|t2.e \wedge \neg t3.e)$, and $Pr(\neg t1.e|t2.e \wedge \neg t3.e)$. The dependencies and conditional probability tables are inferred from the semantics of the dependencies. However, this model is quite general, since it can compactly encode arbitrary dependencies among tuple events. To illustrate, we show how to compute the joint

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

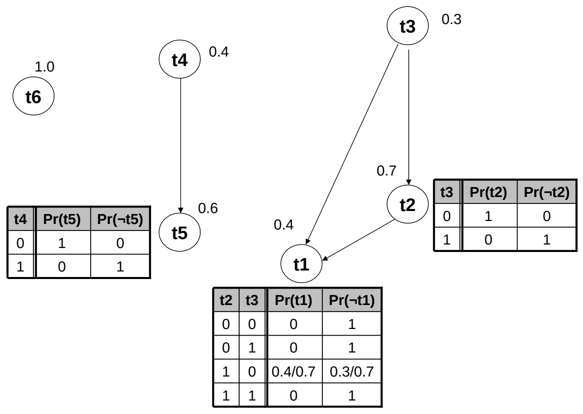
Rules: $(t2 \oplus t3)$, $(t4 \oplus t5)$, $(t1 \rightarrow t2)$

(a)

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

(b)

Figure 3.3: (a) Probabilistic relation (b) Possible Worlds



t4	Pr(t5)	Pr(¬t5)
0	1	0
1	0	1

t3	Pr(t2)	Pr(¬t2)
0	1	0
1	0	1

t2	t3	Pr(t1)	Pr(¬t1)
0	0	0	1
0	1	0	1
1	0	0.4/0.7	0.3/0.7
1	1	0	1

Figure 3.4: Factor graph

probability $Pr(t1.e \wedge t2.e \wedge \neg t3.e)$. Based on Bayes chain rule, such joint probability is expressed as $Pr(\neg t3.e) \times Pr(t2.e|\neg t3.e) \times Pr(t1.e|t2.e \wedge \neg t3.e) = 0.7 \times 1.0 \times \frac{0.4}{0.7} = 0.4$, which is the same as $Pr(t1.e)$ as implied by the semantics of the dependencies.

3.2 Ranking under Attribute Uncertainty

When the values of one or more uncertain attributes are used to compute tuple scores in ranking queries, the resulting scores become uncertain. We are interested in modeling the impact of uncertain scores on the semantics and processing of ranking queries. In the following, for two tuples t_i and t_j, we denote with $(t_i > t_j)$ the preference of t_i over t_j in the computed ranking of query results.

We adopt a general representation of uncertain scores, where the score of tuple t_i is modeled as a probability density function f_i whose domain is a real interval $[lo_i, up_i]$. The density function f_i can be obtained directly from uncertain attributes (e.g., a uniform distribution on possible apartment's rent values as in Figure 1.2). Alternatively, f_i can be computed by density estimation using the predictions of missing/incomplete attribute values that affect tuples' scores [74], or constructed from histories and data correlations as in sensor networks [21]. A deterministic (certain) score is modeled as an interval with equal bounds. For two tuples t_i and t_j with

tID	**Score Interval**	**Score Density**
t_1	[6 , 6]	$f_1 = \delta(x-6)$
t_2	[4 , 8]	$f_2 = 1/4$
t_3	[3 , 5]	$f_3 = 1/2$
t_4	[2 , 3.5]	$f_4 = 2/3$
t_5	[7 , 7]	$f_5 = \delta(x-7)$
t_6	[1 , 1]	$f_6 = \delta(x-1)$

Figure 3.5: Modeling score uncertainty

deterministic equal scores (i.e., $lo_i = up_i = lo_j = up_j$), we assume a tie-breaker $\tau(t_i, t_j)$ that gives a deterministic tuples' relative order. The tie-breaker τ is transitive over tuples with identical deterministic scores (i.e., $[(t_i > t_j) \wedge (t_j > t_k)] \Rightarrow (t_i > t_k)$).

The score intervals shown in Figure 3.5 can simply be the values of an uncertain attribute (e.g., apartment's rent in Figure 1.2), or they can be the result of applying a user-defined scoring function to uncertain attributes (e.g., $0.7 \times \mathit{rent} + 0.3 \times \mathit{deposit}$). For simplicity of presentation, the score densities in Figure 3.5 are assumed to be uniform. Hence, $f_i = 1/(up_i - lo_i)$ (e.g., $f_2 = 1/4$). For tuples with deterministic scores (e.g., t_1), we have an impulse density $f_i = \delta(x - lo_i)$ (effectively, the score density of a tuple t_i with a deterministic score is an impulse function with infinite value at $x = lo_i$ or, equivalently, at $x = up_i$).

Our interval-based score representation induces a *partial order* over database tuples, which extends the following definition of *strict partial orders*:

Definition 3 [Strict Partial Order] *A strict partial order $\mathbb{P}$ is a 2-tuple $(\mathcal{R}, \mathcal{O})$, where $\mathcal{R}$ is a finite set of elements, and $\mathcal{O} \subset \mathcal{R} \times \mathcal{R}$ is a binary relation with the following properties:*
(1) Non-reflexivity: $\forall i \in \mathcal{R} : (i,i) \notin \mathcal{O}$.
(2) Asymmetry: If $(i,j) \in \mathcal{O}$, then $(j,i) \notin \mathcal{O}$.
(3) Transitivity: If $\{(i,j),(j,k)\} \subset \mathcal{O}$, then $(i,k) \in \mathcal{O}$. □

Strict partial orders allow the relative order of some elements to be left undefined.

A widely-used depiction of partial orders is Hasse diagram (e.g., Figure 1.3(b)), which is a directed acyclic graph whose nodes are the elements of $\mathcal{R}$, and edges are the binary relationships in $\mathcal{O}$, except relationships derived by transitivity. An edge (i, j) indicates that i is ranked above j according to $\mathbb{P}$. The *linear extensions* of a partial order are all possible topological sorts of the partial order graph (i.e., the relative order of any two elements in any linear extension does not violate the set of binary relationships $\mathcal{O}$).

We define a strict partial orders that encodes uncertainty in tuple scores based on the following definitions.

Definition 4 [Score Dominance] *A tuple t_i* dominates *another tuple t_j iff $lo_i \geq up_j$.* □

The deterministic tie-breaker τ eliminates cycles when applying Definition 4 to tuples with deterministic equal scores. Based on Definition 4, Property 1 immediately follows:

Property 1 *Score Dominance is a* non-reflexive, asymmetric*, and* transitive *relation.* □

For clarity of presentation, the ranking model we present in this section assumes a single relation under the assumption of independent score densities of individual tuples. This means that there are no constraints or dependencies that determine which combinations of tuple scores co-exist together in a possible world. We address the relaxation of this assumption in Section 6.3, where we show that the score densities of intermediate join results are dependent, which triggers different evaluation techniques.

Under the assumption of independent score densities, the probability that tuple t_i is ranked above tuple t_j, denoted $\Pr(t_i > t_j)$, is given by the following 2-dimensional integral:

$$\Pr(t_i > t_j) = \int_{lo_i}^{up_i} \int_{lo_j}^{x} f_i(x) \cdot f_j(y) dy \; dx \tag{3.1}$$

When neither t_i nor t_j dominates the other tuple, $[lo_i, up_i]$ and $[lo_j, up_j]$ are intersecting intervals, and so $\Pr(t_i > t_j)$ belongs to the open interval $(0, 1)$, and $\Pr(t_j > t_i) = 1 - \Pr(t_i > t_j)$. On the other hand, if t_i dominates t_j, then we have $\Pr(t_i > t_j) = 1$ and $\mathrm{P}(t_j > t_i) = 0$.

We say that a tuple pair (t_i, t_j) belongs to a *probabilistic dominance relation* iff $\Pr(t_i > t_j) \in (0, 1)$.

We next give the formal definition of our ranking model:

Definition 5 [Probabilistic Partial Order (PPO)] *Let $\mathcal{R} = \{t_1, \ldots, t_n\}$ be a set of real intervals, where each interval $t_i = [lo_i, up_i]$ is associated with a density function f_i such that $\int_{lo_i}^{up_i} f_i(x)dx = 1$. The set $\mathcal{R}$ induces a probabilistic partial order* PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$*, where $(\mathcal{R}, \mathcal{O})$ is a strict partial order with $(t_i, t_j) \in \mathcal{O}$ iff t_i dominates t_j. Moreover, $\mathcal{P}$ is the* probabilistic dominance relation *of intervals in $\mathcal{R}$.* □

Definition 5 states that if t_i dominates t_j, then $(t_i, t_j) \in \mathcal{O}$. That is, we can deterministically rank t_i above t_j. On the other hand, if neither t_i nor t_j dominates the other tuple, then $(t_i, t_j) \in \mathcal{P}$. That is, the uncertainty in the relative order of t_i and t_j is quantified by $\Pr(t_i > t_j)$.

Figure 3.6 shows the Hasse diagram and the probabilistic dominance relation of the PPO of tuples in Figure 3.5. We also show the set of linear extensions of the PPO.

The linear extensions of PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$ can be viewed as tree where each root-to-leaf path is one linear extension. The root node is a dummy node since there can be multiple elements in $\mathcal{R}$ that may be ranked first. Each occurrence of an element $t \in \mathcal{R}$ in the tree represents a possible ranking of t, and each level i in the tree contains all elements that occur at rank i in any linear extension. We explain how to construct the linear extensions tree in Section 5.1.

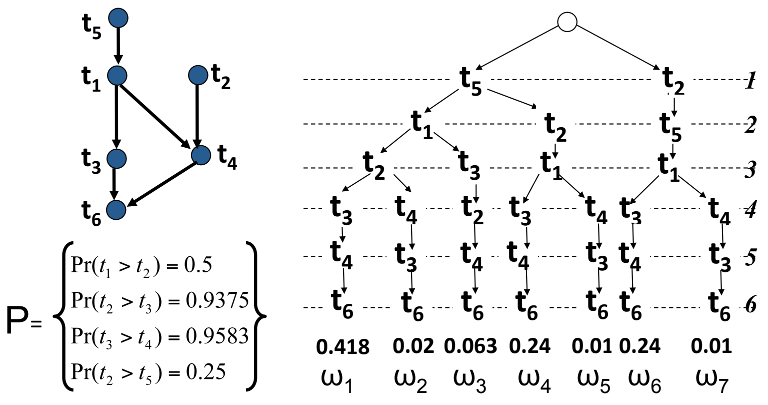


Figure 3.6: Probabilistic partial order and linear extensions

Due to probabilistic dominance, the space of possible linear extensions is viewed as a probability space generated by a probabilistic process that draws, for each tuple t_i, a random score $s_i \in [lo_i, up_i]$ based on the density f_i. Ranking the drawn scores gives a total order on the database tuples (score ties are usually resolved using a deterministic tie-breaker). The probability of such order is the joint probability of the drawn scores. For example, we show in Figure 3.6, the probability value associated with each linear extension. We show next how to compute these probabilities.

Probability Space. The probability of a linear extension is computed as a nested integral over tuples' score densities in the order given by the linear extension. Let $\omega = \langle t_1, t_2, \dots t_n \rangle$ be a linear extension. Then, $\Pr(\omega) = \Pr((t_1 > t_2), (t_2 > t_3), \dots, (t_{n-1} > t_n))$. The individual events $(t_i > t_j)$ in the previous formulation are not independent, since any two consecutive events share a tuple. Hence, For $\omega = \langle t_1, t_2, \dots t_n \rangle$, $\Pr(\omega)$ is given by the following n-dimensional integral with dependent limits:

$$\Pr(\omega) = \int_{lo_1}^{up_1} \int_{lo_2}^{x_1} ... \int_{lo_n}^{x_{n-1}} f_1(x_1) ... f_n(x_n) dx_n ... \, dx_1 \tag{3.2}$$

Since we assume independent tuple scores, Equation 3.2 integrates the product of the score densities of different tuples. If, alternatively, tuples' scores are dependent, a joint score density function needs to replace the product. We discuss this point in Section 6.3.

Monte-Carlo integration (cf. Section 2.3) can be used to compute complex nested integrals such as Equation 3.2. For example, the probabilities of linear extensions $\omega_1, \ldots, \omega_7$ in Figure 3.6 are computed using Monte-Carlo integration.

In the next theorem, we prove that the space of linear extensions of a PPO induces a probability distribution.

Theorem 1 *Let Ω be the set of linear extensions of* PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$. *Then, (1) Ω is equivalent to the set of all possible rankings of $\mathcal{R}$, and (2) Equation 3.2 defines a probability distribution on Ω.* □

Proof. We prove (1) by contradiction. Assume that $\omega \in \Omega$ is an invalid ranking of $\mathcal{R}$. That is, there exist at least two tuples t_i and t_j whose relative order in ω is $t_i > t_j$, while $lo_j \geq up_i$. However, this contradicts the definition of $\mathcal{O}$ in PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$. Similarly, we can prove that any valid ranking of $\mathcal{R}$ corresponds to only one linear extension in Ω.

We prove (2) as follows. First, map each linear extension $\omega = \langle t_1, \ldots, t_n \rangle$ to its corresponding event $e = ((t_1 > t_2) \wedge \cdots \wedge (t_{n-1} > t_n))$. Equation 3.2 computes $\Pr(e)$ or equivalently $\Pr(\omega)$. Second, let ω_1 and ω_2 be two linear extensions in Ω whose events are e_1 and e_2, respectively. By definition, ω_1 and ω_2 must be different in the relative order of at least one pair of tuples. It follows that $\Pr(e_1 \wedge e_2) = 0$ (i.e., any two linear extensions map to mutually exclusive events). Third, since Ω is equivalent to all possible rankings of $\mathcal{R}$ (as proved in (1)), the events corresponding to elements of Ω must completely cover a probability space of 1 (i.e., $\Pr(e_1 \vee e_2 \cdots \vee e_m) = 1$, where $m = |\Omega|$). Since all e_i's are mutually exclusive, it follows that $\Pr(e_1 \vee e_2 \cdots \vee e_m) = \Pr(e_1) + \cdots + \Pr(e_m) = \sum_{\omega \in \Omega} \Pr(\omega) = 1$, and hence Equation 3.2 defines a probability distribution on Ω. □

Based on the formulated probability space, we show in the next example that ranking tuples based on their expected scores can produce unreliable ranking.

Example 4 *Expected Scores under Attribute Uncertainty. Assume 3 tuples, t_1, t_2, and t_3 with score intervals $[0, 100]$, $[40, 60]$, and $[30, 70]$, respectively. Assume that score values are distributed uniformly within each interval. The expected score of each tuple is thus 50, and hence all permutations are equally likely rankings. However, based on Equation 3.2, we compute the probabilities of different rankings of these tuples as follows:* $\Pr(\langle t_1, t_2, t_3\rangle) = 0.25$, $\Pr(\langle t_1, t_3, t_2\rangle) = 0.2$, $\Pr(\langle t_2, t_1, t_3\rangle) = 0.05$, $\Pr(\langle t_2, t_3, t_1\rangle) = 0.2$, $\Pr(\langle t_3, t_1, t_2\rangle) = 0.05$, *and* $\Pr(\langle t_3, t_2, t_1\rangle) = 0.25$. *That is, the rankings have a non-uniform distribution even though the score intervals are uniform with equal expectations.*

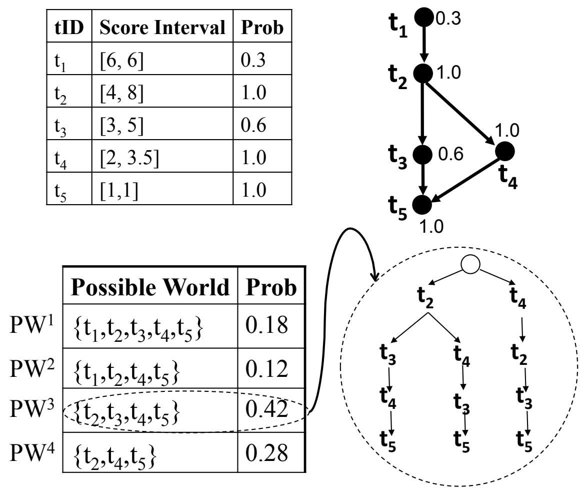
As illustrated in the previous example, the problem with ranking based on expected scores is that for uncertain scores with large ranges, arbitrary rankings that are independent from how the score ranges intersect may be produced. These rankings can be unreliable in some cases. For example, for score intervals that are uniform with equal expectations, the space of possible rankings may have a non-uniform distribution, which is not captured by the equal expected scores.

3.3 Combining Tuple and Attribute Uncertainty

The uncertainty models presented in the previous two sections can be combined to create a more general ranking model capturing both tuple and attribute uncertainty. For example, consider Figure 3.7 which shows a relation of 5 tuples, where each tuple has both probabilistic score and membership probability. We represent the order relationships among tuples using a partial order, where each node in the Hasse diagram is annotated with the membership probability of its corresponding tuple.

Under the previous model, the space of possible rankings can be generated by adopting a two-steps process:

1. Generate different tuple subsets representing the different possible worlds induced by tuple uncertainty.



tID	Score Interval	Prob
t_1	[6, 6]	0.3
t_2	[4, 8]	1.0
t_3	[3, 5]	0.6
t_4	[2, 3.5]	1.0
t_5	[1,1]	1.0

	Possible World	Prob
PW^1	$\{t_1,t_2,t_3,t_4,t_5\}$	0.18
PW^2	$\{t_1,t_2,t_4,t_5\}$	0.12
PW^3	$\{t_2,t_3,t_4,t_5\}$	0.42
PW^4	$\{t_2,t_4,t_5\}$	0.28

Figure 3.7: Space of possible rankings under tuple and attribute uncertainty

2. For each possible world, generate the set of possible tuple permutations induced by score uncertainty.

To illustrate, Figure 3.7 shows the possible worlds induced by tuple uncertainty. We assume independent tuple events, and hence worlds' probabilities are given by multiplication of tuple probabilities. For the possible world PW^3, we generate the set of possible tuple permutations by restricting the partial order to only the tuple set $\{t_2, t_3, t_4, t_5\}$ (i.e., we remove t_1 since it is not included in PW^3). The result is a set of 3 possible permutations (linear extensions), whose probabilities can be computed as we show in Equ 3.2. Note that the probability given by Equ 3.2 in this case is conditioned on the corresponding possible world. In order to obtain the marginal probabilities of possible rankings, we multiply the linear extension probability by the probability of its corresponding possible world.

While our proposed query semantics (discussed in Section 3.4) are generic and apply to any uncertainty model that can be interpreted under possible worlds semantics, the processing techniques we introduce are dependent on the underlying type of uncertainty. In this dissertation, we focus on evaluating ranking queries on uncertainty

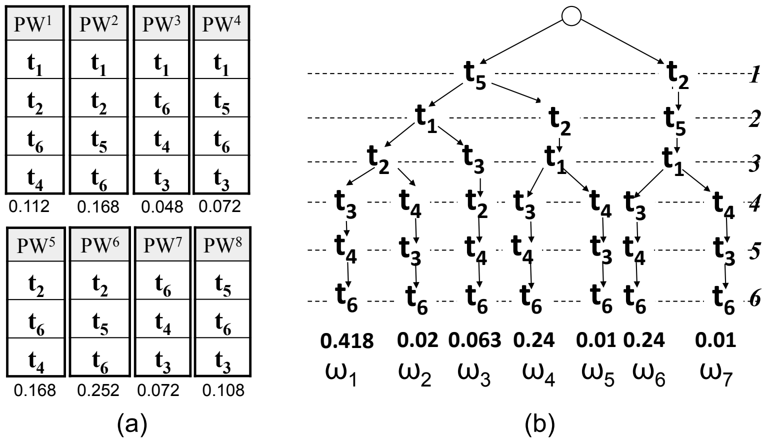


Figure 3.8: Ranked possible worlds under (a) tuple uncertainty (b) attribute uncertainty

models that support either tuple or attribute uncertainty separately. In Section 7.3.1, we discuss potential extensions for handling both uncertainty types conjointly.

3.4 Semantics of Ranking Queries on Uncertain Data

In this section, we present our proposed semantics of top-k queries on uncertain data. We abstract the details of the uncertainty model (which can be either tuple-level or attribute-level uncertainty model) by formulating query semantics on a finite set of possible worlds $W = \{w_1, \dots, w_n\}$, where each world $w_i \in W$ is a valid ranked instance of the database. We elaborate on the ranking requirement of worlds in W using the example given by Figure 3.8.

Under tuple level uncertainty, each world $w_i \in W$ is a subset of database tuples ranked based on the scores given by a query-specified scoring function. For example, Figure 3.8(a) shows the possible worlds of the `Readings` relation in Figure 2.1, ranked on the *Speed* attribute. In this example, $W = \{PW^1, \dots, PW^8\}$.

Under attribute level uncertainty, each world $w_i \in W$ is a permutation of all database tuples corresponding to a possible linear extension of the PPO induced by uncertain scores. For example, Figure 3.8(b) shows the ranked possible worlds (linear extensions) of the PPO in Figure 3.6. In this example, $W = \{\omega_1, \ldots, \omega_7\}$.

We identify two important distinctions between ranked possible worlds generated under each of the tuple level and attribute level uncertainty models:

- Under tuple level uncertainty, for two tuples t_i and t_j, where $score(t_i) > score(t_j)$, we have $(t_i > t_j)$ in all ranked possible worlds containing both t_i and t_j. On the other hand, under attribute level uncertainty, if t_i dominates t_j, then $(t_i > t_j)$ in all ranked possible worlds. Otherwise, $(t_i > t_j)$ in a subset of possible worlds, while $(t_j > t_i)$ in the remaining possible worlds.

- Ranked possible worlds under tuple level uncertainty represent subsets of database tuples, while ranked possible worlds under attribute level uncertainty represent permutations of all database tuples.

As we show in the following, the previous distinctions influence the applicability of some query semantics to each of the two uncertainty models.

We build our proposed query semantics on possible worlds semantics, where we obtain the probability of a query answer by summing the probabilities of possible worlds supporting that answer. Intuitively, we would like to obtain answers that are strongly supported in the space of possible answers. In general, we adopt two formulations to define such answers:

- **F1:** Query answer is the most probable answer in the space of all possible query answers.

- **F2:** Query answer is a consensus answer with the minimum average distance to all possible answers.

In **F1**, we are given a probability distribution of all possible query answers, and the goal is to find the query answer with the highest probability (i.e., the distribution's mode).

In **F2**, we view possible worlds as weighted voters with possibly different opinions regarding the ranking of tuples, and the weight of each voter is the probability of the corresponding possible world. The goal is to aggregate the opinions of such voters in order to find query answer.

An orthogonal construct to the two previous formulations is an answer's granularity. We project the space of possible worlds on different granularity levels, namely, tuple vectors, tuple sets, tuples appearing in a range of ranks, and full tuple orderings. Answering ranking queries at these different granularity levels can be useful to a wide class of applications, as we show at the end of this section.

The input to each of the following query definitions is the set of ranked possible worlds represented as orderings of tuple IDs, while the output is a sequence/set of tuple IDs satisfying a given condition.

We start by defining query semantics under **F1**, where we compute the most probable query answers in the space of all possible answers.

Definition 6 [Uncertain Top Prefix (UTop-Prefix)] *A UTop-Prefix(k) query returns the most probable top-k vector. That is, UTop-Prefix(k) returns* $argmax_p(\sum_{w \in W_{(p,k)}} \Pr(w))$*, where* $W_{(p,k)} \subseteq W$ *is the set of possible worlds having* p *as the top-k vector (i.e., the k-length prefix).* □

UTop-Prefix query returns the tuple vector with the highest probability of being the top-k vector across all worlds. That is, UTop-Prefix query answer is the mode of the distribution of possible top-k answers. For example, in Figure 3.8(a), a UTop-Prefix(2) query returns $\langle t_1, t_2 \rangle$ with probability 0.28, since $\langle t_1, t_2 \rangle$ 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. Similarly, in Figure 3.8(b), a UTop-Prefix(3) query returns $\langle t_5, t_1, t_2 \rangle$ with probability $\Pr(\omega_1) + \Pr(\omega_2) = 0.438$.

Definition 7 **[Uncertain Top Set (UTop-Set)]** *A UTop-Set(k) query returns the most probable top-k set. That is, UTop-Set(k) returns* $argmax_s(\sum_{w \in W_{(s,k)}} \Pr(w))$*, where* $W_{(s,k)} \subseteq W$ *is the set of possible worlds having s as the top-k set.* □

Unlike UTop-Prefix query, UTop-Set query ignores the order within query answer. This allows finding query answers with a relaxed within-answer ranking. However, UTop-Set and UTop-Prefix query answers are related, since the top-k set probability of a set s is the summation of the probabilities of all k-length prefixes involving the same tuples in s. Under tuple level uncertainty, the relative order of any two tuples is fixed across all possible worlds, and hence UTop-Set and UTop-Prefix queries return identical answers. On the other hand, under attribute level uncertainty, UTop-Set and UTop-Prefix query answers are not necessarily identical.

For example, in Figure 3.8(a), a UTop-Set(2) query returns $\{t_1, t_2\}$ with probability 0.28, which is the same as UTop-Prefix(2) query answer. On the other hand, in Figure 3.8(b), the query UTop-Set(3) returns the set $\{t_1, t_2, t_5\}$ with probability $\Pr(\omega_1) + \Pr(\omega_2) + \Pr(\omega_4) + \Pr(\omega_5) + \Pr(\omega_6) + \Pr(\omega_7) = 0.937$. Note that $\{t_1, t_2, t_5\}$ appears as Prefix $\langle t_5, t_1, t_2\rangle$ in ω_1 and ω_2, appears as Prefix $\langle t_5, t_2, t_1\rangle$ in ω_4 and ω_5, and appears as Prefix $\langle t_2, t_5, t_1\rangle$ in ω_6 and ω_7.

Definition 8 **[Uncertain Top Rank (UTop-Rank)]** *A UTop-Rank(*i, j*) query, for* $i \leq j$*, returns the ID of the most probable tuple to appear at any rank* $i \dots j$ *(i.e., from i to j inclusive) in possible worlds in W. That is, UTop-Rank(*i, j*) returns* $argmax_t(\sum_{w \in W_{(t,i,j)}} \Pr(w))$*, where* $W_{(t,i,j)} \subseteq W$ *is the set of possible worlds having t ranked at a position r for* $i \leq r \leq j$*.* □

UTop-Rank query returns the ID of the tuple that appears in a given range of ranks with the highest probability. For example, in Figure 3.8(a), a UTop-Rank$(1, 2)$ query answer is t_2 with probability 0.7, since t_2 appears at rank 1 in PW^5 and PW^6, and appears at rank 2 in PW^1 and PW^2, where $\Pr(PW^1) + \Pr(PW^2) + \Pr(PW^5) + \Pr(PW^6) = 0.7$. Similarly, in Figure 3.8(b), a UTop-Rank$(1, 2)$ query returns t_5 with probability $\Pr(\omega_1)+\cdots+\Pr(\omega_7) = 1.0$, since t_5 appears at all linear extensions at either rank 1 or rank 2. It also follows from possible worlds semantics that UTop-Rank(i,j) probability of a tuple t is the summation of the UTop-Rank(r,r) probabilities of t for $r = i \dots j$.

The three above query definitions can be extended to rank different answers on probability. We define the answer of l-UTop-Prefix(k) query as the l most probable prefixes of length k. We define the answer of l-UTop-Set(k) query as the l most probable top-k sets. We define the answer of l-UTop-Rank(i,j) query as the IDs of the l most probable tuples to appear at a rank $i \dots j$. We assume a tie-breaker that deterministically orders answers with equal probabilities.

We next define query semantics under **F2**, where we compute consensus query answers that minimize the average distance to all possible query answers.

Definition 9 [Uncertain Rank Aggregation Query (URank-Agg)] *A URank-Agg query returns an ordering w^* of all tuple IDs such that $\frac{1}{|W|}\sum_{w \in W} \Pr(w) \cdot d(w^*, w)$ is minimized, where $d(.,.)$ is a measure of the distance between two orderings.* □

URank-Agg query returns an ordering that has the minimum average distance to all ranked possible worlds. The definition of URank-Agg query involves a function d that measures the distance between two orderings. The most common definitions of such functions assume orderings of exactly the same set of elements (we discuss widely-adopted distance functions in Section 5.6). Such assumption applies to possible worlds generated under attribute level uncertainty, since these orderings are permutations of all database tuples. On the other hand, under tuple level uncertainty, the orderings may involve different subsets of database tuples. We thus restrict our evaluation techniques for URank-Agg query to only the attribute level uncertainty model.

We show in Section 5.6 that URank-Agg query can be mapped to a UTop-Rank query under a specific definition of distance measure. We also derive a correspondence between this query definition and the ranking query that orders tuples on their expected scores.

Applications. Our proposed query semantics can be adopted in the following application scenarios:

- A UTop-Prefix query can be used in market analysis to find the most probable product ranking based on evaluations extracted from users' reviews, which may contain uncertain information. Similarly, a UTop-Set query can be used to find a set of products that are most likely to be ranked above all other products.

- A UTop-Rank(i, j) query can be used to find the most probable athlete to end up in a range of ranks in some competition, given a partial order over competitors' strength. A UTop-Rank$(1, k)$ query can be used to find the most likely location to be in the top-k hottest locations based on uncertain sensor readings represented as intervals.

- The semantics of rank aggregation are widely adopted in many applications related to combining votes from different voters to rank a given set of candidates in a way that minimizes the disagreements of voter's opinions. A typical application is building a meta-search engine (a search engine that aggregates the rankings of multiple other engines) as discussed in [23]. An example application of URank-Agg query is preference management in social networking sites, where different users rate objects of interest (e.g., photos, videos, products, etc.) inducing a range of possible scores per object. The ratings can be compactly encoded as a PPO. In these settings, finding a consensus ranking can be important for computing recommendations and planning ad campaigns.

Chapter 4

Ranking with Tuple Level Uncertainty

In this chapter, we describe the techniques we proposed in [64, 65] to support top-k queries in probabilistic databases under tuple level uncertainty model. We start by giving a general query processing framework in Section 4.1. We present the details of our query processing techniques in Section 4.2. We discuss different relaxations of our processing framework assumptions, and their impact on processing techniques, in Section 4.3. We finally present our experimental study in Section 4.4 and summarize this chapter in Section 4.5.

4.1 Processing Framework

Figure 4.1 shows the architecture of our processing framework. The framework is based on two main 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

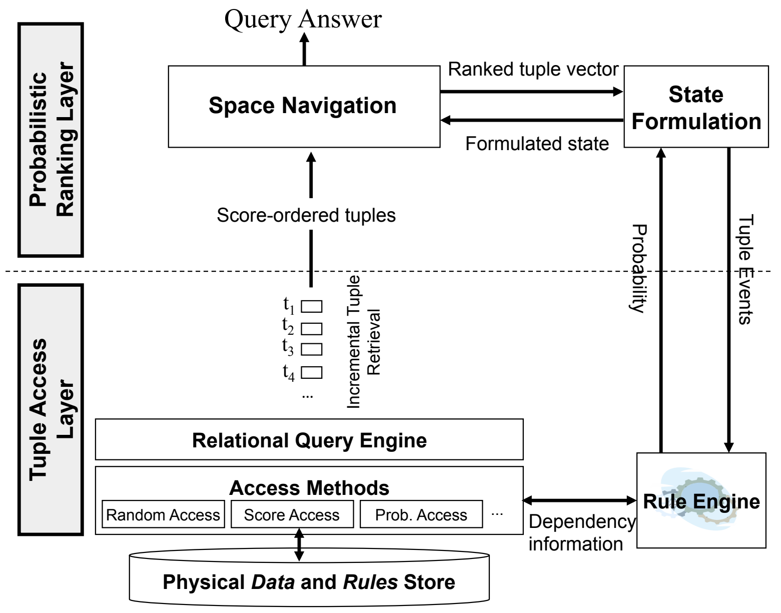


Figure 4.1: Processing framework

query processing capabilities to compute probabilistic top-k queries. Similar arguments are made in the design of the TRIO system [73, 9].

- **DP2:** To leverage current top-k query processing algorithms in deterministic databases. In particular, our framework takes advantage of rank-aware query processing (if supported by the underlying DBMS) to minimize the number of needed-to-access tuples. The framework also adopts the upper-bounding principle, used in several proposals, e.g.,[34, 44], to limit the size of the materialized space.

Tuple Access Layer. Tuple retrieval, indexing and query processing (including filtering and score-based ranking) are the main functionalities of the *Tuple Access Layer*. This layer executes an incoming query, which acts as the tuple source of the upper layer. We show in Section 4.2.1 that sorted score access for output tuples of the *Tuple Access Layer* is necessary for efficient processing.

While our techniques can benefit largely from efficient support to ranking in the *Tuple Access Layer*, our framework is still valid if such support is limited or lacking. However, in that case more tuples would 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 joint probability of such combination. The details of the *Rule Engine* is not the focus of our study, since they vary according to how sophisticated the underlying uncertainty model is, as discussed in Section 3.1. To illustrate, [20] shows 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, [58] uses 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 URank prototype [66], 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 [56, 9], where tuple dependencies are formulated as exclusiveness rules only; and (3) a more complex engine that implements and indexes a Bayesian network that is lazily constructed during query processing to load relevant dependency information on demand, and compute the probabilities of tuple combinations through Bayesian inference techniques.

Probabilistic Ranking Layer. This layer retrieves tuples from the *Tuple Access Layer*, and 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 a formal definition of the problem space of probabilistic top-k queries in Section 4.2.1.

Assumptions. We make several assumptions in our framework design. We relax many of these assumptions in [65], and discuss some of these relaxations in Section 4.3.

- *Tuple Access:* We assume tuples are consumed incrementally, i.e., 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 suffices to get query answer. Moreover, our problem space is exponential in the number of accessed tuples (cf. Section 4.2.2). Hence, minimizing the number of needed-to-see tuples is crucial for efficient processing. 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 RDBMS's. We relax the *Tuple Access* assumption in Section 4.3.2, by considering special cases that allow random access to the underlying database.

- *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 dependent on 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 *Tuple Access Layer*. Dependency information is built incrementally as tuples are produced and pipelined to the

Probabilistic Ranking Layer. Hence, no dependency information, relating tuples currently consumed by the *Probabilistic Ranking Layer* 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 all query output tuples. We address this case in Section 4.3.1, where we relax our assumption by exploiting more available information on tuple dependencies.

- *Event Probability Computation*: We assume the *Rule Engine* responds with *exact* probabilities to the submitted questions (joint probabilities of tuple events' combinations). However, since exact probability computation can be expensive for some query types/plans, particularly projections under set semantics [20], we discuss in [65] relaxing this assumption by dealing with approximate output from the *Rule Engine* in the form intervals enclosing the exact probability value.

4.2 Query Evaluation

In this section, we present our techniques for computing top-k queries over uncertain tuples with deterministic score values. Our query evaluation techniques compute a tuple ranking under each of the query semantics given in Section 3.4. As we discuss in Section 3.4, we do not discuss evaluating URank-Agg queries under tuple uncertainty since possible worlds may not be full permutations of all tuples.

Our techniques are generally based on processing tuples in the order of their scores. Relational processing (e.g., filtering and joining) are conducted in the underlying Tuple Access layer, so that we can incrementally consume qualified tuples in the order of their scores (cf. Section 4.1).

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 4.2.1 and 4.2.2). In order to navigate such space efficiently, we identify the tuple retrieval order that can be used to construct the space incrementally (Section 4.2.1). 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 when finding an answer whose probability is not below the probability upper-bound of all unexplored search paths (Section 4.2.3). We show that our algorithms minimize the number of consumed tuples, and the size of the materialized space to evaluate our queries (Section 4.2.4).

4.2.1 Problem Space

We start our space formulation by defining the search state:

Definition 10 [**Top-l State**] *A top-l state s_l is a* prefix *of* length *l in one or more possible worlds ordered on a scoring function $\mathcal{F}$.* □

That is, a top-l state represents an l-length tuple vector ranked on a user-defined scoring function $\mathcal{F}$, and appearing as the top-l vector in at least one possible world.

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, i.e., 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 assume an uncertain database $\mathcal{D}$ following the tuple level uncertainty model. 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 s_l tuples and the absence of I_{s_l} tuples, i.e., $\mathcal{P}(s_l) = \Pr(s_l \wedge \neg I_{s_l})$, which gives the probability that s_l tuples are the top-l vector in the possible worlds space. For example consider Figure 2.1, where the scoring function $\mathcal{F}$ is simply the *Speed* attribute. For a state

$s_2 = \langle t1, t5 \rangle$, we have $I_{s_2} = \{t2\}$ and hence $\mathcal{P}(s_2) = \Pr((t1.e \wedge t5.e) \wedge (\neg t2.e)) = 0.4 \times 0.6 \times 0.3 = 0.072$. This probability is the same as $\Pr(PW^4)$, which is the only possible world having $\langle t1, t5 \rangle$ as the top-2 tuples. In general, state probability is computed by the *Rule Engine* component of our framework based on tuple probabilities, and their dependencies (Section 4.1).

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

Tuple Retrieval Order. Based on our *Tuple Access*, and *Available Dependency Information* assumptions (Section 4.1), we show that retrieving tuples in sorted score order is *necessary* and *sufficient* to get non-trivial bounds on the probabilities of possible complete states. Such bounding is crucial for early query termination, i.e., 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 5 [**Arbitrary Order**] *Consider Figure 2.1. 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 of* s_2 *tuples, and having a probability of 1, which would make* $\mathcal{P}(s_2) = 0$*. Clearly, we cannot conclude the search goal early, i.e., without fully inspecting all space states, if we cannot compute non-trivial (greater than 0) lower-bounds on states' probabilities.*

Example 6 [**Probability Order**] *Consider Figure 2.1. 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 5* *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 non-trivial 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 dependent on tuples in 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 dependencies exists or not, since t is not retrieved yet from the* Tuple Access Layer*.*

Example 7 [Score Order] *Consider Figure 2.1. 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 non-retrieved 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 4.1, 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-based order in retrieving tuples from the *Tuple Access Layer*. Rank-aware query processing techniques can be used in the *Tuple Access Layer* to incrementally provide tuples in score order by exploiting score indexes and rank-aware query operators [34, 45]. Theorem 2 formally proves the superiority of score-based retrieval.

Theorem 2 *Under the* Tuple Access*, and* Available Dependency Information *assumptions, retrieving query output tuples in score order is (i) sufficient; and (ii) necessary to compute non-trivial bounds on state probabilities.*

Proof. Let X be a non-empty subset of tuples retrieved in score order. For any state s_l whose tuples form a subset of X, we also have $I_{s_l} \subseteq X$. That is, the set I_{s_l}

is completely known based on X. Hence, X is *sufficient* to precisely compute $\mathcal{P}(s_l)$, and thus (i) follows.

We prove (ii) by showing that a tuple order, different from score-based order, fails to provide non-trivial bounds on state probabilities. Let Y be a non-empty subset of tuples retrieved in any order different from the score order. For any state s_l whose tuples form a subset of Y, the set I_{s_l} is not completely known based on Y. Assume a yet non-retrieved tuple $t \notin Y$, such that $t \in I_{s_l}$, and $\Pr(s_l \wedge \neg t) = 0$. If such tuple exists, we would have $\mathcal{P}(s_l) = 0$. Hence, based on Y, the only *safe* lower-bound on state probability is 0, and thus (ii) follows. $\square$

Relaxing the assumptions of Theorem 2 makes other tuple retrieval orders useful in evaluating our queries. We discuss such orders in Section 4.3.2.

4.2.2 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 2.1, 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 our space states.

Property 2 [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.* $\square$

Property 2 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, call it a *parent* state. After retrieving a new tuple t from the *Tuple Access Layer* in score order, we would like to generate two states from s_l representing the two possibilities of including and excluding t at position $(l+1)$ in possible top-k answers generated from s_l. We thus extend s_l into two *child* states: (1) s_{l+1}: a state composed of the tuple vector of s_l appended by t, where $I_{s_{l+1}} = I_{s_l}$; and (2) $\acute{s}_l$: a state with the same tuple vector as s_l, where we define $I_{\acute{s}_l} = I_{s_l} \cup \{t\}$ [†]. Based on Property 2, both $\mathcal{P}(s_{l+1})$ and $\mathcal{P}(\acute{s}_l)$ cannot exceed $\mathcal{P}(s_l)$. In addition, $\mathcal{P}(\acute{s}_l) + \mathcal{P}(s_{l+1}) = \mathcal{P}(s_l)$.

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) $s_3 = \langle t1, t2, t4 \rangle$, where $I_{s_3} = \{t3\}$, and hence $\mathcal{P}(s_3) = \Pr((t1.e \wedge t2.e \wedge t4.e) \wedge (\neg t3.e))$; and (2) $\acute{s}_2 = \langle t1, t2 \rangle$, where $I_{\acute{s}_2} = \{t3, t4\}$, and hence $\mathcal{P}(\acute{s}_2) = \Pr((t1.e \wedge t2.e) \wedge (\neg t3.e \wedge \neg t4.e))$

We illustrate state generation using the interaction of our framework components depicted by Figure 4.2, which describes UTop-Prefix query processing over the database in Figure 2.1. 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 request its probability from the *Rule Engine*, which responds back with the value 0.072. These computed probability values are used to guide the search in the space in order to locate the most

[†]The state $\acute{s}_l$ is the parent state of all possible top-k answers having a tuple different from t at position $l+1$. Hence, we can think of the tuple vector of $\acute{s}_l$ as the tuple vector of s_l appended by a yet unknown tuple t' following t in score order, which will make setting $I_{\acute{s}_l} = I_{s_l} \cup \{t\}$ consistent with our previous definition of the set I. Moreover, since t' is unknown at this point, computing $\mathcal{P}(\acute{s}_l)$ as $\Pr(s_l \wedge \neg I_{\acute{s}_l})$ correctly upper bounds the probability of any top-k answer generated from s_l and having a tuple different from t at position $l+1$.

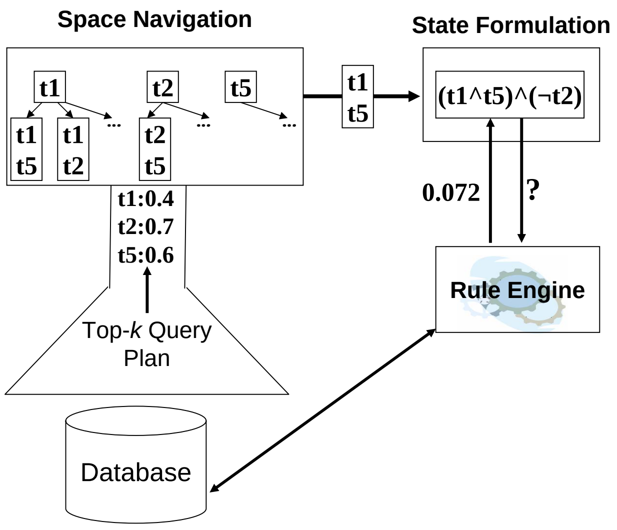


Figure 4.2: Interaction of framework components

probable ranking of query results. In Section 4.2.3, we give efficient search algorithms that partially materialize the problem space by retrieving the least possible number of tuples, and materializing the least possible number of search states.

Cost Metric. Based on our problem definition in Section 3.4, 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 with typically small value, 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.

4.2.3 Navigating the Search Space

In this section, we give the details of our probability-guided search algorithms for computing UTop-Prefix and UTop-Rank queries. Note that UTop-Set and UTop-Prefix queries return identical answers under tuple level uncertainty (cf. Section 3.4).

Processing UTop-Prefix Query. We describe Algorithm `OPTU-Top`k, our processing algorithm for UTop-Prefix(k) query. 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, i.e., 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.

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 non-materialized state, since any non-materialized state is an extension of the empty state (cf. Property 2).

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 4.2.2), 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 non-materialized 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 2.

In addition to extending the space *lazily*, i.e., only the top $\mathcal{Q}$ state is extended at each step, Algorithm 1 also applies a *pruning criterion* to significantly cut down

Algorithm 1 `OptU-Top`k

```
UTop-Prefix ($source$ : Score-ordered tuple stream, $k$ : result size)
1  $\mathcal{Q} \leftarrow$ empty p-queue of states ordered on probabilities ; $d \leftarrow 0$ ; $max_p \leftarrow 0$;
2  Insert $s_{0,0}$ into $\mathcal{Q}$  {*initialize $\mathcal{Q}$ with an empty state*}
3  while ($\mathcal{Q}$ is not empty)
4      do
5        $s_{l,i} \leftarrow$ dequeue ($\mathcal{Q}$)  {*get the state with the highest probability*}
6        if ($l = k$)
7          then return $s_{l,i}$   {*highest probability state is complete, then terminate*}
8          else
9              $t \leftarrow$ NULL
10             if ($i = d$ )
11               then
12                   if ($source$ is not exhausted)
13                     then
14                         $t \leftarrow$ get next tuple from $source$ ; $d \leftarrow d + 1$
15               else
16                   $t \leftarrow$ tuple at pos $i + 1$ in seen tuples buffer
17             if ($t$ is not NULL)
18               then
19                   Extend $s_{l,i}$ using $t$ into $s_{l,i+1}$, $s_{l+1,i+1}$  {*See Section 4.2.2*}
                     { *prune loser states*}
20                   if ($l + 1 = k$ AND $\mathcal{P}(s_{l+1,i+1}) > max_p$) then $max_p \leftarrow \mathcal{P}(s_{l+1,i+1})$
21                   if ($\mathcal{P}(s_{l+1,i+1}) \geq max_p$) then Insert $s_{l+1,i+1}$ into $\mathcal{Q}$
22                   if ($\mathcal{P}(s_{l,i+1}) > max_p$) then Insert $s_{l,i+1}$ into $\mathcal{Q}$
```

the size of $\mathcal{Q}$ (line 20): The algorithm maintains a variable max_p representing the maximum probability of a complete state reached so far. Any other reached state with probability smaller than max_p can be safely pruned (i.e., not inserted in $\mathcal{Q}$), based on Property 2.

In Section 4.2.4, we analyze the complexity and performance guarantees of Algorithm `OPTU-Top`k.

Algorithm 2 `OptU-`k`Ranks`

```
UTop-Rank ($source$ : 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  $reported \leftarrow 0$ ; $depth \leftarrow 1$; $space \leftarrow \emptyset$  $\{$current set of materialized states $\}$
 4  while ( $source$ is not exhausted AND $reported < k$)
 5      do
 6         $t \leftarrow$ next tuple from $source$
 7         Extend all states in $space$ based on $t$
 8         for ($i$=1 to $min(k, depth)$)
 9             do
10                Set $ubound_i \leftarrow \sum_{j<i} Z_j$ based on $space$
11                if ($answer_i$ is previously reported)
12                   then Continue
13                Compute $P_{t,i}$
14                if (  ($answer_i$ is null) OR
                    ($answer_i$ is not null AND $P_{t,i} > answer_i.prob$)  )
15                   then  $\{$found a better answer at rank $i\}$
16                         $answer_i \leftarrow t$
17                         $answer_i.prob \leftarrow P_{t,i}$
18                         if ($answer_i.prob \geq ubound_i$)
19                            then  $\{$termination condition satisfied at rank $i\}$
20                                  Report $answer_i$
21                                  $reported \leftarrow reported + 1$
22         $depth \leftarrow depth + 1$
```

Processing UTop-Rank Query. We describe `OPTU-`k`Ranks`, our query processing algorithm for UTop-Rank(i, i) query, for $i = 1$ to k (extending the algorithm to compute UTop-Rank(i, j) query answer, where $1 \leq i \leq j \leq k$ is straightforward). Let t be the n^{th} tuple in score-ordered 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-`k`Ranks` builds on 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 upper-bound on the probability of any non-retrieved tuple being at rank i.

Algorithm 2 describes the details of `OPTU-`k`Ranks`. 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, as u has a smaller score than t. 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 5). Let t^* be the current UTop-Rank(i, i) query answer. The termination condition of Algorithm `OPTU-`k`Ranks`, for rank i, is thus $P_{t^*,i} \geq \sum_{j<i} Z_j$.

We analyze the complexity and performance guarantees of Algorithm `OPTU-`k`Ranks` in Section 4.2.4.

4.2.4 Analysis 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 [30] over the space of all possible top-k answers. We start by giving a description of the $\mathcal{A}^*$ algorithm [30]. We then show how `OptU-Top`k is effectively an instance of $\mathcal{A}^*$.

$\mathcal{A}^*$ Search Algorithm. Assume a finite non-empty set of states $\mathcal{S}$ that is represented by a weighted directed graph, where each graph node denotes a state $s \in \mathcal{S}$, and the weight of the edge connecting two states s and s' represents the cost of moving from s and s'. Let $s_0 \in \mathcal{S}$ be a designated start state, and $S_g \subseteq \mathcal{S}$ be a non-empty set of designated goal states. The objective of Algorithm $\mathcal{A}^*$ is to find the path with the *least cost* from s_0 to a goal state $s_g \in S_g$, where path cost is the summation of the weights of the edges involved in the path.

$\mathcal{A}^*$ applies a best-first search strategy by traversing the states based on 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 the best goal state s_g reachable from s. The state with the least $f(\cdot)$ value is visited next, until the goal is reached. It was proven in Theorem 1 in [30] that if the function f is *admissible*, i.e., it does not *overestimate* actual path cost, the $\mathcal{A}^*$ 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, given available domain knowledge about the problem's space, if $\mathcal{A}^*$ uses the tightest possible estimate $h(\cdot)$ 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 $\mathcal{A}^*$. This means that $\mathcal{A}^*$ is optimal in the number of visited states.

Reducing `OptU-Top`k to $\mathcal{A}^*$ Search. 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. The state space in Algorithm `OptU-Top`k can be represented as a tree whose root is the initial state, and leaves are the complete states (i.e., states with length k). For example, Figure 4.3 shows the state space corresponding to the given score-ordered tuple stream (we show the probability of the existence event corresponding to each tuple). We assume independent tuple events to simplify the computation of state probabilities, which are given by the number beside each state. For $k = 2$, the shaded tree nodes show two possible complete states. Other possible complete states can be reached by completing the generation of the space. Such full generation of the space can be avoided in `OptU-Top`k by early pruning of tree paths, as we show next.

[†]For example, if states represent cities, and the cost of moving from one state to another is the traveling distance, the tightest possible estimate $h(s)$ is given by the smallest airline distance from s to a goal state [30].

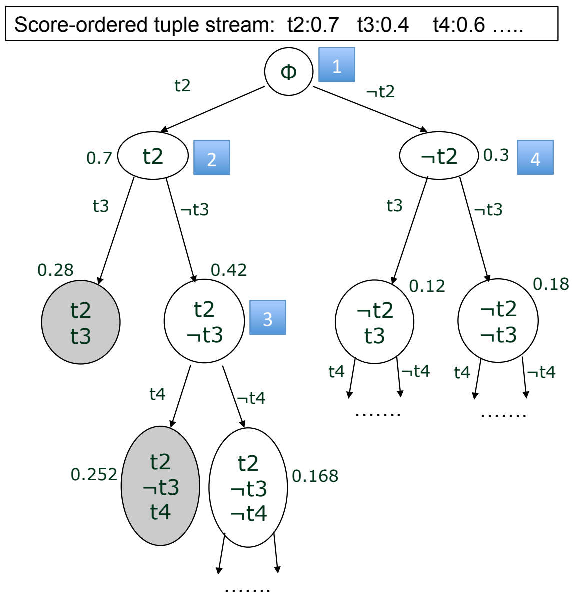


Figure 4.3: State space of Algorithm `OptU-Top`k (For $k = 2$, the order of opening a state is given by the square beside each node, and complete states are represented by shaded nodes)

Algorithm `OptU-Top`k *opens* a non-complete state s_l by extending s_l using the next tuple from the score-ordered input stream (cf. Section 4.2.2), and *closes* (i.e., prunes) a non-complete state s_l if a complete state s_k is reached, where $\mathcal{P}(s_k) > \mathcal{P}(s_l)$ (cf. Section 4.2.3). The selection of which state to open next is based on computed upper-bounds of the probabilities of possible complete states. For example in Figure 4.3, the algorithm opens the states in the order shown in the square beside each node, and terminates with $\langle t2, t3 \rangle$ as the most probable top-2 answer. We next illustrate, using the constructs of $\mathcal{A}^*$ search, computing the probability upper-bounds of a given state.

The probability of state s_l represents the *cost* of reaching s_l from the initial state. Our objective is to find a complete state with the highest probability among all possible complete states. Let $A(s_l)$ be the conjunction of tuple events on the tree path from the initial state to s_l. Let $B(s_l)$ be the conjunction of tuple events on the tree path from s_l to the most probable complete state s_k reachable from s_l. Based on the principles of probability theory, it follows that $\mathcal{P}(s_k) = \Pr(A \wedge B) \leq$

$\min(\Pr(A), \Pr(B))$.

In the terminology of $\mathcal{A}^*$, for a state s_l, let $g(s_l) = \mathcal{P}(s_l)$, and $h(s_l)$ be an upper-bound on the probability of the conjunction of tuple events on the path from s_l to a complete state (i.e., $h(s_l)$ is the maximum cost incurred in reaching a complete state from s_l). Further, define $f(s_l)$ as $f(s_l) = \min(g(s_l), h(s_l)) = \min(\mathcal{P}(s_l), h(s_l))$. It follows that $f(s_l)$ is an upper-bound on the probability of a complete state reachable from s_l.

The remaining point is choosing an *admissible* upper-bounding function $h(s_l)$. A clear choice is to set $h(s_l) = 1$ for any state s_l, which means that we have $f(s_l) = \mathcal{P}(s_l)$. We further show in Lemma 1 that, under our assumptions, such choice of $f(s_l)$ is the tightest upper-bound on the probability of a complete state reachable from s_l.

Lemma 1 *Under the* Tuple Access*, and* Available Dependency Information *assumptions, $\mathcal{P}(s_l)$ is the tightest upper-bound on the probability of a complete state reachable from s_l.*

Proof. Based on Property 2, $\mathcal{P}(s_l) \geq \mathcal{P}(s_k)$ for any complete state s_k reachable from s_l. Hence, $\mathcal{P}(s_l)$ is admissible since it does not *underestimate* the probability of the most probable complete state reachable from s_l.

Further, we show that $\mathcal{P}(s_l)$ is the tightest bound. That is, the probability of the most probable complete state reachable from s_l can be exactly the same as $\mathcal{P}(s_l)$. Under the *Tuple Access* and *Available Dependency Information* assumptions, assume that there exist $k - l$ non-retrieved tuples that can extend the state s_l, where each tuple has a probability 1, and is independent of all other tuples. In this case, the probability of a complete state reachable from s_l is the same as $\mathcal{P}(s_l)$. It follows that the tightest upper-bound on the probability of a complete state reachable from s_l is $\mathcal{P}(s_l)$. □

Correctness Proof. Based on the admissibility of using non-complete state probabilities to upper-bound the probabilities of reachable complete states, we next give a formal proof of correctness of Algorithm `OptU-Top`k.

Lemma 2 *At any point during Algorithm* ***OptU-Top****k operation, the most probable complete state is reachable from some opened state.*

Proof. We need to show that any closed state by Algorithm `OptU-Top`k does not lead to the most probable complete state.

Assume that a non-complete state x is closed by Algorithm `OptU-Top`k where x leads to the most probable complete state x^*. Hence, at this point there exists a complete state y^* with $\mathcal{P}(y^*) > \mathcal{P}(x)$. However, based on Property 2, we have $\mathcal{P}(x) \geq \mathcal{P}(x^*)$, and thus we also have $\mathcal{P}(y^*) > \mathcal{P}(x^*)$. This contradicts the initial assumption that x^* is the most probable complete state. $\square$

Theorem 3 *Algorithm* ***OptU-Top****k correctly finds the most probable complete state.*

Proof. The proof is similar to the correctness proof of the $\mathcal{A}^*$ algorithm ([30], Theorem 1). We prove the stated claim by eliminating three possible cases:

[Case 1] *Termination is at a non-complete state.* This case contradicts the termination condition of `OptU-Top`k, which requires reporting a complete state (i.e., a state of length k). Hence, we can eliminate this case.

[Case 2] *Termination is not reached.* This case can be eliminated based on how `OptU-Top`k generates the search space. Let $t_{(1)}, t_{(2)}, \ldots, t_{(n)}$ be the input tuples in score order. A state that is opened (extended) using the tuple $t_{(i)}$ results in two new states to be later extended using the tuple $t_{(i+1)}$. Hence, each state is opened exactly once (i.e., there are no cycles among states). The set of opened states can thus be organized in a tree (e.g., Figure 4.3) whose maximum depth is n (the number of tuples), and leaves are either complete or closed states. In the worst case, the full tree needs to be constructed. Since the number of tuples in the underlying database is finite, Algorithm `OptU-Top`k must reach termination in a finite time.

[Case 3] *Termination is at a complete state that is not the most probable complete state.* Assume that Algorithm `OptU-Top`k terminated at a state x^*, while the most probable complete state is another state $y^* \neq x^*$. Hence, based on Lemma 2, there exists some opened state y such that y^* can be reached from y, which means that $\mathcal{P}(y) \geq \mathcal{P}(y^*)$ (cf. Property 2). However, based on the termination condition of Algorithm `OptU-Top`k, we have $\mathcal{P}(x^*) > \mathcal{P}(s)$ for any opened state s. It follows that $\mathcal{P}(x^*) > \mathcal{P}(y) \geq \mathcal{P}(y^*)$, which contradicts the assumption that y^* is the most probable complete state. □

Similarly, we prove the correctness of Algorithm `OPTU-`k`Ranks` in Lemma 3.

Lemma 3 *Algorithm OPTU-kRanks correctly finds the most probable tuple to appear at each rank i in $1 \ldots k$.*

Proof. Based on the goal of the algorithm, we need to eliminate two cases:

[Case 1] *Termination is not reached.* Similar to the proof of Theorem 3, this case can be eliminated based on the finite number of input tuples, and the tree structure of the search space.

[Case 2] *Termination for rank i is at a tuple that is not the most probable tuple to appear at rank i.* From possible worlds semantics, we have for a tuple t, $P_{t,i} = \sum_{s_i} \mathcal{P}(s_i)$, where s_i is any possible state (i.e., a prefix of at least one possible world) with length i and having t as the tuple ranked last. Based on the admissibility of using $\mathcal{P}(s_l)$ as an upper bound on the probability of any child state generated from s_l (Lemma 1), it follows that or any non-retrieved tuple t, we have $P_{t,i} \leq \sum_{j<i} Z_j \quad \ldots^{(\ddagger)}$, where Z_j is the summation of the probabilities of currently opened states with length j. Assume that Algorithm `OptU-Top`k reports tuple t^* as the most probable tuple to appear at rank i. Hence, based on the algorithm's termination condition, we have $P_{t^*,i} \geq \sum_{j<i} Z_j$. Let u be a non-retrieved tuple with $P_{u,i} > P_{t^*,i}$. However, this means that $P_{u,i} > \sum_{j<i} Z_j$, which is a contradiction with $^{(\ddagger)}$. □

Performance Guarantees. We next analyze the performance guarantees of our proposed search algorithms.

We first define a class of search algorithms $\mathcal{C}$ to which we compare our proposed algorithms. Let $\mathcal{C}$ be the class of algorithms that search the problem space defined in

Section 4.2.1 under the assumptions given in Section 4.1 to correctly find a complete state with the highest probability (i.e., members of $\mathcal{C}$ can open states in any order, not necessarily based on state probability, until reaching the most probable complete state).

Theorem 4 *Any algorithm $\mathcal{A} \in \mathcal{C}$ must open all the states opened by `OptU-Top`k.*

Proof. The proof is similar to the optimality proof of the $\mathcal{A}^*$ algorithm ([30], Theorems 2 and 3).

We consider two possible cases, and give the proof by contradiction:

[Case 1] *State probabilities have no ties.* Assume that some state s_l is opened by `OptU-Top`k but not by $\mathcal{A}$. Let the answer reported by $\mathcal{A}$ be x^*, and the answer reported by `OptU-Top`k be y^*. Since both $\mathcal{A}$ and `OptU-Top`k are correct and we have no ties in state probabilities, it follows that $x^* = y^*$. Since s_l is opened by `OptU-Top`k, then we know that, at some point, s_l was the most probable state among all opened states, and that y^* is either an extension of s_l or an extension of another non-complete state with a smaller probability. It follows that $\mathcal{P}(s_l) > \mathcal{P}(y^*)$. Hence, we have $\mathcal{P}(s_l) > \mathcal{P}(x^*) \quad \ldots^{(\dagger)}$. However, since $\mathcal{A}$ has not opened s_l, and $\mathcal{A}$ is correct, it follows that $\mathcal{P}(s_l) < \mathcal{P}(x^*) \quad \ldots^{(\ddagger)}$. By contradiction of (†) and (‡), Theorem 4 is proven for [Case 1].

[Case 2] *State probabilities can have ties.* Let `OptU-Top`k^* be the set of algorithms identical to `OptU-Top`k, but each algorithm resolves ties in state probabilities (when picking the next state to open) in a different way, such that members of `OptU-Top`k^* cover all possible tie-breaking rules. In the following, we show that there exists some member $T^* \in$ `OptU-Top`k^* such that any algorithm $\mathcal{A} \in \mathcal{C}$ must open all the states opened by T^*. Assume $\mathcal{A}$ opens the same states as T^*, until some state s_l that is opened by T^* but not by $\mathcal{A}$. Let the next state opened 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 only consider possibility (ii). Let $T^* = T^*_1$, where

$T_1^* \in$ `OptU-Top`k^*, and T_1^* picks $\acute{s}_l$ as the next state to open. By repeating the above procedure at each state opened by T^* but not by $\mathcal{A}$, we can show that there exists a member $T^* \in$ `OptU-Top`k^*, such that any state opened by T^* must be opened by $\mathcal{A}$ as well, and hence Theorem 4 is proven for [Case 2]. □

It follows from Theorem 4 that $\mathcal{A}$ must consume at least the same number of tuples as `OptU-Top`k (or a member of `OptU-Top`k^* in case of probability ties) since $\mathcal{A}$ opens at least the same states as `OptU-Top`k.

The performance guarantees of Algorithm `OPTU-`k`Ranks` are defined in terms of the number of retrieved tuples, since `OPTU-`k`Ranks` materializes all space states based on the retrieved tuples. Our result is given in Theorem 5 below. The assumptions on the class of search algorithms $\mathcal{C}$ are the same as stated in our previous definition of $\mathcal{C}$, however the goal is to find the most probable tuple to appear at each rank in $1 \dots k$.

Theorem 5 *Any algorithm $\mathcal{A} \in \mathcal{C}$ must retrieve at least the same number of tuples retrieved by Algorithm **OPTU-kRanks** to compute UTop-Rank(i, i) query answer, for $i = 1$ to k.*

Proof. We prove the stated claim by showing that the computed probability upper bound on $P_{u,i}$, for a non-rerieved tuple u, is tight, and hence each retrieved tuple by `OPTU-`k`Ranks` is necessary for termination.

At any step during `OPTU-`k`Ranks` processing, let ω_j be the set of opened 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 UTop-Rank(i, i) query answer, while the termination condition is not satisfied, i.e., $P_{t,i} < \sum_{j<i} Z_j$. Assume the next i tuples, non-retrieved yet, are $u_1, \dots, u_i$, 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 $l \neq j-1$ does not change. Hence, it follows that $P_{u_i,i} = \sum_{j<i} Z_j$. Then, $\sum_{j<i} Z_j$ upper-bounds $P_{u,i}$ for a non-retrieved tuple u. Further, assuming a higher upper-bound would be loose, based on Property 2 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 $P_{u,i}$.

Assume an algorithm $\mathcal{A} \in \mathcal{C}$ that reports UTop-Rank(i, i) query answer, while retrieving less number of tuples than `OPTU-`k`Ranks`. At the point $\mathcal{A}$ terminates, the termination condition of `OPTU-`k`Ranks` is not reached based on the set of tuples retrieved by $\mathcal{A}$. Since `OPTU-`k`Ranks` is correct (Lemma 3) and $\mathcal{A}$ is correct, then they must return identical answers. However, we can construct a case, similar to the above, in which a non-retrieved tuple u is the UTop-Rank(i, i) query answer that is reported by `OPTU-`k`Ranks`. Since $\mathcal{A}$ terminates before reaching u, it must have reported a different answer $u' \neq u$, which contradicts $\mathcal{A}$ being correct. □

Complexity Analysis. We next give time and space complexity analysis. We denote with 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.

According to our *Tuple Access* assumption in Section 4.1, we focus on analyzing the space search algorithms implemented in the *Probabilistic Ranking Layer* by assuming that the *Tuple Access Layer* indexes the tuples qualified by relational processing (e.g., filtering and joining) in the order of their scores. Rank-aware query processing (cf. Section 2.1) can efficiently provide incremental score-ordered retrieval of these tuples. In the absence of such tuple access methods, a complexity of $O(m\ log(m))$, where m is the total number of qualified tuples in the *Tuple Access Layer*, needs to be added to our complexity bounds in order to account for the cost of sorting tuples on score.

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 algorithm description in Section 4.2.3, 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. 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, i.e., 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 in Section 4.4 through experiments conducted on different data distributions and tuple dependencies.

Algorithm `OPTU-`k`Ranks`. The lower-bound time complexity of `OPTU-`k`Ranks` 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 correlation between the distributions of tuple scores and probabilities leads to finding high-probability tuples early in the score-ordered tuple stream, which allows for early termination.

2. Complexity of tuple dependencies: Complex dependencies negatively affect the performance by increasing the probability computation cost in the *Rule Engine*. For example, a dense set of dependencies may entail tracking tuple dependencies on a long chain of parent tuples, which negatively affects the performance.

3. Distribution of tuple probabilities: Distribution of tuple probabilities affects 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 our experimental study in Section 4.4.

4.3 Relaxing Framework Assumptions

Our framework assumptions, discussed in Section 4.1, 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 4.3.1, we discuss the effect of relaxing the *Available Dependency Information* assumption, where we assume all tuples are known to be independent. In Section 4.3.2, we discuss the effect of relaxing our score-ordered *Tuple Access* assumption, where we use different tuple orders.

4.3.1 Exploiting Known 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 looser 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.

UTop-Prefix Query under Independence. In some cases, the given query and the underlying dependency model may provide additional (free) information on potential dependencies among query output tuples before query evaluation starts. For example, a simple selection query that involves a single relation with independent tuples does not generate dependencies among query output tuples.

Under the independence of tuples' existence events (tuple independence, for short), we can aggressively prune incomplete states, early in our search, to keep only the states that can lead to query answer. The pruning criterion is formulated in Lemma 4.

Lemma 4 *Under tuple independence, a state y_m can be safely pruned if there exists another state x_n with $\mathcal{P}(x_n) > \mathcal{P}(y_m)$, where x_n and y_m are both maintained after seeing the same set of score-ranked tuples and $n \geq m$.*

Proof . Let t_{next} be first unseen tuple by x_n and y_m. Let Λ_x be the set of all possible sequences of tuple existence/absence events, where each sequence in Λ_x starts from t_{next}, and contains exactly $k-n$ existence events with the last event being an existence event. That is, appending the events of a sequence $\lambda \in \Lambda_x$ to the state x_n gives a complete state of length k. Based on tuple independence, we have $\Pr(\lambda) = \prod_{e \in \lambda} \Pr(e)$, for any sequence $\lambda \in \Lambda_x$. Let $\lambda_x = argmax_{\lambda \in \Lambda_x} \Pr(\lambda)$. Similarly, define Λ_y and λ_y for the state y_m.

Let p_x and p_y be the probabilities of the most probable complete states reachable from x_n and y_m, respectively. It follows that $p_x = \mathcal{P}(x_n) \cdot \Pr(\lambda_x)$, and similarly $p_y = \mathcal{P}(y_m) \cdot \Pr(\lambda_y)$. We give the lemma proof by considering two cases:

[Case 1] $n = m$. In this case we have $\Lambda_x = \Lambda_y$, and hence $\lambda_x = \lambda_y$. Since we have $\mathcal{P}(x_n) > \mathcal{P}(y_m)$ as a given, it follows that $p_x > p_y$.

[Case 2] $n > m$. In this case we have $\forall \lambda \in \Lambda_y$ there exists a sequence $\lambda' \in \Lambda_x$ such that λ' is part of λ. The reason is that in order to generate a sequence with $k - m$ existence events, we must first obtain some sequence with $k - n$ existence events. It follows that $\forall \lambda \in \Lambda_y$ there exists a sequence $\lambda' \in \Lambda_x$ with $\Pr(\lambda') \geq \Pr(\lambda)$. Hence, we have $\Pr(\lambda_x) \geq \Pr(\lambda_y)$, and it follows that $p_x > p_y$.

In the two previous cases, since $p_x > p_y$, we can safely prune y_m since the most probable complete states reachable from y_m is smaller in probability than the most probable complete state reachable from x_n. $\square$

Lemma 4 formulates a setting where an incomplete state can be early pruned from the search space. The reason is that, based on the conditions in Lemma 4, 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 sequence of existence/absence events of unseen tuples to reach *complete* states. However, x_n will reach a *complete* state at most in the same number of steps as y_m, because $n \geq m$. Since $\mathcal{P}(x_n) > \mathcal{P}(y_m)$, and based on Property 2, 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 exploits Lemma 4 by grouping states into equivalence classes based on their *lengths*. Algorithm `IndepU-Top`k keeps *at most one* state for each *length* $0 \dots k$ in a candidate set. The candidate set is extended upon retrieving each new tuple. `IndepU-Top`k terminates when at least k tuples have been retrieved, and the probability of any current state is not above the probability of the current *complete* candidate.

Consider for example the score-ranked tuple stream in Figure 4.4 (fractions indicate probabilities, and scores are omitted for brevity), where we are interested in UTop-Prefix(3) answer. We represent each state s_l with its tuple vector, and distinguish tuples seen but not included by s_l with the $\neg$ symbol. In step (a), after retrieving the first tuple $t1$, we construct two states $\langle \neg t1 \rangle$ 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 4.

Step (c) continues in the same manner by updating the candidate set based on tuple $t3$, and pruning the less probable candidate from each equivalence class. Note that the candidate $\langle \neg t1, \neg t2, \neg t3 \rangle$ is pruned because there is another candidate

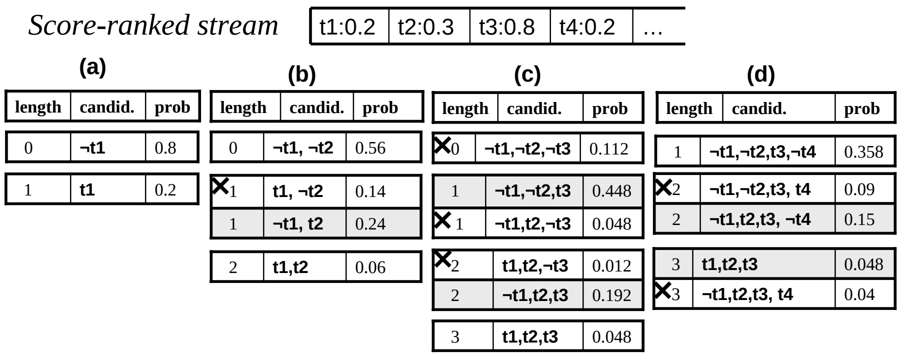


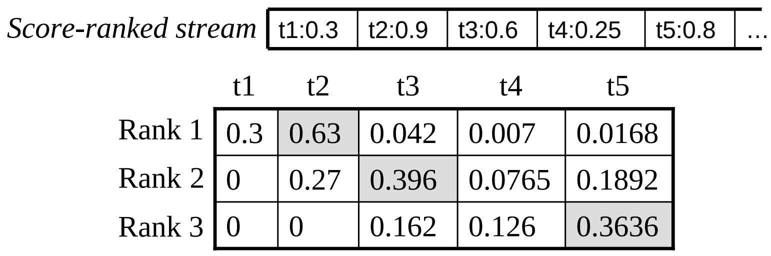
Figure 4.4: `IndepU-Top`k processing steps

$\langle \neg t1, \neg t2, t3 \rangle$ with a larger length and higher probability. In step (c) we have constructed 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 4 results in significant performance improvements as we illustrate in our experiments in Section 4.4.

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 $O(k)$.

UTop-Rank Query under Independence Under tuple independence, a UTop-Rank query exhibits the *optimal substructure* property, i.e. the optimal solution of the larger problem is constructed from solutions of smaller problems. This allows using a dynamic programming algorithm. We now describe `IndepU-`k`Ranks`, a dynamic programming algorithm to answer UTop-Rank query under the independence of tuple events.

Consider the example depicted by Figure 4.5, where we are interested in UTop-Rank(i, i) query answer, for $i = 1$ to 3. In the shown table, a cell at row i and column



	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

Figure 4.5: `IndepU-`k`Ranks` processing steps

x contains $P_{x,i}$ (the probability of tuple x to be at rank i). For a scoring function $\mathcal{F}$, the rank 1 probability of a tuple x is computed as $\Pr(x.e) \times \prod_{z:\mathcal{F}(x)<\mathcal{F}(z)} (1 - \Pr(z.e))$, 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 3 [Recurrence of Rank Probability] *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 3 is that under independence for tuple x to appear at rank i, we need only to consider the probability that x is consecutive to every other tuple y at rank $i-1$. This probability is computed using the probability that x exists, each 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 4.5, $P_{t2,2} = 0.9 \times 0.3 = 0.27$, while $P_{t3,2} = (0.6 \times 0.63) + (0.6 \times 0.1 \times 0.3) = 0.396$. The shaded cells indicate the UTop-Rank(i, i) answer, for $i = 1$ to 3. 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 ranked possible worlds. This means that we can report an answer from any row whenever the maximum probability in that row is greater than the row probability remainder. Notice also that the computation in each row depends solely on the row above.

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

$$M[i,t] = \begin{cases} \Pr(t.e) \times \prod_{z:\mathcal{F}(t)<\mathcal{F}(z)} (1-\Pr(z.e)) & if\ i=1 \\ \\ \Pr(t.e) \times \sum_{y:\mathcal{F}(y)>\mathcal{F}(t)} M[i-1,y] \times \prod_{z:\mathcal{F}(t)<\mathcal{F}(z)<\mathcal{F}(y)} (1-\Pr(z.e)) & if\ i>1 \end{cases} \tag{4.1}$$

For example in Figure 4.5, $M[2,3] = \Pr(t3.e) \times (M[1,2] + (1-\Pr(t2.e)) \times M[1,1])$. Algorithm `IndepU-`$k$`Ranks` returns a set of k tuples $\{t_1 \dots t_k\}$, where $t_i = argmax_x\ M[i,x]$.

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

4.3.2 Using Other Tuple Retrieval Orders

Our previous algorithms retrieve tuples in score order, which is necessary to compute non-trivial lower-bounds on state probabilities for general tuples dependencies, as we show in Section 4.2.1. 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 4.3.1 to combine score and probability orders in a *threshold* *algorithm*-like fashion. Threshold Algorithm (TA) [24] 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 2, state probability is monotone in the number of state tuple events, i.e., adding a tuple event to some state does not increase its probability. We thus use score list to construct states, and probability list to further shrink state probabilities by upper-bounding the probabilities of non-retrieved tuples in the score list. A high-level description of our TA-adaptation of Algorithm `IndepU-Top`k is 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-Top`k.

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 non-retrieved 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 s_l materialized by `IndepU-Top`k, 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 pruning loser states in `IndepU-Top`k using the same pruning criteria discussed in Lemma 4.

4. Repeat from step (1) until the termination condition of `IndepU-Top`k is met.

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 non-retrieved 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-Top`k.

When *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 early pruned by comparing their bounds, since a lower-bound on state probability is still trivially 0 (e.g., consider the case of a non-retrieved tuple in score order 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 non-trivial lower bounds on state probabilities. For example, assume the two top tuples in probability order are t_1 and t_2 with probabilities 1.0 and 0.1, and scores 1 and 10, respectively. Then, a state $s_2 = \langle t_2, t_1 \rangle$ has a probability lower-bound of $0.1 \times 1.0 \times (1 - 0.1)^{(N-2)}$, which is computed pessimistically by assuming all non-retrieved tuples have the highest possible probability (0.1), and higher scores than t_1. 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 non-retrieved tuples cannot be bounded. Non-trivial lower-bounds of state probabilities clearly allow for early query termination.

We note that, in practice, the lower-bounds computed using probability order can be very loose, i.e., they heavily underestimate actual probabilities, since the lower-bound is polynomial in the number of non-retrieved tuples which is usually a large number.

4.4 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 [45],

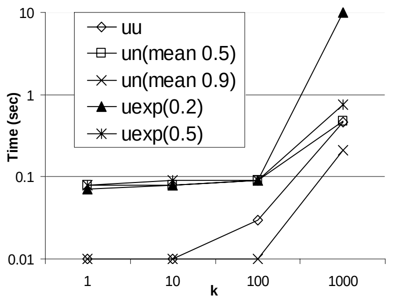


Figure 4.6: IndepU-Top$k$ time (different distributions)

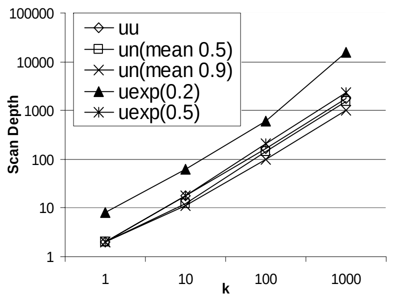


Figure 4.7: IndepU-Top$k$ depth (different distributions)

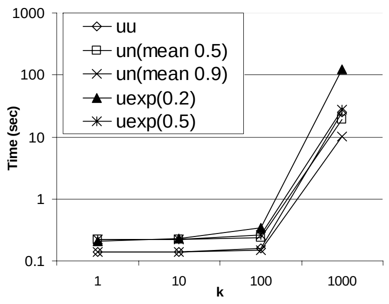


Figure 4.8: IndepU-kRanks time (different distributions)

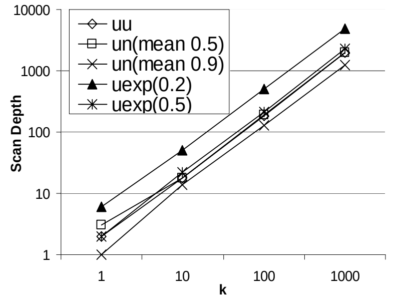


Figure 4.9: IndepU-kRanks depth (different distributions)

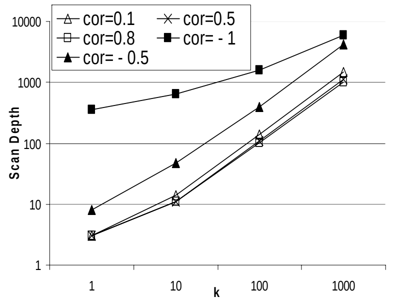


Figure 4.10: IndepU-Top$k$ (correlations)

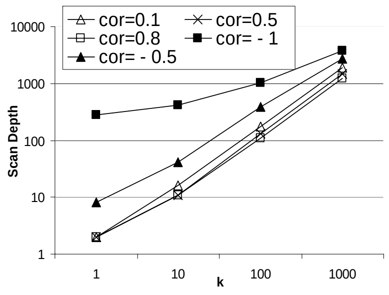


Figure 4.11: IndepU-kRanks (correlations)

which provides rank-awareness support. We used synthesized datasets generated by R [53]. 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.

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 independent of the underlying source of score-ranked tuples as we discuss in Section4.1.

The Naïve Approach

We illustrate the infeasibility of applying the naïve approach of *materializing* possible worlds space, *sorting* each world individually, and *merging* identical top-k answers. Due to space explosion, we applied this approach to small databases of sizes less than 30 tuples with different sets of generation rules. The *materialization* phase was the bottleneck in this approach consuming, on the average, one order of magnitude longer times than the *merging* phase. For example, processing a database of only 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.

Effect of the Distribution of Tuples Probabilities

We evaluate here the effect of the distribution of tuples probabilities on execution time and 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 4.6 and 4.7 show the time and scan depth of `IndepU-Top`k, respectively, while Figures 4.8 and 4.9 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 find highly probable tuples frequently in the score-ranked stream. This allows obtaining strong candidates to prune other candidates aggressively, and thus terminate the search quickly. This scenario applies to *un(mean 0.9)* distribution pair where a considerable number of tuples are highly probable. The counter scenario applies to *uexp(0.2)* where the exponential rate of decay in 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

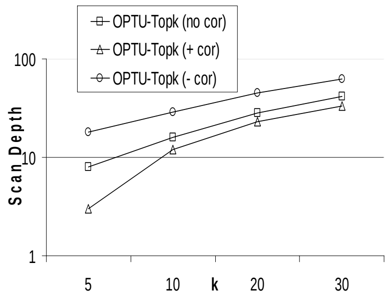


Figure 4.12: `OPTU-Top`k (depth)

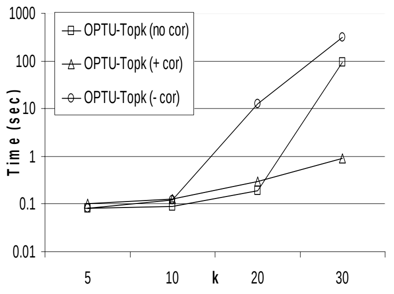


Figure 4.13: `OPTU-Top`k (time)

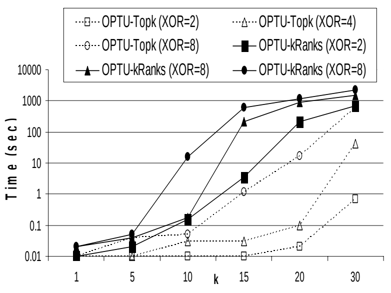


Figure 4.14: Rule set complexity

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-Top`k.

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 4.10 and 4.11 show the effect of the correlation coefficient on the scan depth of `IndepU-Top`k and `IndepU-`k`Ranks`, respectively. Increasing the correlation coefficient from 0.1 to 0.8 reduces the scan depth of `IndepU-Top`k and `IndepU-`k`Ranks` 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-`k`Ranks`. The effect on execution time is similar.

Evaluating Algorithms Performance

We evaluate the performance of `OPTU-Top`k. We used data sets with exclusiveness dependencies and uncorrelated, positively correlated, and negatively correlated score and probability distributions. Figures 4.12 and 4.13 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 distributions, 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 distributions. This can be explained based on the fact that for positive correlation, highly-probable states are reached quickly after retrieving a small number of tuples, while for negative correlation, more tuples need to be retrieved before concluding an answer, which leads to materializing more states.

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 dissertation, 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 4.14 shows the execution times of `OPTU-Top`k and `OPTU-`k`Ranks` with different *XOR* degrees. Increasing the *XOR* degree results generally in increasing the execution time with an average of one order of magnitude when going from *XOR=2* to *XOR=4*, or 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.

4.5 Summary and Lessons Learned

This chapter presents the first work that studies formulating and processing top-k queries under tuple level uncertainty model. The presented work provides insights on the interplay between the score and probability dimensions in this query type and gives efficient mechanisms to aggregate probabilities across possible worlds.

We modeled the 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, and uncertainty management in the same framework, while leveraging existing capabilities of RDBMSs. We also studied several relaxations of our assumptions, and discussed other problem variants.

Lessons Learned. Based on the study and the experiments presented in this chapter, we make the following high level observations:

- The correlation between score and probability distributions impact the performance of top-k query evaluation algorithms. Positive correlation usually leads to a small scan depth in the tuples' score order, while negative correlation usually leads to consuming a large number of tuples before termination. The number of consumed tuples has a direct impact on the size of materialized space.
- In practice, the sources of data uncertainty and preference scores may be independent, and hence we do not expect to see clear correlation between score and probability distributions in many cases.
- Incremental materialization of the dependencies among intermediate query results is an effective approach for processing probabilistic top-k queries. The reason is that the number of needed-to-see tuples is usually small, and all of their dependencies can be indexed in memory (e.g., using factor graphs). We observe reasonable performance by adopting this approach in our URank research prototype [66].

- The main performance bottleneck of our space search algorithms is the size of materialized answers space. Materializing a large portion of the space is infeasible in many cases. Developing techniques that lazily materialize the answers space guided by probability achieves the goal of early termination. Extending these techniques to compute tighter bounds on the probabilities of unexplored search paths clearly improves the performance. For example, by knowing that tuple events are independent, we can reduce the computation cost immensely by heavily pruning the search space, going from exponential worst-case complexity to polynomial worst-case complexity.

- Processing tuples in score order is both necessary and sufficient for early query termination under the assumption of incremental tuple access of the output of the relational query engine. This assumption fits query processing environments that pipeline query results as soon as they are ready. The argument for the optimality of score order cannot be made if we drop these assumptions. Using other tuple retrieval orders requires making different assumptions such as knowing the total number of tuples in query result beforehand.

- It is possible to combine both score and probability orders in a TA-like top-k algorithm. However the lack of a clear aggregation function in this case does not make the properties of the TA algorithm carry over smoothly to the probabilistic ranking problem, which motivates finding different evaluation techniques.

- Many variations of the probabilistic ranking problem have optimal substructure properties that allow designing efficient dynamic programming solutions.

Chapter 5

Ranking with Attribute Level Uncertainty

In this chapter we describe our proposal in [62, 63] to support ranking queries under attribute level uncertainty. We give a baseline exact evaluation algorithm in Section 5.1. We show how to early-prune tuples that are not part of query answer in Section 5.2. Next, we give a set of query evaluation algorithms in Sections 5.4 and 5.5. We discuss the problem of computing consensus ranking in Section 5.6. We present our experimental study in Section 5.7, and summarize the contents of this chapter in Section 5.8.

5.1 A Baseline Exact Algorithm

We describe a baseline algorithm that computes the queries in Section 3.4 by materializing the linear extensions space (we discuss computing URank-Agg query in Section 5.6). Algorithm 3 gives a simple recursive technique to build the linear extensions tree (Section 3.2). The first call to Procedure BUILD_TREE is passed the parameters PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$, and a dummy root node. A tuple $t \in \mathcal{R}$ is a *source* if no other tuple $\acute{t} \in \mathcal{R}$ dominates t. The children of the tree root will be the initial sources

Algorithm 3 Build Linear Extension Tree

```
BUILD_TREE (PPO($\mathcal{R}, \mathcal{O}, \mathcal{P}$) : $PPO$, $n$ : $Tree\ node$)
1  for each source $t \in \mathcal{R}$
2      do
3          $child \leftarrow$ create a tree node for $t$
4          Add $child$ to $n$'s children
5          ṔPO $\leftarrow$ PPO($\mathcal{R}, \mathcal{O}, \mathcal{P}$) after removing $t$
6          BUILD_TREE( ṔPO, $child$)
```

in $\mathcal{R}$, so we can add a source t as a child of the root, remove it from PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$, and then recurse by finding new sources in PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$ after removing t.

The space of all linear extensions of PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$ grows exponentially in $|\mathcal{R}|$. As a simple example, suppose that $\mathcal{R}$ contains m elements, none of which is dominated by any other element. A counting argument shows that there are $\Sigma_{i=1}^{m} \frac{m!}{(m-i)!}$ nodes in the linear extensions tree.

When we are only interested in tuples occupying the top ranks, we can terminate the recursive construction algorithm at level k, which means that our space is reduced from complete linear extensions to linear extensions' prefixes of length k. Under our probability space, the probability of each prefix is the summation of the probabilities of linear extensions sharing that prefix. We can compute prefix probabilities more efficiently as follows. Let $\omega^{(k)} = \langle t_1, t_2, \ldots, t_k \rangle$ be a linear extension prefix of length k. Let $T(\omega^{(k)})$ be the set of tuples not included in $\omega^{(k)}$. Let $\Pr(t_k > T(\omega^{(k)}))$ be the probability that t_k is ranked above all tuples in $T(\omega^{(k)})$. Let $F_i(x) = \int_{lo_i}^{x} f_i(y)dy$ be the cumulative distribution function (CDF) of f_i. Hence, $\Pr(\omega^{(k)}) = \Pr((t_1 > t_2), \ldots, (t_{k-1} > t_k), (t_k > T(\omega^{(k)})))$, where

$$\Pr(t_k > T(\omega^{(k)})) = \int_{lo_k}^{up_k} f_k(x) \cdot (\prod_{t_i \in T(\omega^{(k)})} F_i(x))dx \tag{5.1}$$

Hence, we have

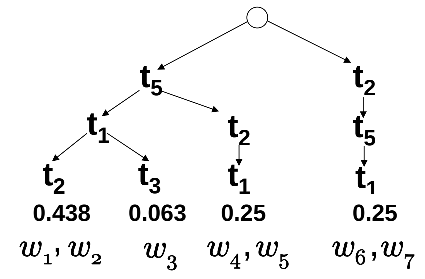


Figure 5.1: Prefixes of linear extensions at depth 3

$$\Pr(\omega^{(k)}) = \int_{lo_1}^{up_1} \int_{lo_2}^{x_1} ... \int_{lo_k}^{x_{k-1}} f_1(x_1)...f_k(x_k) \cdot (\prod_{t_i \in T(\omega^{(k)})} F_i(x_k)) \; dx_k \dots dx_1 \qquad (5.2)$$

Figure 5.1 shows the prefixes of length 3 and their probabilities for the linear extensions tree in Figure 3.6. We annotate the leaves with the linear extensions that share each prefix. Unfortunately, prefix enumeration is still infeasible for all but the smallest sets of elements, and, in addition, finding the probabilities of nodes in the prefix tree requires computing an l dimensional integral, where l is the node's level.

Algorithm BASELINE. The algorithm computes UTop-Prefix query by scanning the nodes in the prefix tree in depth-first search order, computing integrals only for the nodes at depth k (Equation 5.2), and reporting the prefixes with the highest probabilities. We can use these probabilities to answer UTop-Rank query for ranks $1 \dots k$, since the probability of a node t at level $l < k$ can be found by summing the probabilities of its children. Once the nodes in the tree have been labeled with their probabilities, the answer of UTop-Rank(i, j), $\forall i, j \in [1, k]$ and $i \leq j$, can be constructed by summing up the probabilities of all occurrences of a tuple t at levels $i \dots j$. This is easily done in time linear to the number of tree nodes using a breadth-first traversal of the tree. Here, we compute $\frac{m!}{(m-k)!}$ k-dimensional integrals to answer both queries. However, the algorithm still grows exponentially in m. Answering UTop-Set query can be done using the relationship among query answers discussed in Section 3.4.

Algorithm BASELINE exposes two fundamental challenges for efficient query evaluation:

1. Database size: The naïve algorithm is exponential in database size. How to make use of special indexes and other data structures to access a small proportion of database tuples while computing query answers?

2. Query evaluation cost: Computing probabilities by straightforward aggregation is prohibitively expensive. How to exploit query semantics for faster computation?

In Section 5.2, we answer the first question by using indexes to prune tuples that do not contribute to query answers, while in Sections 5.3, 5.4 and 5.5, we answer the second question by exploiting query semantics for faster computation.

5.2 k-Dominance: Shrinking the Database

We denote with $\mathcal{D}$, an uncertain database following the attribute level uncertainty model (cf. Section 3.2). We call a tuple $t \in \mathcal{D}$ "k-dominated" if at least k other tuples in $\mathcal{D}$ dominate t. For example in Figure 3.6, the tuples t_4 and t_6 are 3-dominated. Our main insight to shrink the database $\mathcal{D}$ used in query evaluation is based on Theorem 6.

Theorem 6 *Let $\mathcal{D} = \{t_1, \ldots, t_n\}$ be a set of tuples with uncertain scores. Let $\mathcal{T}$ be the set of tuples in $\mathcal{D}$, where each tuple in $\mathcal{T}$ is dominated by less than k tuples. Let $t_{(k)} \in \mathcal{D}$ be the tuple with the k^{th} largest score lower bound. Then, $\forall t_i \in \mathcal{D}$ we have:*

1. $(up_i > lo_{(k)}) \Rightarrow t_i \in \mathcal{T}$

2. $[(up_{(k)} = lo_{(k)} = up_i = lo_i) \wedge (\tau(t_i, t_{(k)}) = t_i)] \Rightarrow t_i \in \mathcal{T}$. □

Proof. If $(up_i > lo_{(k)})$, then the number of score intervals with lo scores $\geq up_i$ is less than k. Then, $t_i \in \mathcal{T}$, and hence (1) follows.

If t_i and $t_{(k)}$ have equal single-valued scores and the tie breaker τ favors t_i to $t_{(k)}$, then t_i must appear before $t_{(k)}$ in the order of $\mathcal{D}$ on lo scores. Hence, there are less than k tuples with lo scores $\geq up_i$, and hence (2) follows. $\square$

Based on Theorem 6, k-dominated tuples do not occupy ranks $\leq k$ in any linear extension, and so they do not affect the probability of any k-length prefix. Hence, k-dominated tuples can be safely pruned from $\mathcal{D}$.

In the following, we describe a simple and efficient technique to shrink the database $\mathcal{D}$ by removing all k-dominated tuples. Our technique assumes a list U ordering tuples in $\mathcal{D}$ in descending score upper-bound (up_i) order, and that $t_{(k)}$, the tuple with the k^{th} largest score lower-bound (lo_i), is known (e.g., by using an index maintained over score lower-bounds). Ties among tuples are resolved using our deterministic tie breaker τ (Section 3.2).

Algorithm 4 gives the details of our technique. The central idea is to conduct a binary search on U to find the tuple t^*, such that t^* is dominated by $t_{(k)}$, and t^* is located at the highest possible position in U. Based on Theorem 6, t^* is k-dominated. Moreover, let pos^* be the position of t^* in U, then all tuples located at positions $\geq pos^*$ in U are also k-dominated.

Complexity Analysis. Since Algorithm 4 conducts a binary search on U, its worst case complexity is in $O(log(m))$, where $m = |\mathcal{D}|$. The list U is given by sorting $\mathcal{D}$ on up_i in $O(m\ log(m))$, while $t_{(k)}$ is found in $O(m\ log(k))$ by scanning $\mathcal{D}$ while maintaining a k-length priority queue for the top-k tuples with respect to lo_i's. The overall complexity is thus $O(m\ log(m))$, which is the same complexity of sorting $\mathcal{D}$.

In the remainder of this chapter, we use $\acute{\mathcal{D}}$ to refer to the database $\mathcal{D}$ after removing all k-dominated tuples.

5.3 Overview of Query Processing

There are two main factors impacting query evaluation cost: the size of answer space, and the cost of answer computation.

Algorithm 4 Remove k-Dominated Tuples

```
SHRINK_DB ($\mathcal{D}$: database, $k$: dominance level, $U$: score upper-bound list)
 1  $start \leftarrow 1$; $end \leftarrow |\mathcal{D}|$
 2  $pos^* \leftarrow |\mathcal{D}| + 1$
 3  $t_{(k)} \leftarrow$ the tuple with the $k^{th}$ largest $lo_i$
 4  while ($start \leq end$) {binary search}
 5        do
 6            $mid \leftarrow \frac{start+end}{2}$
 7            $t_i \leftarrow$ tuple at position $mid$ in $U$
 8            if ($t_{(k)}$ dominates $t_i$)
 9              then
10                  $pos^* \leftarrow mid$
11                  $end \leftarrow mid - 1$
12              else  {$t_{(k)}$ does not dominate tuples above $t_i$}
13                  $start \leftarrow mid + 1$
14  return $\mathcal{D} \setminus$ {$t$: $t$ is located at position $\geq pos^*$ in $U$ }
```

The size of the answer space of a UTop-Rank query is bounded by $|\acute{\mathcal{D}}|$ (the number of tuples in $\acute{\mathcal{D}}$), while for UTop-Set and UTop-Prefix queries, it is exponential in $|\acute{\mathcal{D}}|$ (the number of tuple subsets of size k in $\acute{\mathcal{D}}$). Hence, materializing the answer space for UTop-Rank queries is feasible, while materializing the answer space of UTop-Set and UTop-Prefix queries is very expensive (in general, it is intractable).

The computation cost of each answer can be heavily reduced by replacing the straightforward probability aggregation algorithm (Section 5.1) with Monte-Carlo integration exploiting the query semantics to avoid enumerating the probability space.

Let $\acute{\mathcal{D}} = \{t_1, t_2, \ldots, t_n\}$, where $n = |\acute{\mathcal{D}}|$. Let Γ be the n-dimensional hypercube that consists of all possible combinations of tuples' scores. That is, $\Gamma = ([lo_1, up_1] \times [lo_2, up_2] \times \cdots \times [lo_n, up_n])$. A vector $\gamma = (x_1, x_2, \ldots, x_n)$ of n real values, where $x_i \in [lo_i, up_i]$, represents one point in Γ. Let $\Pi_{\acute{\mathcal{D}}}(\gamma) = \prod_{i=1}^{n} f_i(x_i)$, where f_i is the score density of tuple t_i. Tuples with deterministic (single-valued) scores are represented by the same score value in all possible γ's. On the other hand, tuples with uncertain scores can be represented by different score values in different γ's according to the intervals that enclose their possible scores.

In case of a continuous f_i, the component x_i is assumed to be a tiny score interval in $[lo_i, up_i]$, and $f_i(x_i)$ is the result of integrating f_i over x_i. We assume that the components x_i's of any possible vector $\gamma = (x_1, x_2, \ldots, x_n)$ can always be totally ordered based on their values.

We next discuss the details of our Monte-Carlo integration method.

5.4 Computing UTop-Rank Query

We start by defining tuple's rank interval, which is the range of possible ranks that can be occupied by some tuple.

Definition 11 [Rank Interval] *The rank interval of a tuple $t \in \acute{\mathcal{D}}$ is the range of all possible ranks of t in the linear extensions of the PPO induced by $\acute{\mathcal{D}}$.* □

For a tuple $t \in \acute{\mathcal{D}}$, let $\overline{\acute{\mathcal{D}}}(t) \subseteq \acute{\mathcal{D}}$ and $\underline{\acute{\mathcal{D}}}(t) \subseteq \acute{\mathcal{D}}$ be the tuple subsets dominating t and dominated by t, respectively. Then, based on the semantics of partial orders, the rank interval of t is given by $[|\overline{\acute{\mathcal{D}}}(t)| + 1, n - |\underline{\acute{\mathcal{D}}}(t)|]$.

For example, in Figure 3.6, for $\acute{\mathcal{D}} = \{t_1, t_2, t_3, t_5\}$, we have $\overline{\acute{\mathcal{D}}}(t_5) = \phi$, and $\underline{\acute{\mathcal{D}}}(t_5) = \{t_1, t_3\}$, and thus the rank interval of t_5 is $[1, 2]$.

The database shrinking algorithm in Section 5.2 does not affect tuple ranks smaller than k, since any k-dominated tuple appears only at ranks $> k$. Hence, given a range of ranks $i \ldots j$, we know that a tuple t has non-zero probability to be in the answer of UTop-Rank(i, j) query only if its rank interval intersects $[i, j]$.

We compute UTop-Rank(i, j) query using Monte-Carlo integration. The main insight is transforming the complex space of linear extensions, that have to be aggregated to compute query answer, to the simpler space of all possible score combinations Γ. Such space can be sampled uniformly and independently to find the probability of query answer without enumerating the linear extensions. The accuracy of the result depends only on the number of drawn samples s (cf. Section 2.3). We assume that the number of samples is large enough so that the approximation error can be ignored. We experimentally show in Section 5.7 that we obtain query answers with high accuracy using such strategy.

For a tuple t_k, we draw a sample $\gamma \in \Gamma$ as follows:

1. Generate the value x_k in γ
2. Generate $n-1$ independent values for other components in γ one by one.
3. If at any point there are j values in γ greater than x_k, reject γ.
4. Eventually, if the rank of x_k in γ is in $i \dots j$, accept γ.

Let $\lambda_{(i,j)}(t_k)$ be the probability of t_k to appear at rank $i \dots j$. The above procedure describes a sampler that can be used to evaluate the following integral using the Monte-Carlo integration method discussed in Section 2.3:

$$\lambda_{(i,j)}(t_k) = \int_{\Gamma_{(i,j,t_k)}} \Pi_{\acute{\mathcal{D}}}(\gamma)\ d\gamma \tag{5.3}$$

where $\Gamma_{(i,j,t_k)} \subseteq \Gamma$ is the volume defined by the points $\gamma = (x_1, \dots, x_n)$, with x_k's rank is in $i \dots j$.

Complexity Analysis. Let s be the total number of samples drawn from Γ to evaluate Equation 5.3. In order to compute the l most probable tuples to appear at a rank in $i \dots j$, we need to apply Equation 5.3 to each tuple in $\acute{\mathcal{D}}$ whose rank interval intersects $[i, j]$, and use a heap of size l to maintain the l most probable tuples. Hence, computing l-UTop-Rank(i, j) query has a complexity of $O(s \cdot n_{(i,j)} \cdot log(l))$, where $n_{(i,j)}$ is the number of tuples in $\acute{\mathcal{D}}$ whose rank intervals intersect $[i, j]$. In the worst case, $n_{(i,j)} = n$.

5.5 Computing UTop-Prefix and UTop-Set Queries

Let v be an ordered list of k tuples, and s be a set of k tuples. We denote with $\Pr(v)$ the top-k prefix probability of v and, similarly, we denote with $\Pr(s)$ the top-k set probability of s.

Similar to our discussion of UTop-Rank queries in Section 5.4, $\Pr(v)$ is computed using Monte-Carlo integration on the volume $\Gamma_{(v)} \subseteq \Gamma$ which consists of the points $\gamma = (x_1, \ldots, x_n)$ such that the values in γ that correspond to tuples in v have the same ranking as the ranking of tuples in v, and any other value in γ is smaller than the value corresponding to the last tuple in v. On the other hand, $\Pr(s)$ is computed by integrating on the volume $\Gamma_{(u)} \subseteq \Gamma$ which consists of the points $\gamma = (x_1, \ldots, x_n)$ such that any value in γ, that does not correspond to a tuple in s, is smaller than the minimum value that corresponds to a tuple in s.

The cost of the previous Monte-Carlo integration procedure can be further improved using the CDF product of remaining tuples in $\acute{\mathcal{D}}$, as described in Equation 5.2.

The cost of the above integrals is similar to the cost of the integral in Equation 5.3 (mainly proportional to the number of samples). However, the number of integrals we need to evaluate here is exponential (one integral per each top-k prefix/set), while it is linear for UTop-Rank queries (one integral per each tuple).

In the following, we describe a branch-and-bound search algorithm to compute exact query answers (Section 5.5.1). We also describe sampling techniques, based on the (M-H) algorithm (cf. Section 2.3), to compute approximate query answers at a lower computational cost (Section 5.5.2).

5.5.1 A Branch-and-Bound Algorithm

Our branch-and-bound algorithm employs a systematic approach to enumerate all possible candidate solutions (i.e., possible top-k prefixes/sets), while discarding a large subset of these solutions by upper-bounding the probability of unexplored candidates. We discuss our algorithm by describing how candidates are generated, and how candidate pruning is conducted. We conclude our discussion by giving the overall branch-and-bound algorithm. For clarity of presentation, we focus our discussion on the evaluation of UTop-Prefix query. We show how to extend the algorithm to evaluate UTop-Set query at the end of this section.

Candidate Generation. Based on our discussion in Section 5.4, the rank intervals of different tuples can be derived from the score dominance relationships in the underlying PPO. Using the rank intervals of different tuples, we can incrementally generate candidate top-k prefixes by selecting a distinct tuple $t_{(i)}$ for each rank $i = 1 \dots k$ such that the rank interval of $t_{(i)}$ encloses i, and the selected tuples at different ranks form together a valid top-k prefix (i.e., a prefix of at least one linear extension of the underlying PPO). A top-k prefix v is valid if for each tuple $t_{(i)} \in v$, all tuples dominating $t_{(i)}$ appear in v at ranks smaller than i. For example in Figure 3.6, the set of tuples that appear at ranks 1 and 2 are $\{t_5, t_2\}$ and $\{t_1, t_2, t_5\}$, respectively. The top-2 prefix $\langle t_2, t_1 \rangle$ is invalid since the tuple t_5, that dominates t_1, is not selected at rank 1. On the other hand, the top-2 prefix $\langle t_5, t_1 \rangle$ is valid since t_1 can be ranked after t_5.

Candidate Pruning. Pruning unexplored candidates is mainly done based on the following property (Property 4). We use subscripts to denote prefixes' lengths (e.g., v_x is a top-x prefix).

Property 4 [Prefix Inclusion] *Let v_x be a top-x prefix and v_y be a top-y prefix, where $v_x \subseteq v_y$. Then,* $\Pr(v_y) \leq \Pr(v_x)$. □

The correctness of Property 4 follows from the definition of our probability space: The set of linear extensions prefixed by v_x includes all linear extensions prefixed by v_y. Since the probability of a prefix v_l is the summation of all linear extensions prefixed by v_l, Property 4 follows.

Hence, given a top-k prefix v_k, any top-x prefix v_x with $x \leq k$ and $\Pr(v_x) < \Pr(v_k)$ can be safely pruned from the candidates set since $\Pr(v_x)$ upper-bounds the probability of any top-k prefix $\acute{v}_k$ where $v_x \subseteq \acute{v}_k$.

The Overall Search Algorithm. The details of the branch-and-bound search algorithm are given in Algorithm 5. The algorithm works in the two following phases:

- An *initialization phase* that builds and populates the data structures necessary for conducting the search.
- A *searching phase* that applies greedy search heuristics to lazily explore the answer space and prune all candidates that do not lead to query answers.

In the initialization phase, the algorithm reduces the size of the input database, based on the parameter k, by invoking the shrinking algorithm discussed in Section 5.2. The techniques described in Section 5.4 are then used to compute the distribution $\lambda_{(i,i)}$ for $i = 1 \dots k$. The algorithm maintains k lists $L_1 \dots L_k$ such that list L_i sorts tuples in $\lambda_{(i,i)}$ in a descending probability order.

In the searching phase, the algorithm maintains a priority queue $\mathcal{Q}$ that maintains generated candidates in descending order of probability. The priority queue is initialized with an empty prefix v_0 of length 0 and probability 1. Each maintained candidate v_x of length $x < k$ keeps a pointer $v_x.ptr$ pointing at the position of the next tuple in the list L_{x+1} to be used in extending v_x into a candidate of length $x+1$. Initially, $v_x.ptr$ is set to the first position in L_{x+1}. The positions are assumed to be 0-based. Hence, the value of $v_x.ptr$ ranges between 0 and $|L_{x+1}| - 1$.

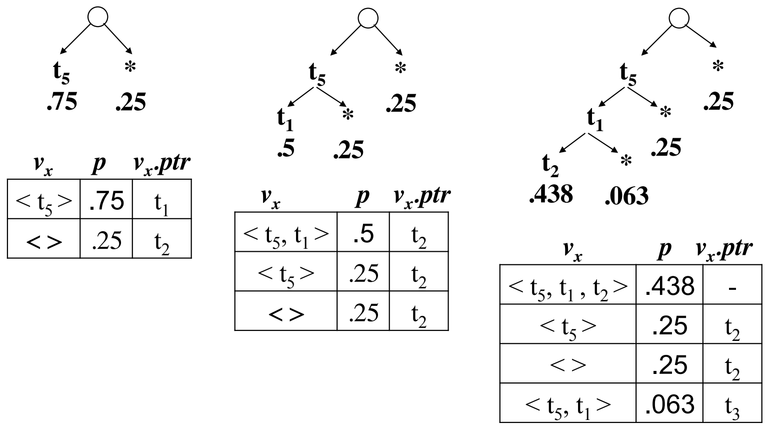
Extending candidates is done lazily (i.e., one candidate is extended at a time). Following the greedy criteria of A* search, the algorithm selects the next candidate to extend as follows. At each iteration, the algorithm evicts the candidate v_x^* at the top of $\mathcal{Q}$ (i.e., $\Pr(v_x^*)$ is the highest probability in $\mathcal{Q}$). If $x = k$, the algorithm reports v_x^* as the query answer. Otherwise, if $x < k$, a new candidate v_{x+1} is generated by augmenting v_x^* with the tuple t^* at the first position $\geq v_x^*.ptr$ in L_{x+1} such that

Algorithm 5 Branch-and-Bound UTop-Prefix Query Evaluation

```
BB-UTop-Prefix ($D$ : *database*, $k$ : *answer size*)
1  $U \leftarrow$ score upper-bound list {*starting Initialization Phase*}
2  $\acute{\mathcal{D}} \leftarrow$ Shrink_DB$(D, k, U)$ {*cf. Section 5.2*}
3  for $i = 1$ to $k$
4      Compute $\lambda_{(i,i)}$ based on $\acute{\mathcal{D}}$ {*cf. Section 5.4*}
5      $L_i \leftarrow$ sort tuples in $\lambda_{(i,i)}$ in a descending probability order
6  $\mathcal{Q} \leftarrow$ a priority queue of prefixes ordered on probability {*starting Searching Phase*}
7  $v_0 \leftarrow$ an empty prefix with probability 1 ; $v_0.ptr \leftarrow 0$ {*first position in* $L_1$}
8  Insert $v_0$ into $\mathcal{Q}$
9  while ($\mathcal{Q}$ is not empty)
10     $v_x^* \leftarrow$ evict top prefix in $\mathcal{Q}$
11     if $(x = k)$ then return $v_x^*$ {*reached query answer*}
12     $t^* \leftarrow$ first tuple in $L_{x+1}$ at position $pos^* \geq v_x^*.ptr$ s.t. $\langle v_x^*, t^* \rangle$ is a valid prefix
13     $v_x^*.ptr \leftarrow pos^* + 1$
14     $v_{x+1} \leftarrow \langle v_x^*, t^* \rangle$ ; Compute $\Pr(v_{x+1})$
15     if $(x + 1 = k)$
16     then
17         Prune all prefixes in $\mathcal{Q}$ with prob. $< \Pr(v_{x+1})$
18     else
19         $v_{x+1}.ptr \leftarrow 0$ {*first position in* $L_{x+2}$}
20     if $(v_x^*.ptr < |L_{x+1}|)$
21     then {*$v_x^*$ can be further extended*}
22         $\Pr(v_x^*) \leftarrow \Pr(v_x^*) - \Pr(v_{x+1})$
23         Insert $v_x^*$ into $\mathcal{Q}$
24     Insert $v_{x+1}$ into $\mathcal{Q}$
```

$v_{x+1} = \langle v_x^*, t^* \rangle$ is a valid prefix. The pointer $v_x^*.ptr$ is set to the position right after the position of t^* in L_{x+1}, while the pointer $v_{x+1}.ptr$ is set to the first position in L_{x+2} (only if $x + 1 < k$). The probabilities of v_{x+1} and v_x^* are computed ($\Pr(v_x^*)$ is reduced to $\Pr(v_x^*) - \Pr(v_{x+1})$) and the two candidates are reinserted in $\mathcal{Q}$. Furthermore, if $x + 1 = k$ (line 23), the algorithm prunes all candidates in $\mathcal{Q}$ whose probabilities are less than $\Pr(v_{x+1})$ according to Property 4. Further, if $v_x^*.ptr > |L_{x+1}|$, then v_x^* cannot be extended into candidates of larger length, and so v_x^* is removed from $\mathcal{Q}$.

Figure 5.2 gives an example illustrating how Algorithm 5 works. We use the PPO in Figure 3.6 in this example. Figure 5.2 shows how the branch-and-bound algorithm



v_x	p	$v_x.ptr$
$<t_5>$	.75	t_1
$<>$	.25	t_2

v_x	p	$v_x.ptr$
$<t_5, t_1>$	.5	t_2
$<t_5>$	.25	t_2
$<>$	.25	t_2

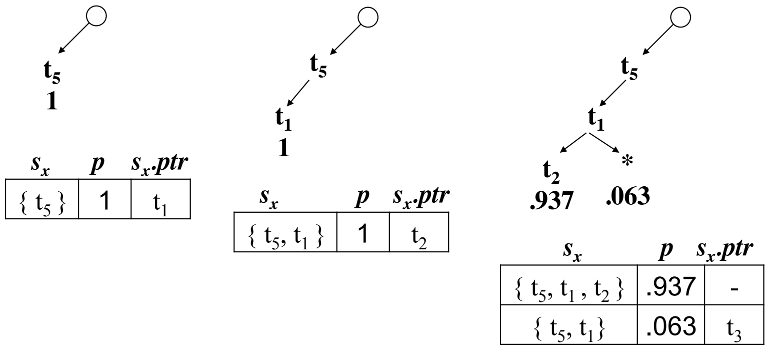
v_x	p	$v_x.ptr$
$<t_5, t_1, t_2>$	.438	-
$<t_5>$	.25	t_2
$<>$	.25	t_2
$<t_5, t_1>$	.063	t_3

Figure 5.2: Evaluating UTop-Prefix(3) query

computes for the answer of a UTop-Prefix(3) query, where the ordered tuples lists $L_1 = \langle t_5, t_2 \rangle$, $L_2 = \langle t_1, t_2, t_5 \rangle$, and $L_3 = \langle t_1, t_2, t_3 \rangle$. The search starts with an empty prefix v_0 with probability 1. The prefix v_0 is extended using t_5 (the first tuple in L_1). The algorithm then computes $\Pr(\langle t_5 \rangle)$ as .75 (the probability is computed using Monte-Carlo integration as discussed in the beginning of Section 5.5), while $\Pr(v_0)$ decreases by .75. Both prefixes are inserted into $\mathcal{Q}$ after updating their *ptr* fields to point to the next tuple that can be used to create valid prefixes later. After three steps, the search terminates since the top prefix in $\mathcal{Q}$ has length 3.

Computing UTop-Set Query by Branch-and-Bound. The Branch-and-Bound prefix search algorithm can be extended to compute UTop-Set queries. The reason is that Property 4 also holds on sets. That is, let s_x and s_y be two tuple sets with sizes x and y, respectively. Then, if $s_x \subseteq s_y$, we have $\Pr(s_y) \leq \Pr(s_x)$. Hence, $\Pr(s_y)$ upper-bounds the probability of any set that can be created by appending more tuples to s_y.

The main difference between prefix search and set search is that multiple prefixes map to the same set. For example, both prefixes $\langle t_2, t_5 \rangle$ and $\langle t_5, t_2 \rangle$ map to the set $\{t_2, t_5\}$. We thus need to filter out prefixes that map to already instantiated sets. This is done by maintaining an additional hash table of instantiated sets. Each generated candidate is first looked up in the hash table, and a new set is instantiated only if the



s_x	p	s_x.ptr
{ t5 }	1	t1

s_x	p	s_x.ptr
{ t5, t1 }	1	t2

s_x	p	s_x.ptr
{ t5, t1 , t2 }	.937	-
{ t5, t1}	.063	t3

Figure 5.3: Evaluating UTop-Set(3) query

hash table does contain a corresponding set.

Figure 5.3 shows how the branch-and-bound algorithm computes for the answer of a UTop-Set(3) query. The search starts by instantiating an empty set s_0 with probability 1. The set s_0 is extended using t_5 (the first tuple in L_1), which results in having $\Pr(\{t_5\}) = 1$ (i.e., t_5 appears in all linear extensions at ranks $1 \ldots 3$), and hence $\Pr(s_0)$ is set to 0, and can thus be removed from $\mathcal{Q}$. After three steps, the search terminates since the top set in $\mathcal{Q}$ has size 3.

5.5.2 A Sampling-based Algorithm

In this section we describe a sampling-based algorithm to compute approximate answers of UTop-Prefix and UTop-Set Queries. Based on our method of eliminating k-dominated tuples (cf. Section 5.2), the input to our sampling algorithms is a reduced set of tuples with non-zero probabilities to appear at the top k ranks in a random linear extension. The objective is to sample from the space of linear extensions to approximate the most probable answers of ranking queries.

Sampling Space. A state in our space is a linear extension ω of the PPO induced by $\acute{\mathcal{D}}$. Let θ and Θ be the probability distributions of top-k prefixes and top-k sets, respectively. Let $\pi(\omega)$ be the probability of the top-k prefix, or the top-k set in ω, depending on whether we simulate θ or Θ, respectively. The main intuition of our sample generator is to *propose* states with high probabilities in a light-weight fashion. This is done by shuffling the ranking of tuples in ω biased by the weights of pairwise rankings (Equation 3.1). This approach guarantees sampling valid linear extensions since ranks are shuffled only when tuples probabilistically dominate each other.

Given a state ω_i, a candidate state ω_{i+1} is generated as follows:

1. Generate a random number $z \in [1, k]$.
2. For $j = 1 \dots z$ do the following:
   (a) Randomly pick a rank r_j in ω_i. Let $t_{(r_j)}$ be the tuple at rank r_j in ω_i.
   (b) If $r_j \in [1, k]$, move $t_{(r_j)}$ downward in ω_i, otherwise move $t_{(r_j)}$ upward. This is done by swapping $t_{(r_j)}$ with lower tuples in ω_i if $r_j \in [1, k]$, or with upper tuples if $r_j \notin [1, k]$. Swaps are conducted one by one, where swapping tuples $t_{(r_j)}$ and $t_{(m)}$ is committed with probability $P_{(r_j,m)} = \Pr(t_{(r_j)} > t_{(m)})$ if $r_j > m$, or with probability $P_{(m,r_j)} = \Pr(t_{(m)} > t_{(r_j)})$ otherwise. Tuple swapping stops at the first uncommitted swap.

The (M-H) algorithm is proven to converge with arbitrary proposal distributions [31]. Our proposal distribution $q(\omega_{i+1}|\omega_i)$ is defined as follows. In the above sample generator, at each step j, assume that $t_{(r_j)}$ has moved to a rank $r < r_j$. Let $R_{(r_j,r)} = \{r_j - 1, r_j - 2, \dots, r\}$. Let $P_j = \prod_{m \in R_{(r_j,r)}} P_{(r_j,m)}$. Similarly, P_j can be defined for $r > r_j$. Then, the proposal distribution $q(\omega_{i+1}|\omega_i) = \prod_{j=1}^{z} P_j$, due to independence of steps. Based on the (M-H) algorithm, ω_{i+1} is accepted with probability $\alpha = min(\frac{\pi(\omega_{i+1}).q(\omega_i|\omega_{i+1})}{\pi(\omega_i).q(\omega_{i+1}|\omega_i)}, 1)$.

Computing Query Answers. The (M-H) sampler simulates the top-k prefixes/sets distribution using a Markov chain (a random walk) that visits states biased by probability. Gelman and Rubin [28] argued that it is not generally possible to use a single simulation to infer distribution characteristics. The main problem is that the initial state may trap the random walk for many iterations in some region in the target distribution. The problem is solved by taking dispersed starting states and running multiple iterative simulations that independently explore the underlying distribution.

We thus run multiple independent Markov chains, where each chain starts from an independently selected initial state, and each chain simulates the space independently of all other chains. The initial state of each chain is obtained by independently selecting a random score value from each score interval, and ranking the tuples based on the drawn scores, resulting in a valid linear extension.

A crucial point is determining whether the chains have mixed with the target distribution (i.e., whether the current status of the simulation closely approximates the target distribution). At mixing time, the Markov chains produce samples that closely follow the target distribution and hence can be used to infer distribution characteristics. In order to judge chains mixing, we used the Gelman-Rubin diagnostic [28], a widely-used statistic in evaluating the convergence of multiple independent Markov chains [18]. The statistic is based on the idea that if a model has converged, then the behavior of all chains simulating the same distribution should be the same. This is evaluated by comparing the within-chain distribution variance to the across-chains variance. As the chains mix with the target distribution, the value of the Gelman-Rubin statistic approaches 1.0.

At mixing time, which is determined by the value of convergence diagnostic, each chain approximates the distribution's mode as the most probable visited state (similar to simulated annealing). The l most probable visited states across all chains approximate the l-UTop-Prefix (or l-UTop-Set) query answers. Such approximation improves as the simulation runs for longer times. The question is, at any point during simulation, how far is the approximation from the exact query answer?

We derive an upper-bound on the probability of any possible top-k prefix/set as follows. The top-k prefix probability of a prefix $\langle t_{(1)}, \ldots, t_{(k)} \rangle$ is equal to the probability of the event $e = ((t_{(1)} \text{ ranked } 1^{st}) \wedge \cdots \wedge (t_{(k)} \text{ ranked } k^{th}))$. Let $\lambda_i(t)$ be the probability of tuple t to be at rank i. Based on the principles of probability theory, we have $\Pr(e) \leq \min_{i=1}^{k} \lambda_i(t_{(i)})$. Hence, the top-$k$ prefix probability of any k-length prefix cannot exceed $\min_{i=1}^{k}(\max_{j=1}^{n} \lambda_i(t_j))$. Similarly, Let $\lambda_{1,k}(t)$ be the probability of tuple t to be at rank $1 \ldots k$. It can be shown that the top-k set probability of any k-length set cannot exceed the k^{th} largest $\lambda_{1,k}(t)$ value. The values of $\lambda_i(t)$ and $\lambda_{1,k}(t)$ are computed as discussed in Section 5.4. The approximation error is given by the difference between the top-k prefix/set probability upper-bound and the probability of the most probable state visited during simulation.

We note that the previous approximation error can overestimate the actual error, and that chains mixing time varies based on the fluctuations in the target distribution. However, we show in Section 5.7 that, in practice, using multiple chains can closely approximate the true top-k states, and that the actual approximation error diminishes by increasing the number of chains.

Caching. Our sample generator mainly uses 2-dimensional integrals (Equation 3.1) to bias generating a sample by its probability. Such 2-dimensional integrals are shared among many states. Similarly, since we use multiple chains to simulate the same distribution from different starting points, some states can be repeatedly visited by different chains. Hence, we cache the computed $\Pr(t_i > t_j)$ values and state probabilities during simulation to be reused at a small cost.

Combining MCMC method with Branch-and-Bound Search. The MCMC method and branch-and-bound search are two alternative techniques for computing prefix/set-based probabilistic top-k queries. The need of both techniques is justified by the traditional tradeoff between efficiency and accuracy. The MCMC method is an efficient sampling technique that produces approximate query answers, while branch-and-bound is a more expensive A^* search that produces exact query answers. However, it is also possible to think of ways to combine the benefits of the two methods. In the following, we give a high level description of two possible combination possibilities:

- A quick MCMC simulation can be initially run to reach some state s^* with a high probability. Then, we can prune incomplete paths, with probabilities below $\Pr(s^*)$, in the search tree constructed by the branch-and-bound algorithm.

- Pruned paths in the branch-and-bound search tree are used to constrain the MCMC simulation. For example, if we know that the most probable prefix does not start with tuple t (because the tree branch starting with t is pruned), we constrain the MCMC simulation by rejecting all proposed states with t positioned at rank 1. We thus constrain the simulation to other parts of the space where more probable states may exist.

5.6 Computing URank-Agg Query

Rank aggregation is the problem of computing a consensus ranking for a set of candidates $\mathcal{C}$ using input rankings of $\mathcal{C}$ coming from different voters. The problem has immediate applications in Web meta-search engines [23].

While our work is mainly concerned with ranking under possible worlds semantics (Section 3.4), we note that a strong resemblance exists between ranking in possible worlds and the rank aggregation problem. To the best of our knowledge, we give the first identified correspondence between the two problems.

Measuring the distance between two rankings of the set of candidates $\mathcal{C}$ is central to rank aggregation. Given two rankings ω_i and ω_j, let $\omega_i(c)$ and $\omega_j(c)$ be the positions of a candidate $c \in \mathcal{C}$ in ω_i and ω_j, respectively. Two widely used measures of the distance between two rankings are the Spearman footrule distance and the Kendall tau distance.

The Spearman footrule distance is the summation, over all candidates, of the distance between the positions of the same candidate in the two lists, formally defined as follows:

$$\mathrm{F}(\omega_i, \omega_j) = \sum_{c \in \mathcal{C}} |\omega_i(c) - \omega_j(c)| \tag{5.4}$$

On the other hand, the Kendall tau distance is the number of pairwise disagreements in the relative order of candidates in the two lists, formally defined as follows:

$$\mathrm{K}(\omega_i, \omega_j) = |\{(c_a, c_b) : a < b,\ \omega_i(c_a) < \omega_i(c_b), \omega_j(c_a) > \omega_j(c_b)\}| \tag{5.5}$$

The optimal rank aggregation is the ranking with the minimum average distance to all input rankings. It is well known that optimal rank aggregation under Kendall tau distance (also known as Kemeny-optimal aggregation) is the only aggregation that satisfies the following intuitive properties [23, 41]:

- Neutrality: if two candidates switch their positions in all input rankings, then their positions must be switched in the aggregate ranking.
- Consistency: if the set of input rankings is split into two sets A and B, such that the aggregate rankings of both A and B prefer candidate c_1 to candidate c_2, then the overall aggregate ranking must also prefer c_1 to c_2.
- Extended Condorcet Criterion: for two candidate sets $\mathcal{C}_1$ and $\mathcal{C}_2$, if for every $c_i \in \mathcal{C}_1$ and $c_j \in \mathcal{C}_2$, the majority of input rankings prefer c_i to c_j, then the aggregate ranking must prefer $\mathcal{C}_1$ to $\mathcal{C}_2$.

Unfortunately, rank aggregation under Kendall tau distance is NP-Hard in general. The optimal aggregation under Spearman footrule distance is a 2-approximation of the Kendall tau aggregation [23, 41]. That is, if $\omega^{(K)}$ is the optimal Kendall tau aggregation of full lists $\omega_1, \ldots, \omega_m$, and $\omega^{(F)}$ is the optimal footrule aggregation, then $\sum_{i=1}^{m} \mathrm{K}(\omega^{(F)}, \omega_i) \leq 2 \cdot \sum_{i=1}^{m} \mathrm{K}(\omega^{(K)}, \omega_i)$.

In the following sections we discuss evaluating URank-Agg query, based on our probabilistic partial order model, under each of the Spearman footrule distance (Section 5.6.1) and the Kendall tau distance (Section 5.6.2).

5.6.1 URank-Agg with Footrule Distance

Optimal rank aggregation under footrule distance can be computed in polynomial time by the following algorithm [23]. Given a set of rankings $\omega_1 \ldots \omega_m$, the objective is to find the optimal ranking ω^* that minimizes $\frac{1}{m} \sum_{i=1}^{m} \mathrm{F}(\omega^*, \omega_i)$. The problem is modeled using a weighted bipartite graph G with two sets of nodes. The first set has a node for each candidate, while the second set has a node for each rank. Each candidate

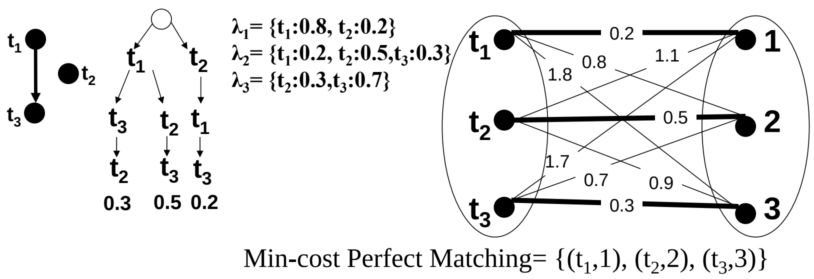


Figure 5.4: Bipartite graph matching

c and rank r are connected with an edge (c, r) whose weight $w(c, r) = \sum_{i=1}^{m} |\omega_i(c) - r|$. Then, ω^* (the optimal ranking) is given by "the minimum cost perfect matching" of G, where a perfect matching is a subset of graph edges such that every node is connected to exactly one edge, while the matching cost is the summation of the weights of its edges. Finding such matching can be done in $O(n^{2.5})$, where n is the number of graph nodes [23].

In our settings, viewing each linear extension as a voter gives us an instance of the rank aggregation problem on a huge number of voters. The objective is to find the optimal linear extension that has the minimum average distance to all linear extensions. We show that we can solve this problem in polynomial time, under footrule distance, given $\lambda_i(t)$ (the probability of tuple t to appear at each rank i, or, equivalently, the summation of the probabilities of all linear extensions having t at rank i).

Theorem 7 *For a* PPO$(\mathcal{R}, \mathcal{O}, \mathcal{P})$ *defined on n tuples, the optimal rank aggregation of the linear extensions, under footrule distance, can be solved in time polynomial in n using the distributions $\lambda_i(t)$ for $i = 1 \dots n$.* □

Proof. For each linear extension ω_i of PPO, assume that we duplicate ω_i a number of times proportional to $\Pr(\omega_i)$. Let $\acute{\Omega} = \{\acute{\omega_1}, \dots, \acute{\omega_m}\}$ be the set of all linear extensions' duplicates created in this way. Then, in the bipartite graph model, the edge connecting tuple t and rank r has a weight $w(t, r) = \sum_{i=1}^{|\acute{\Omega}|} |\acute{\omega_i}(t) - r|$, which is the same as $\sum_{j=1}^{n}(n_j \times |j - r|)$, where n_j is the number of linear extensions in $\acute{\Omega}$ having t at rank j. Dividing by $|\acute{\Omega}|$, we get $\frac{w(t,r)}{|\acute{\Omega}|} = \sum_{j=1}^{n}(\frac{n_j}{|\acute{\Omega}|} \times |j - r|) = \sum_{j=1}^{n}(\lambda_j(t) \times |j - r|)$. Hence, using $\lambda_i(t)$'s, we can compute $w(t,r)$ for every edge (t,r) divided by a fixed constant $|\acute{\Omega}|$, and thus the polynomial matching algorithm applies. $\square$

The intuition of Theorem 7 is that λ_i's provide compact summaries of voter's opinions, which allows us to efficiently compute the weights of graph edge without expanding the space of linear extensions. The distributions λ_i's are obtained by applying Equation 5.3 at each rank i separately, yielding a quadratic cost in the number of tuples n.

Figure 5.4 shows an example illustrating our technique. The probabilities of the depicted linear extensions are summarized as λ_i's without expanding the space (Section 5.4). The λ_i's are used to compute the weights in the bipartite graph yielding $\langle t_1, t_2, t_3\rangle$ as the optimal linear extension.

5.6.2 URank-Agg with Kendall Tau Distance

Optimal rank aggregation under Kendall tau distance is known to be NP-Hard in general by reduction to the problem of minimum feedback arc set [41]: Construct a complete weighted directed graph whose nodes are the candidates, such that an edge connecting nodes c_i and c_j is weighted by the proportion of voters who rank c_i before c_j. The problem is to find the set of edges with the minimum weight summation whose removal converts the input graph to a DAG. Since the input graph is complete, the resulting DAG defines a total order on the set of candidates, which is the optimal rank aggregation.

The hardness of the rank aggregation problem gives rise to approximation methods similar to the Markov chains-based methods in [23] to find the optimal rank aggregation. Spearman footrule aggregation is also known to be a 2-approximation of Kendall tau aggregation [41].

However, under our settings, we identify key properties that influence the hardness of computing optimal Kendall tau rank aggregation. We show that optimal rank aggregation can be computed in polynomial time depending on the properties of the underlying PPO, summarized as follows:

1. If the PPO is induced by tuples with non-uniform score densities, and the PPO is weak stochastic transitive (see Definition 12 below), then query computation cost is polynomial in n (the database size).

2. If the PPO is induced by tuples with uniform score densities, then the PPO is guaranteed to be weak stochastic transitive, and a polynomial time algorithm to compute Kendall tau aggregation exists. Moreover, by exploiting score uniformity, the complexity can be further reduced to $O(nlog(n))$.

We start our discussion by defining the property of Weak Stochastic Transitivity in the context of probabilistic partial orders.

Definition 12 [Weak Stochastic Transitivity] *A PPO induced by a database $\mathcal{D}$ is weak stochastic transitive iff* $\forall$ *tuples* $x, y, z \in D$: $[\Pr(x > y) \geq 0.5$ *and* $\Pr(y > z) \geq 0.5] \Rightarrow \Pr(x > z) \geq 0.5$. □

The property of weak stochastic transitivity is formulated and used in many probabilistic preference models. We refer the reader to [71] for a detailed discussion. We briefly contrast our interpretation of probabilistic preference against current interpretations in the following.

In many probabilistic preference models [71, 39, 25], for a pair of alternatives x and y, $\Pr(x > y)$ is interpreted as the probability that x is chosen over y. The origin of such probabilistic preferences can be related to changes in the internal state of the selecting agent (e.g., as a result of learning), to noise in the preferences obtained from users, or to the process of condensing users' votes into pairwise comparisons among candidates. In our settings, however, the origin of probabilistic preferences is the uncertainty in attribute values in the database, which in turn induces uncertainty in tuples' scores that we use for comparison and ranking. Our underlying probability space gives a concrete interpretation of $\Pr(x > y)$ as the summation of the probabilities of linear extensions (possible ranked instances of the database) where x is ranked above y.

Given an input PPO, the property of weak stochastic transitivity can be decided in $O(n^3)$, where n is the database size, since the property needs to be checked on tuple triples.

URank-Agg on a PPO with Non-uniform Score Densities

Let $\Omega = \{\omega_1 \dots \omega_m\}$ be the set of linear extensions of a PPO. The members of Ω represent voters associated with probabilistic weights. Hence, our objective is to find the optimal rank aggregation ω^* that minimizes $\frac{1}{m}\sum_{i=1}^{m} \Pr(\omega_i) \cdot \mathrm{K}(\omega^*, \omega_i)$.

Let $\Omega_{(t_i > t_j)} \subseteq \Omega$ be the set of linear extensions where t_i is ranked above t_j. Then, $\Pr(t_i > t_j) = \sum_{\omega \in \Omega_{(t_i > t_j)}} \Pr(\omega)$. Hence, ω^* is the ranking that minimizes the probability summation of pairwise preferences violating the order given by ω^*. That is, ω^* is the ranking that minimizes the following penalty function:

$$pen(\omega) = \sum_{t_i, t_j \in D : i<j, \omega(t_j) < \omega(t_i)} \Pr(t_i > t_j) \tag{5.6}$$

If the property of weak stochastic transitivity holds on the underlying PPO, then ω^* can be efficiently computed based on Theorem 8:

Theorem 8 *Given a weak stochastic transitive PPO induced by a database $\mathcal{D}$, the optimal rank aggregation ω^* under Kendall tau distance is defined as: $\forall$ tuples $x, y \in D$: $[\omega^*(x) < \omega^*(y)] \Leftrightarrow [\Pr(x > y) \geq 0.5]$ while breaking probability ties deterministically.* □

Proof. Since the underlying PPO is weak stochastic transitive, then ω^* is a valid ranking of $\mathcal{D}$, since the definition of ω^* does not introduce cycles in the relative order of tuples in $\mathcal{D}$.

Assume a rank aggregation $\acute{\omega}$ that is identical to ω^* except for the relative order of two tuples x and y. We consider the following three possible cases:

1. [$\Pr(x > y) = p > 0.5$] In this case we have $\omega^*(x) < \omega^*(y)$ while $\acute{\omega}(x) > \acute{\omega}(y)$. Hence, $pen(\omega^*) = pen(\acute{\omega}) - (2p - 1)$.
2. [$\Pr(x > y) = p < 0.5$] In this case we have $\omega^*(y) < \omega^*(x)$ while $\acute{\omega}(y) > \acute{\omega}(x)$. Hence, $pen(\omega^*) = pen(\acute{\omega}) - (1 - 2p)$.
3. [$\Pr(x > y) = p = 0.5$] In this case assume that the deterministic tie-breaker $\tau(x, y)$ states that $(x > y)$. Then, $\omega^*(x) < \omega^*(y)$ while $\acute{\omega}(x) > \acute{\omega}(y)$. Hence, $pen(\omega^*) = pen(\acute{\omega})$. The same result also holds if $\tau(x, y)$ states that $(y > x)$.

Moreover, for any other rank aggregation $\acute{\acute{\omega}}$ that is different from ω^* in the relative order of more than two tuples, we have $pen(\acute{\acute{\omega}}) \geq pen(\acute{\omega}) \geq pen(\omega^*)$. It follows that ω^* is the optimal rank aggregation. $\square$

Query Evaluation and Complexity Analysis. The result given by Theorem 8 allows for an efficient evaluation procedure to find the optimal rank aggregation in a weak stochastic transitive PPO. The procedure computes $\Pr(x > y)$ for each pair of tuples (x, y), and uses the computed probabilities to sort the database. That is, starting from an arbitrary ranking of tuples of $\mathcal{D}$, the positions of any two tuples x and y need to be swapped iff $\Pr(x > y) \geq 0.5$ and x is ranked below y. Based on the weak stochastic transitivity of the PPO, this procedure yields a valid ranking of $\mathcal{D}$ since transitivity does not introduce cycles in the relative order of tuples. Hence, the overall complexity of the query evaluation procedure is $O(n^2)$, where $n = |D|$, which is the complexity of computing $\Pr(x > y)$ on each pair of tuples (x, y). If it is not apriori known if the property of weak stochastic transitivity holds on the PPO, then the overall complexity becomes $O(n^3)$ since the PPO needs to be checked for being weak stochastic transitive first.

URank-Agg on a PPO with Uniform Score Densities

When tuples' uncertain scores have uniform densities, the cost of computing URank-Agg query drops considerably. We first prove in Theorem 9 below an important property that holds on the PPO induced by uniform score densities. In the following, we denote with $\mathrm{E}[f_i]$ the expected value of the score density f_i.

Theorem 9 *Given a PPO induced by tuples with uniform score densities in a database $\mathcal{D}$, then $\forall$ tuples $t_i, t_j \in D$: $(\mathrm{E}[f_i] \geq \mathrm{E}[f_j]) \Leftrightarrow (\Pr(t_i > t_j) \geq 0.5)$.* $\square$

Proof. First, we prove that $(\mathrm{E}[f_i] \geq \mathrm{E}[f_j]) \Rightarrow (\Pr(t_i > t_j) \geq 0.5)$. We first compute the integral that defines $\Pr(t_i > t_j)$ as follows. $\Pr(t_i > t_j) = \frac{1}{(up_i - lo_i) \times (up_j - lo_j)} \times \int_{lo_i}^{up_i} \int_{lo_j}^{x} dydx$. By solving the integral we get $\Pr(t_i > t_j) = \frac{1}{up_j - lo_j} \times (\frac{up_i + lo_i}{2} - lo_j) = \frac{1}{up_j - lo_j} \times (\mathrm{E}(f_i) - lo_j)$. We rewrite the given $(\mathrm{E}[f_i] \geq \mathrm{E}[f_j])$ as $\mathrm{E}[f_i] = \mathrm{E}[f_j] + \epsilon$, where $\epsilon \geq 0$. Then, $\Pr(t_i > t_j) = \frac{1}{up_j - lo_j} \times ((\mathrm{E}(f_j) - lo_j) + \epsilon) = \frac{1}{2} + \frac{\epsilon}{up_j - lo_j}$, which means that $\Pr(t_i > t_j) \geq 0.5$.

Second, we prove that $(\Pr(t_i > t_j) \geq 0.5) \Rightarrow (\mathrm{E}[f_i] \geq \mathrm{E}[f_j])$. Assume that $\mathrm{E}[f_i] - \mathrm{E}[f_j] = \epsilon$, where ϵ is an arbitrary (positive/negative) real number. Since we have $\Pr(t_i > t_j) = \frac{1}{2} + \frac{\epsilon}{up_j - lo_j}$, and based on the given $(\Pr(t_i > t_j) \geq 0.5)$, we get $\frac{1}{2} + \frac{\epsilon}{up_j - lo_j} \geq \frac{1}{2}$, which means that $\epsilon \geq 0$. It follows that $\mathrm{E}[f_i] \geq \mathrm{E}[f_j]$, which concludes the proof. $\square$

Based on Theorem 9, for tuples $t_i, t_j, t_k \in D$, if $\Pr(t_i > t_j) \geq 0.5$ and $\Pr(t_j > t_k) \geq 0.5$, then we have $\mathrm{E}[f_i] \geq \mathrm{E}[f_j]$ and $\mathrm{E}[f_j] \geq \mathrm{E}[f_k]$. It follows that $\mathrm{E}[f_i] \geq \mathrm{E}[f_k]$, which also means that $\Pr(t_i > t_k) \geq 0.5$. Hence, a PPO that is induced by uniform score densities is weak stochastic transitive.

Query Evaluation and Complexity Analysis. Since the PPO is weak stochastic transitive, we do not need to conduct the transitivity checking step. We can compute URank-Agg query using the polynomial algorithm we described previously for weak stochastic transitive PPO's. However, based on Theorem 9, we can further optimize the computation cost. Specifically, for any two tuples t_i and t_j, we have $(\mathrm{E}[f_i] \geq \mathrm{E}[f_j]) \Leftrightarrow (\Pr(t_i > t_j) \geq 0.5)$. Hence, we can avoid computing $\Pr(t_i > t_j)$ for all tuple pairs (t_i, t_j), and sort the database based on the expected tuples' scores, which results in the same sorting based on $\Pr(t_i > t_j)$ values. Computing $\mathrm{E}[f_i]$ for every tuple t_i requires a linear scan over $\mathcal{D}$, which has a complexity of $O(n)$, while the subsequent sorting step has a complexity of $O(nlog(n))$. It follows that the query evaluation procedure has an overall complexity of $O(nlog(n))$.

5.7 Experiments

All experiments are conducted on a machine with two Dual Core 2.2GHz processors, 2GB of RAM, and 80GB of disk space. We used both real and synthetic data to evaluate our methods under different configurations. We experiment with two real datasets: (1) Apts: 33,000 apartment listings obtained by scraping the search results of `apartments.com`, and (2) Cars: 10,000 car ads scraped from `carpages.ca`. The *rent* attribute in Apts is used as the scoring function (65% of scraped apartment listings have uncertain rent values), and similarly, the *price* attribute in Cars is used as the scoring function (10% of scraped car ads have uncertain price).

The synthetic data sets have different distributions of score intervals' bounds: (1) Syn-u-p: bounds are uniformly distributed, (2) Syn-g-p: bounds are drawn from Gaussian distribution, and (3) Syn-e-p: bounds are drawn from exponential distribution. The parameter p represents the proportion of tuples with uncertain scores in each dataset is (default is 0.5). The size of each dataset is 100,000 tuples. In all experiments, unless otherwise is specified, the score densities (f_i's) are taken as uniform.

For synthetic data, the bounds of the score interval of each tuple t_i is generated by drawing a random interval starting point lo_i from the dataset corresponding distribution (uniform, Gaussian($\mu = 0.5, \sigma = 0.05$), or exponential($\mu = 0.1$)) defined on the score range [0,1]. The width of the interval is uniform in [0,1]. The main intuition is to create different patterns of filling the score range with uncertain scores of different tuples. For example, while uniform distribution distributes the uncertain scores uniformly over the score range, exponential distribution creates a skewed pattern in which a few tuples have high scores, while the majority of tuples have low scores.

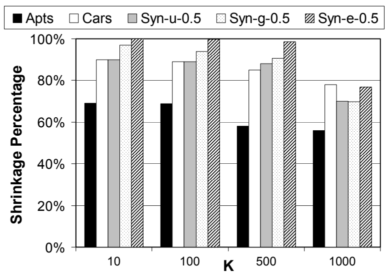


Figure 5.5: Reduction in database size

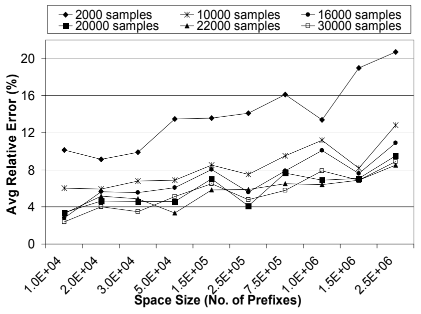


Figure 5.6: Accuracy of Monte-Carlo integration

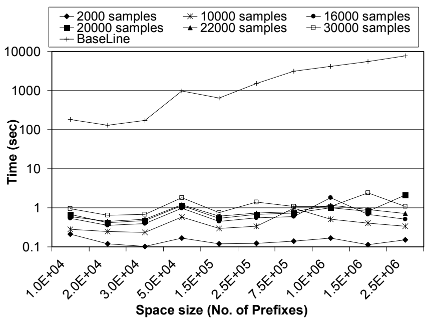


Figure 5.7: Comparison with BASELINE

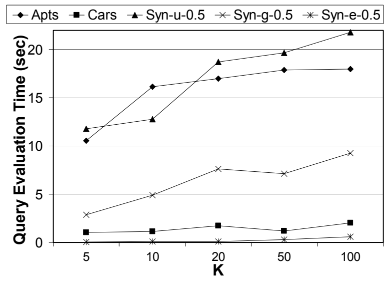


Figure 5.8: UTop-Rank Query evaluation time

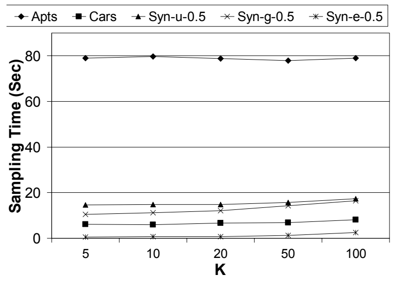


Figure 5.9: UTop-Rank Sampling time (10,000 samples)

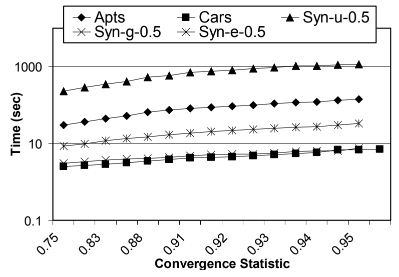


Figure 5.10: Chains convergence

Shrinking Database by k-Dominance

We evaluate the performance of the database shrinking algorithm (Algorithm 4). Figure 5.5 shows the database size reduction due to k-dominance with different k values. The maximum reduction, around 98%, is obtained with the Syn-e-0.5 dataset. The reason is that the skewed distribution of score bounds results in a few tuples dominating the majority of other database tuples.

We also evaluate the number of tuple accesses used to find the pruning position pos^* in the list U (Section 5.2). The logarithmic complexity of the algorithm guarantees a small number of tuple accesses of under 20 accesses in all datasets. The time consumed to construct the list U is under 1 second, while the time consumed by Algorithm 4 is under 0.2 second, in all datasets.

Accuracy and Efficiency of Monte-Carlo Integration

We evaluate the accuracy and efficiency of Monte-Carlo integration in computing UTop-Rank queries. The probabilities computed by the BASELINE algorithm are taken as the ground truth in accuracy evaluation. For each rank $i = 1 \ldots 10$, we compute the relative difference between the probability of tuple t to be at rank i, computed as in Section 5.4, and the same probability as computed by the BASELINE algorithm. We average this relative error across all tuples, and then across all ranks to get the total average error. Figure 5.6 shows the relative error with different space sizes (different number of linear extensions' prefixes processed by BASELINE). The different space sizes are obtained by experimenting with different subsets from the Apts dataset. The relative error is more sensitive to the number of samples than to the space size. For example, increasing the number of samples from 2,000 to 30,000 diminishes the relative error by almost half, while for the same sample size, the relative error only doubled when the space size increased by 100 times.

Figure 5.7 compares (in log-scale) the efficiency of Monte-Carlo integration against the BASELINE algorithm. While the time consumed by Monte-Carlo integration is fixed with the same number of samples regardless the space size, the time consumed by the BASELINE algorithm increases exponentially when increasing the space size. For example, for a space of 2.5 million prefixes, Monte-Carlo integration consumes only 0.025% of the time consumed by the BASELINE algorithm.

Scalability with respect to k

We evaluate the efficiency of our query evaluation for UTop-Rank$(1, k)$ queries with different k values. Figure 5.8 shows the query evaluation time, based on 10,000 samples. On the average, query evaluation time doubled when k increased by 20 times. Figure 5.9 shows the time consumed in drawing the samples.

The difference in sampling and ranking times for different datasets is attributed to two main factors:

- The variance in the reduced sizes of the datasets based on the k-dominance criterion. For example, the majority of tuples in Syn-e-0.5 dataset are pruned using k-dominance, while a much smaller number of tuples are pruned in Syn-u-0.5 dataset. This happens due to the different distributions of the bounds of score intervals. In general, the dataset size is inversely proportional to processing time.

- The percentage of tuples with uncertain scores. For example, the percentage of tuples with uncertain scores in Apts is 65%, while it is only 10% in Cars. tuples with uncertain score results in longer processing times since space size (number of possible rankings) increases with score uncertainty.

Markov Chains Convergence

We evaluate the Markov chains mixing time (Section 5.5). For 10 chains and $k = 10$, Figure 5.10 illustrates the Markov chains convergence based on the value of Gelman-Rubin statistic as time increases. While convergence consumes less than one minute in all real datasets, and most of the synthetic datasets, the convergence is notably slower for the Syn-u-0.5 dataset. The interpretation is that the uniform distribution of the score intervals in Syn-u-0.5 increases the size of the prefixes space, and hence the Markov chains consume more time to cover the space and mix with the target distribution. In real datasets, however, we note that the score intervals are mostly clustered, since many tuples have similar or the same attribute values. Hence, such delay in covering the space does not occur.

Markov Chains Accuracy

We evaluate the ability of Markov chains to discover states whose probabilities are close to the most probable states. We compare the most probable states discovered by the Markov chains to the true envelop of the target distribution (taken as the 30 most probable states). After mixing, the chains produce representative samples from the space, and hence states with high probabilities are frequently reached. This

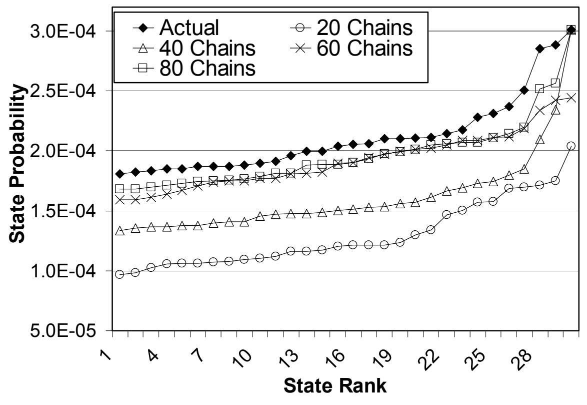


Figure 5.11: Space coverage

behavior is illustrated by Figure 5.11 for UTop-Prefix(5) query on a space of 2.5 million prefixes drawn from the Apts dataset. We compare the probabilities of the actual 30 most probable states and the 30 most probable states discovered by a number of independent chains after convergence, where the number of chains ranges from 20 to 80 chains.

The relative difference between the actual distribution envelop and the envelop induced by the chains decreases as the number of chains increase. The relative difference goes from 39% with 20 chains to 7% with 80 chains. The largest number of drawn samples is 70,000 (around 3% of the space size), and is produced using 80 chains. The convergence time increased from 10 seconds to 400 seconds when the number of chains increased from 20 to 80.

Branch-and-Bound Search

In this experiment, we evaluate the Branch-and-Bound techniques we propose in Section 5.5.1 to evaluate UTop-Prefix and UTop-Set queries. Figures 5.12 and 5.13 compare the processing time of Branch-and-Bound prefix search (Algorithm 5) and the MCMC sampling method (using 5 chains) for the datasets Syn-u-0.5 and Syn-g-0.5, respectively. The Branch-and-Bound search shows smaller running times with

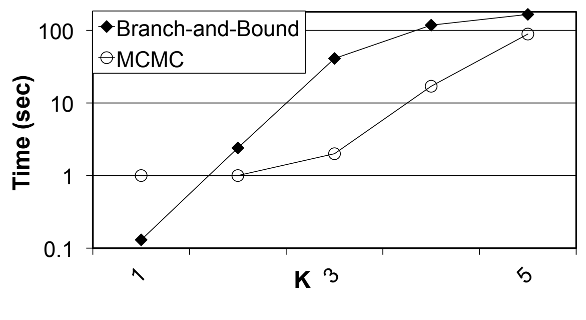


Figure 5.12: Evaluation time (Syn-u-0.5, UTop-Prefix)

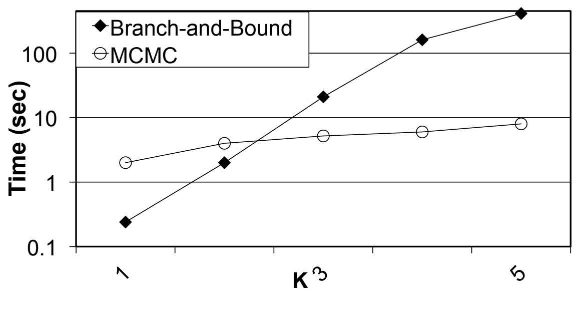


Figure 5.13: Evaluation time (Syn-g-0.5, UTop-Prefix)

small k values, as it does not have the overhead of proposing states as in the MCMC method. As the value of k increases, the number of materialized candidates by the Branch-and-Bound search increases, which negatively impacts the running times.

The MCMC method is, on the average, one order of magnitude faster than the Branch-and-Bound search. The savings in processing time in MCMC method comes with the price of giving approximate answers. The average absolute error in the probability of the answer reported by the MCMC method, with respect to the Branch-and-Bound exact search, is 0.0012 and 0.0007 for Syn-u-0.5 and Syn-g-0.5, respectively. The error decreases as the number of MCMC chains increases as we show in Section 5.7. Figures 5.14 and 5.15 show similar result for Apts dataset.

We next evaluate the effectiveness of the greedy criteria adopted by Branch-and-Bound search. Figures 5.16 and 5.17 compare the processing times of Branch-and-Bound search against the Baseline algorithm using Apts dataset for UTop-Prefix and UTop-Set queries, respectively. The Baseline algorithm shows an exponential increase in running time as space size (number of prefixes) increases (we omit running times that are significantly large). On the other hand, Branch-and-Bound search locates query answer in times below 30 seconds for both query types. Figure 5.18 compares the memory requirements (computed as the number of materialized candidates) of Branch-and-Bound and Baseline algorithms. The Baseline algorithm has, on the average, 3 orders of magnitude larger number of materialized candidates,

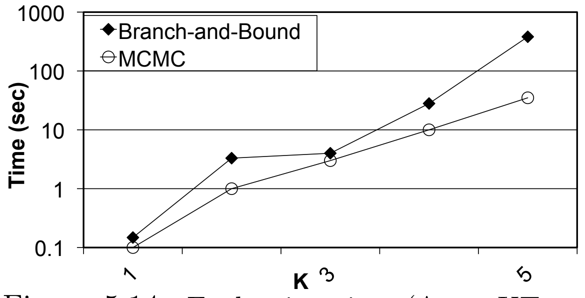


Figure 5.14: Evaluation time (Apts, UTop-Prefix)

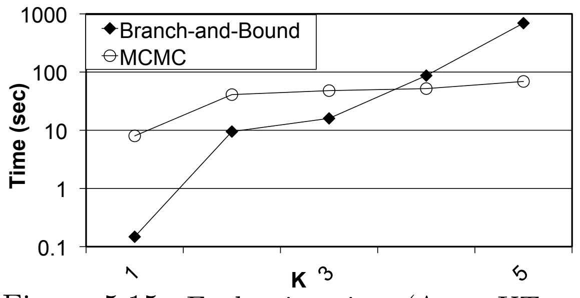


Figure 5.15: Evaluation time (Apts, UTop-Set)

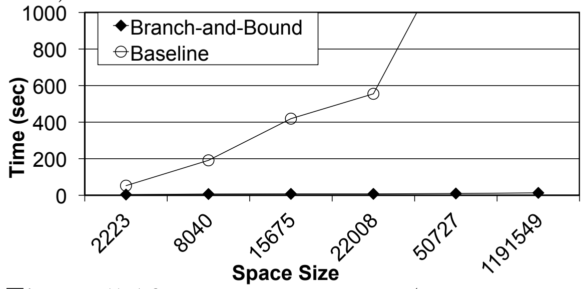


Figure 5.16: Evaluation time (Apts, UTop-Prefix)

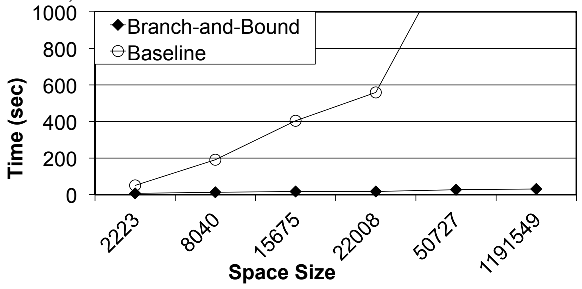


Figure 5.17: Evaluation time (Apts, UTop-Set)

which illustrates the effectiveness of the pruning techniques adopted by Branch-and-Bound search.

Score Uncertainty

In this experiment, we evaluate the effect of score uncertainty on algorithms performance. Figures 5.19 and 5.20 show the effect of the parameter p (the proportion of tuples with uncertain scores) on the running times of MCMC and Branch-and-Bound search in different datasets. Increasing p results in linear increase in the running times of both algorithms. On the average, as p doubled by 3.5 times, the running time of the MCMC method doubled by 5 times, while the running time of the Branch-and-Bound search doubled by 2.5 times.

We next evaluate the effect of the width of score interval on algorithms performance. We create synthetic data with different score interval width, where the interval width is represented as a percentage of the whole score range. As the score interval width increases, the number of tuples with incomparable scores increases. This results

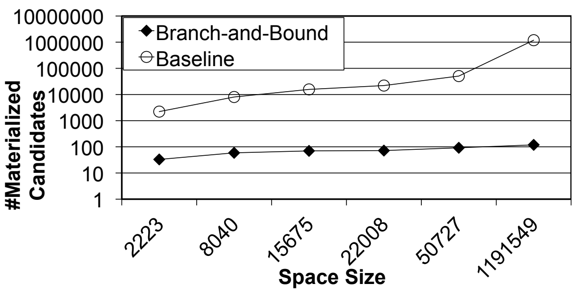


Figure 5.18: Consumed memory (Apts, UTop-Prefix)

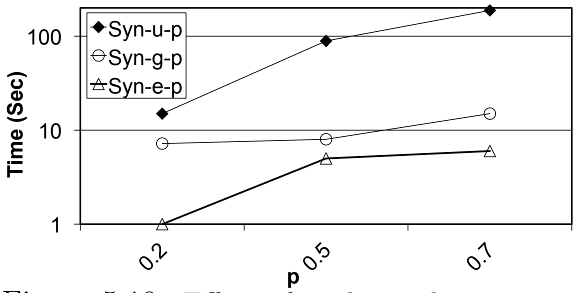


Figure 5.19: Effect of tuples with uncertain scores (MCMC,UTop-Prefix(5))

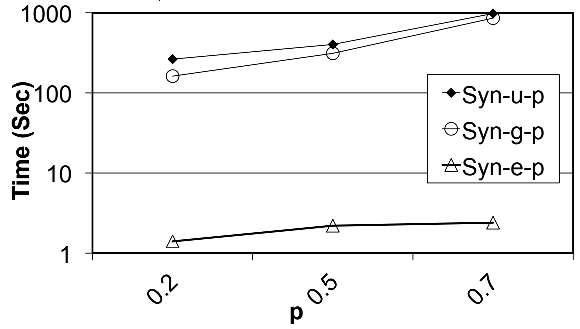


Figure 5.20: Effect of tuples with uncertain scores (Branch-and-Bound,UTop-Prefix(5))

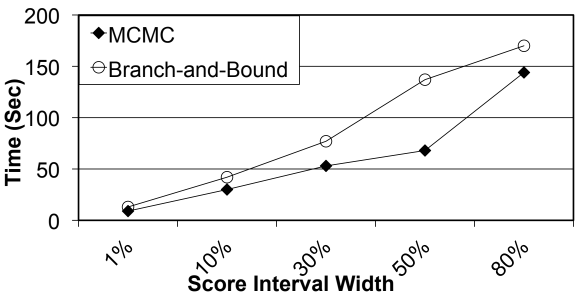


Figure 5.21: Effect of score interval width (UTop-Prefix(5))

in limiting the effect of pruning by score dominance, and hence increasing the overall running times. Figure 5.21 shows linear increase in the running times of MCMC and Branch-and-Bound search as the score interval width increases.

5.8 Summary and Lessons Learned

We build on the probabilistic model in Section 3.2 that extends partial orders to represent the uncertainty in the scores of database tuples. The model encapsulates a probability distribution on all possible rankings of database tuples. In this chapter, we present evaluation techniques for new formulations of several types of ranking queries under such model.

We design novel branch-and-bound and sampling-based query processing techniques to compute query answers. We also give polynomial time algorithms to solve the rank aggregation problem in the context of probabilistic partial orders. Our experimental study on both real and synthetic datasets demonstrates the scalability and accuracy of our techniques.

Lessons Learned. Based on the study and the experiments presented in this chapter, we make the following high level observations:

- For top-k queries with uncertain scores, exploiting k-dominance is a key optimization for efficient processing. Our experiments show that, on the average, the size of the database to be considered for top-k processing can be reduced by around 90% when exploiting k-dominance optimization. The main reason is that for the typically small values of k, many tuples can be pruned based on their score ranges before applying any probabilistic ranking techniques.
- The MCMC and branch-and-bound methods are alternative algorithms for computing prefix/set-based top-k queries. The need of both techniques is justified by the traditional tradeoff between efficiency and accuracy. The MCMC method provides a faster alternative whose cost can be controlled by selecting the sampling budget and the number of chains that are used to simulate the top-k prefix/set distribution. However, the answer reported by the MCMC method is only approximate, and we provide an upper bound of the involved approximation error which can act as a conservative stopping criterion of the MCMC simulation. On the other hand, the branch-and-bound method is an exact method that gives the most probable answer in the space by lazily generating the search space, and upper-bounding the probabilities of unexplored search path.
- The cost of the branch-and-bound method is usually higher than the cost of the MCMC method with typical parameter selection. We note that by increasing k, the required prefix/set size, the performance of the branch-and-bound method is negatively impacted in a more tangible way compared to the MCMC method. The main reason is that increasing k contributes linearly to the cost of sample generation in the MCMC method, while it adds more levels to the search tree of

the branch-and-bound method, leading to an exponential growth in the search space. This observation favors the MCMC method for relatively large k values, at the cost of obtaining only approximate answers.

Chapter 6

Uncertain Rank Join

In this chapter we discuss the methods we propose in [67] to handle uncertain scores when computing rank join queries. We present in Section 6.1 a new formulation for rank join queries on uncertain data. We give in Section 6.2 an uncertain rank join algorithm, and show how it can be implemented as a pipelined query operator. We discuss integrating join operation with probabilistic ranking in Section 6.3. Then, we discuss in Section 6.4 the technical details of MashRank prototype, where we apply our proposed techniques in Web mashup scenarios. We give our experimental study in Section 6.5, and summarize the contents of this chapter in Section 6.6.

6.1 Uncertain Rank Join Problem

Under our proposed score uncertainty model (Section 3.2), we present a formulation for top-k join queries on uncertain data.

We assume a process that joins a number of input relations based on a given deterministic join condition (e.g., join hotel and restaurant relations based on the traveling distance being smaller than 1 mile). That is, given a set of input tuples (one from each input relation), the join condition evaluates to either true or false. The result is a set of output tuples representing all the join results (e.g., hotel-restaurant joined tuples). For each join result, a score is computed based on a user-defined scoring function $\mathcal{F}$ (e.g., hotel.stars+restaurant.rating). The scoring function is assumed to be a monotone function (i.e., $\mathcal{F}(x_1, \ldots, x_n) \geq \mathcal{F}(\acute{x}_1, \ldots, \acute{x}_n)$ whenever $x_i \geq \acute{x}_i$ for every i). Typical scoring functions, such as summation, multiplication, min, max, and average, are monotone functions. We are interested in obtaining the k join results that maximize the value of $\mathcal{F}$.

When the scoring function $\mathcal{F}$ is defined on deterministic attributes, the top-k join results are obtained by sorting the join results on $\mathcal{F}$, and returning the top-k results. Many top-k join techniques [35, 51] address the interaction between computing the join results and producing the top-k answers. The main insight of these methods is exploiting pre-sorted input relations as well as the scoring function monotonicity to avoid complete sorting of the join results before producing the top-k joins.

We consider a different variant of the problem, where the scoring function $\mathcal{F}$ is defined on uncertain attributes, and hence there is a space of possible rankings of the join results. The objective is to integrate the join operation with the computation of a ranking of join results under our proposed ranking semantics in Chapter 3. We start by formulating the uncertain rank join problem under the score uncertainty model described in Section 3.2.

Definition 13 [Uncertain Rank Join (URankJoin)] *Let $\mathcal{R}$ be a set of relations $\{R_1, \ldots, R_m\}$, $\mathcal{F} : R_1 \bowtie \ldots \bowtie R_m \rightarrow \mathcal{I}$ be a monotone scoring function, where $\mathcal{I}$ is the domain of all possible sub-intervals of [0,1], and k be an integer $\leq |R_1 \bowtie \ldots \bowtie R_m|$. The query* URankJoin$(\mathcal{R}, \mathcal{F}, k)$ *computes a total order ω^* under some probabilistic ranking semantics (described below) of tuples in the set $\mathcal{J}_k \subseteq R_1 \bowtie \ldots \bowtie R_m$, where $|\mathcal{J}_k| \geq k$, and*
$\forall t_i \in \mathcal{J}_k$: $|\{t_j \in R_1 \bowtie \ldots \bowtie R_m : t_j > t_i\}| < k$, and
$\forall t_j \notin \mathcal{J}_k$: $|\{t_i \in \mathcal{J}_k : t_i > t_j\}| \geq k$. □

That is, URankJoin returns an ordering of tuples that have less than k other dominating tuples. To illustrate, consider Figure 6.1. URankJoin$(\{R, S\}, (R.a_1 +$

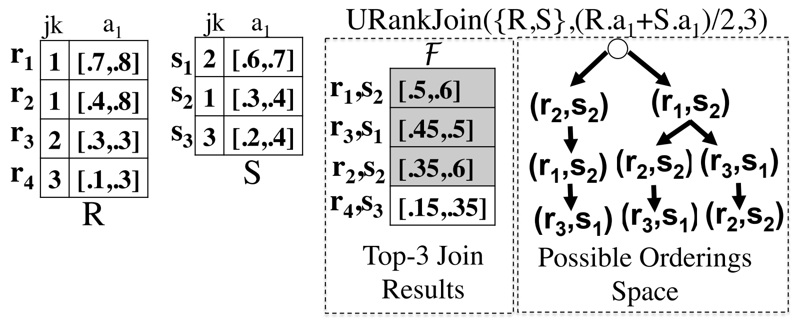


Figure 6.1: URANKJOIN example

$S.a_1)/2, 3)$, where the join condition is equality of attribute 'jk', returns a total order of join tuples in $\mathcal{J}_3 = \{(r_1, s_2), (r_3, s_1), (r_2, s_2)\}$, since all join tuples in $\mathcal{J}_3$ are dominated by less than 3 join tuples, and all join tuples not in $\mathcal{J}_3$ (only (r_4, s_3) in this example) are dominated by at least 3 tuples. Based on the monotonicity of $\mathcal{F}$, the *lo* and *up* scores of join tuples are given by applying $\mathcal{F}$ to the *lo* and *up* scores of the corresponding base tuples. For example, the score of (r_1, s_2) is given by $[\mathcal{F}(.7, .3), \mathcal{F}(.8, .4)] = [\frac{.7+.3}{2}, \frac{.8+.4}{2}] = [.5, .6]$.

Computing $\mathcal{J}_k$ does not require knowledge of f_i's (the score densities of individual tuples) of base or join tuples, since $\mathcal{J}_k$ is based on score dominance only. However, computing ω^* requires knowledge of f_i's.

We view computing ω^* as a sophisticated tie-breaking rule that maps tuples with overlapping score distributions to a total order. Total order is a widely accepted means for presenting a ranking (e.g., on the Web, results are usually totally ordered based on relevance to user's query). We thus assume total order as an easier to grasp presentation of results. Nevertheless, computing a total order is an added feature of the techniques we propose, since we can stop at computing $\mathcal{J}_k$ if results incomparability is not a concern.

Let Ω be the set of all possible orderings of tuples in $\mathcal{J}_k$, and $\omega[t]$ be the rank of t in an ordering $\omega \in \Omega$. An intuitive requirement in the total order ω^* is that it complies with score dominance (i.e., $(t_i > t_j) \Rightarrow (\omega^*[t_i] < \omega^*[t_j])$). To illustrate, consider

ID	Score	E[t]	ER[t]
t_1	[.6 , .6]	.6	2.5
t_2	[.4 , .8]	.6	2.316
t_3	[.3 , .5]	.4	3.981
t_4	[.2 , .35]	.275	4.965
t_5	[.7 , .7]	.7	1.251
t_6	[.1 , .1]	.1	6

R

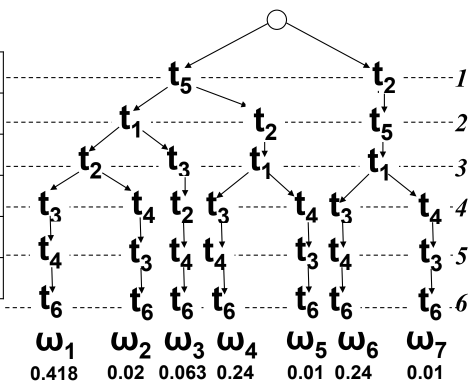


Figure 6.2: Space of possible orderings for tuples with uniform scores

Figure 6.2 which shows a relation R with uniform uncertain scores. The relation R has 7 possible orderings $\{\omega_1, \ldots, \omega_7\}$. Similar to our discussion in Section 3.4, multiple query semantics can be adopted to order tuples in $\mathcal{J}_k$ in a way complying with score dominance. We list some examples of these semantics in the following:

(1) Expected Scores. Let $\text{E}[t_i] = \int_{lo_i}^{up_i} x \cdot f_i(x)dx$. Then, $\omega^*[t_i] = 1 + |\{t_j : \text{E}[t_j] > \text{E}[t_i]\}|$, while resolving ties deterministically. For example in Figure 6.2, based on expected scores, $\omega^* = \langle t_5, t_1, t_2, t_3, t_4, t_6 \rangle$, assuming that the tie between t_1 and t_2 is resolved in favor of t_1.

(2) Expected Ranks. Let $\text{ER}[t_i] = \sum_{\omega \in \Omega} \omega[t_i] \cdot \Pr(\omega)$. Then, $\omega^*[t_i] = 1 + |\{t_j : \text{ER}[t_j] < \text{ER}[t_i]\}|$, while resolving ties deterministically. In Figure 6.2, based on expected ranks, $\omega^* = \langle t_5, t_2, t_1, t_3, t_4, t_6 \rangle$. The same definition is used in [17] with the addition that tuples can be excluded from some orderings due to their membership uncertainty.

(3) Most Probable Ordering. Similar to the UTop-Prefix query semantics (Definition 6), ω^* is defined as $argmax_{\omega \in \Omega} \Pr(\omega)$, where $\Pr(\omega)$ is computed using Equ 3.2. For example in Figure 6.2, ω^* is the ordering $\omega_1 = \langle t_5, t_1, t_2, t_3, t_4, t_6 \rangle$.

Contrasting the properties of different semantics of ω^*, and studying their potential applications have been addressed in [17, 62, 64, 76]. We focus, however, on building an infrastructure that incrementally computes both $\mathcal{J}_k$ and ω^* under multiple semantics in the context of URankJoin queries.

6.2 Computing the Top-k Join Results

Current rank join methods [36, 37, 51, 57] build on using sorted inputs to incrementally report ranked join results by bounding the scores of non-materialized join results. The proposed techniques mainly differ in the maintained state of partial joins (i.e., joins that may lead to valid join results), which can either be a lightweight state that gives loose score bounds (e.g.,[37]), or a dense state, of all partial joins, that gives tight score bounds (e.g.,[57]). The scoring model in all of these methods is deterministic (i.e., each record has a single score), and hence they cannot be applied to settings with uncertain scores.

The objective of top-k queries is to produce the set of top-ranked tuples based on computed scores. Under the attribute level uncertainty model in Section 3.2, tuples dominated by $\geq k$ tuples can be safely pruned from the answer space of top-k queries. This is done by finding $t_{(k)}$, the k^{th} largest tuple based on score lower bounds (cf. Theorem 6). When computing top-k join queries, we would like to integrate the join operation with such k-dominance criterion such that we produce the top-k join results as early as possible. In the following, we describe how to compute and sort join results incrementally (as needed) using a rank join algorithm that early prunes all dominated tuples.

A common interface to most rank join algorithms, is to assume input relations sorted on per-relation scores, while output (join) relation is generated incrementally in join scores order. We show how to use a generic rank join algorithm RJ, complying with the previous interface, as a building block to compute $\mathcal{J}_k$ (the set of join results dominated by less than k join results) incrementally.

Our algorithm Compute-$\mathcal{J}_k$ assumes two sorted inputs (e.g., indexes) L^i_{lo} and L^i_{up}, for each input relation R_i, giving relation tuples ordered on lo and up scores,

Algorithm 6 Compute Top-k Join Results

```
COMPUTE-$\mathcal{J}_k$($L^1_{lo}, L^1_{up}, \ldots, L^m_{lo}, L^m_{up}$:Ranked Inputs, $k$:Answer Size, $\mathcal{F}$:Scoring Function)
1  RJ$_{lo}$ ← an instance of RJ ($L^1_{lo}, \ldots, L^m_{lo}, k, \mathcal{F}$)
2  RJ$_{up}$ ← an instance of RJ ($L^1_{up}, \ldots, L^m_{up}, \infty, \mathcal{F}$)
3  active$_{lo}$ ← TRUE ; active$_{up}$ ← TRUE
4  $T_{up}$ ← 1 {initialize score upper bound in RJ$_{up}$}
5  count ← 0 {number of results reported by RJ$_{lo}$}
6  while (active$_{lo}$ OR active$_{up}$) do
7      if (active$_{lo}$) then
8          $t$ ← get next result from RJ$_{lo}$
9          count ← count + 1
10         if (count = $k$) then active$_{lo}$ ← FALSE
11         $T_{lo}$ ← score of $t$
12     while ( $T_{up} > T_{lo}$) do
13         Report results available in RJ$_{up}$ with scores > $T_{lo}$
14         $T_{up}$ ← score upper bound in RJ$_{up}$
15     if (NOT active$_{lo}$ AND $T_{up} < T_{lo}$) then
16         active$_{up}$ ← FALSE
```

respectively. By processing the *lo* and *up* inputs simultaneously, COMPUTE-$\mathcal{J}_k$ incrementally computes $\mathcal{J}_k$. This is done by using two instances of RJ, denoted RJ$_{lo}$ and RJ$_{up}$, where RJ$_{lo}$ rank-joins tuples on their overall *lo* scores to find exactly k join results, while RJ$_{up}$ rank-joins tuples on their overall *up* scores to find all join results with *up* scores above the k^{th} largest score reported by RJ$_{lo}$ (cf. Theorem 6). The execution of RJ$_{lo}$ and RJ$_{up}$ is interleaved, such that, at any point during execution, RJ$_{up}$ reports all tuples whose *up* scores are above the last *lo* score reported by RJ$_{lo}$. Tuples in $\mathcal{J}_k$ are reported in *up* scores order to allow for incremental ranking (cf. Section 6.3).

Algorithm 6 gives the details of our method to compute the set of top-k join results $\mathcal{J}_k$.

Pipelined Operator. We design pipelined URANKJOIN query plans by wrapping COMPUTE-$\mathcal{J}_k$ into a query operator. For clarity of presentation, we focus on 2-way joins plans. However, our techniques can also handle multi-way joins. A pipelined operator implementation of COMPUTE-$\mathcal{J}_k$ requires making the algorithm independent of k. The knowledge of k is only available to the query plan root that drives plan execution, while the operator only responds to incoming requests of join results ordered on either *lo* or *up* scores.

A URANKJOIN plan is rooted by ULIMIT, a new operator we propose to drive URANKJOIN plan execution. The operator takes two inputs I_{lo} and I_{up} representing two streams of query output tuples ordered on their *lo* scores and *up* scores, respectively. One GETNEXT implementation is to consume k tuples from I_{lo} and to report tuples in I_{up} with scores above the k^{th} score in I_{lo}. An alternative GETNEXT implementation is to interleave drawing tuples from I_{lo} and I_{up}, similar to Algorithm COMPUTE-$\mathcal{J}_k$.

Algorithm 7 gives the details of the ULIMIT operator that we use to drive the execution of URANKJOIN plan under score uncertainty (cf. Section 6.2).

A URANKJOIN operator is a logical operator that accepts two inputs each has two sorted access paths, corresponding to the *lo* and *up* score orders of the two input relations. The operator produces two output tuple streams corresponding to sorted join results based on *lo* and *up* scores.

Figure 6.3 gives an example logical URANKJOIN query plan. The shown plan rank-joins three relations R, S, and T with uncertain scores x, y, and z, respectively. The bottom URANKJOIN operator uses indexes on the *lo* and *up* scores in Relations R and S as its input access paths, while the top URANKJOIN operator uses indexes on Relation T and the output of the bottom URANKJOIN operator as its input access paths. The ULIMIT operator consumes both *lo* and *up* inputs from the top URANKJOIN operator.

URANKJOIN operator can have different physical implementation. One implementation is to use two regular rank join operators wrapped within a physical operator with 4 inputs (the *lo* and *up* orders of the two input relations) and 2 outputs (the *lo* and *up* orders of the join results). This implementation requires, however, making

Algorithm 7 ULIMIT Operator

```
OPEN($I_{lo}$: $lo$ input stream, $I_{up}$: $up$ input stream, $k$: Answer Size)
1  $I_{lo}$.OPEN()
2  $I_{up}$.OPEN()
3  $\overline{\mathcal{F}_{lo}} \leftarrow 1.0$
4  $count \leftarrow 0$

GETNEXT()
1  while ($count < k$) do
2      $t \leftarrow I_{lo}$.GETNEXT()
3      $count \leftarrow count + 1$
4      if ($count = k$) then $\overline{\mathcal{F}_{lo}} \leftarrow lo$ score of $t$
5  $t \leftarrow I_{up}$.GETNEXT()
6  if ($up$ score of $t > \overline{\mathcal{F}_{lo}}$) then return $t$ else return null

CLOSE()
1  $I_{lo}$.CLOSE()
2  $I_{up}$.CLOSE()
```

other query operators aware of the URankJoin operator input/output interface.

An alternative implementation is to use two separate rank join operators, which allows building URankJoin plan as two parallel plans that can be optimized independently based on available data access paths. We use this implementation in our prototype discussed in Section 6.4. The algebra proposed in [45] can be used in these settings to exploit properties like associativity and commutativity of rank join operators while searching for the query plan with least estimated cost. The logical design of URankJoin operator does not restrict the physical rank join algorithm. Hence, an arbitrary rank join algorithm can be plugged in physical URankJoin plans.

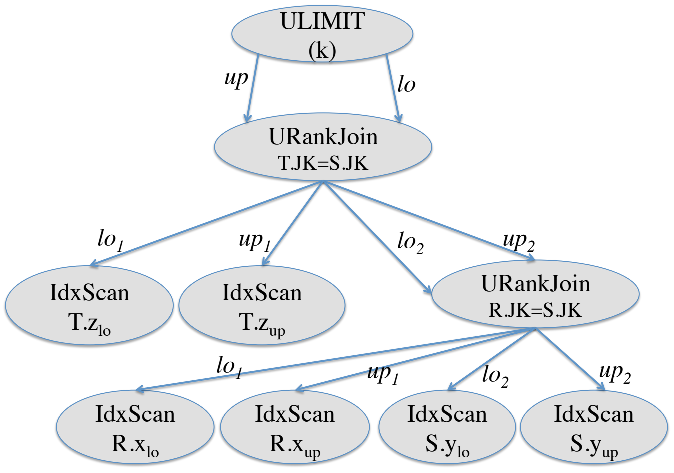


Figure 6.3: A logical URankJoin query plan

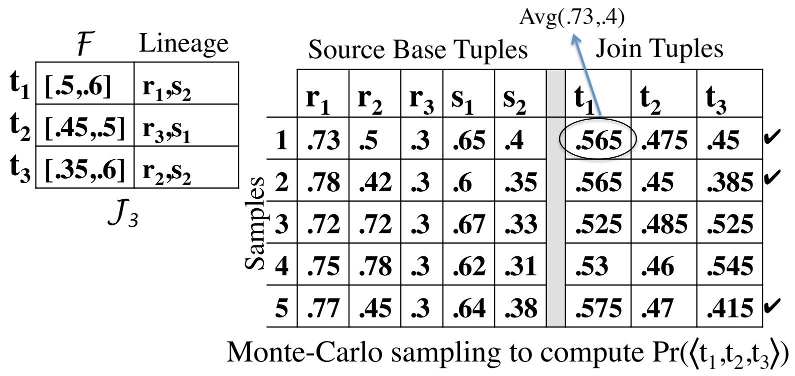
6.3 Ranking the Top-k Join Results

A major challenge for ordering tuples with uncertain scores is managing the exponentially large space of possible orderings (cf. Section 3.2). We tackle this challenge using a sampling-based infrastructure for computing ω^* under multiple semantics based on two novel techniques: *Join-aware Sampling* and *Incremental Ranking*.

6.3.1 Join-aware Sampling

Join induces dependencies among join results. For example in Figure 6.1, (r_1, s_2) and (r_2, s_2) are dependent, since they originate from one tuple $s_2 \in S$ and different tuples in R. Such dependency means that the joint score density of (r_1, s_2) and (r_2, s_2) (which produces the probability of any ordering involving (r_1, s_2) and (r_2, s_2)) is not given by multiplying the marginal score densities of (r_1, s_2) and (r_2, s_2). That is, the score random variables of (r_1, s_2) and (r_2, s_2) are dependent.

We handle score dependencies by associating join results with lineage representing the keys of their origin base tuples. The main idea is to use the space with independent score random variables (i.e., base tuples) as a generator of the space with dependent score random variables (i.e., join results). Hence, the probability of an ordering of


Avg(.73,.4)
Source Base Tuples
Join Tuples
Samples
$\mathcal{J}_3$
Monte-Carlo sampling to compute Pr(⟨t1,t2,t3⟩)

	$\mathcal{F}$	Lineage
t_1	[.5,.6]	r_1,s_2
t_2	[.45,.5]	r_3,s_1
t_3	[.35,.6]	r_2,s_2

	r_1	r_2	r_3	s_1	s_2		t_1	t_2	t_3	
1	.73	.5	.3	.65	.4		.565	.475	.45	✓
2	.78	.42	.3	.6	.35		.565	.45	.385	✓
3	.72	.72	.3	.67	.33		.525	.485	.525	
4	.75	.78	.3	.62	.31		.53	.46	.545	
5	.77	.45	.3	.64	.38		.575	.47	.415	✓

Figure 6.4: Handling score dependencies in Monte-Carlo sampling

(possibly dependent) join results is computed using independent samples drawn from the space of base scores.

Figure 6.4 illustrates our approach. The set $\mathcal{J}_3$ is produced by the URankJoin query in Figure 6.1. Each join result in $\mathcal{J}_3$ is associated with the keys of its origin base tuples. To compute $\Pr(\langle t_1, t_2, t_3\rangle)$, we independently sample a score value in each origin base tuple of $\mathcal{J}_3$ (i.e., a score value in each of $\{r_1, r_2, r_3, s_1, s_2\}$). We simply call such vector of base tuples' score samples a *base sample*. Since base tuple scores are independent, the probability of each base sample is the product of the probabilities of its constituent score values. Applying $\mathcal{F}$ (the average) to score values in a base sample gives a score sample for each join results in $\mathcal{J}_3$. We mark in Figure 6.4 the base samples that correspond to the ordering $\langle t_1, t_2, t_3\rangle$. Such base samples are called *hits*.

We use Monte-Carlo integration to estimate the probability of an ordering of join results based on the proportion of its corresponding hits with respect to the number of samples.

Algorithm 8 shows how to use Monte-Carlo integration method to compute $\Pr(\omega)$, where ω is an ordering of join results (cf. Section 6.3). The union of the lineage of join results in ω is first computed. Independent samples are drawn from the score distributions of base tuples included in the lineage. The drawn scores produce an ordering of join results. If such ordering agrees with ω, then the sample is a *hit*.

Algorithm 8 Compute Probability of Join Results Ordering

```
MC-Probability($\omega$: Join results ordering, $s$: Number of samples)
 1  $sources \leftarrow \bigcup_{t \in \omega} t.sources$ {compute lineage of $\omega$}
 2  $hits \leftarrow 0$ {no. of samples matching ordering given by $\omega$}
 3  $sum \leftarrow 0$ {summation of hits probabilities}
 4  for $i = 1$ to $s$ do
 5      $sample \leftarrow [\,]$ {sample of base tuples scores}
 6      for each tuple $t_i \in sources$ do
 7          $sample[t_i.key] \leftarrow$ random score value based on $f_i$
 8      $\acute{\omega} \leftarrow$ ordering of join tuples based on base scores in $sample$
 9      if ($\acute{\omega}$ agrees with the tuple ordering given by $\omega$) then
10          $hits \leftarrow hits + 1$
11          $sum \leftarrow sum + \Pi_{z \in sample}(\Pr(z))$
12  $v \leftarrow$ volume of hypercube enclosing score combinations in $\omega$
13  return $\frac{hits}{s} \cdot v \cdot \frac{sum}{hits}$
```

The probabilities of hits corresponding to a join results ordering ω are averaged when using Monte-Carlo integration method to compute $\Pr(\omega)$.

We show how to use the previous infrastructure to compute the total order ω^* of tuples in $\mathcal{J}_k$ under *expected ranks* semantics.

The expected rank of a tuple t_i ($ER(t_i)$) is computed as $\sum_{\omega \in \Omega} \Pr(\omega) \cdot \omega[t_i]$ (cf. Section 6.1). Computing $ER(t_i)$ can be done efficiently in our settings by rewriting $ER(t_i)$ as $\sum_{r=1}^{|\mathcal{J}_k|} r \cdot \Pr(t_i, r)$, where $\Pr(t_i, r)$ is the probability of t_i to be at rank r in the possible orderings of $\mathcal{J}_k$. The correctness of the previous rewrite follows from the fact that $\Pr(t_i, r) = \sum_{\omega_{(t_i,r)} \in \Omega} \Pr(\omega_{(t_i,r)})$, where $\omega_{(t_i,r)}$ has t_i at rank r.

We compute $\Pr(t_i, r)$ by considering as a *hit* each sample of base scores that results in the join tuple t_i being at rank r. For example in Figure 6.4, to compute $\Pr(t_3, 1)$, we consider sample 3 and sample 4 as *hits*.

We describe how to compute an ordering ω^* of join results under other probabilistic ranking semantics:

(1) The Most Probable Ordering. In Section 5.5.2, we proposed using Markov Chains Monte-Carlo (MCMC) methods to approximate the most probable ordering by drawing samples from the orderings space biased by probability. The main idea is that for a current sample (ordering) ω_i, and a newly proposed sample ω_{i+1}, we always accept ω_{i+1} if $\Pr(\omega_{i+1}) > \Pr(\omega_i)$, otherwise we accept ω_{i+1} with probability proportional to $\Pr(\omega_{i+1})/\Pr(\omega_i)$. The MCMC method provably converges to the target distribution of possible orderings, and hence it can be used as a generator of orderings biased by their probabilities. Algorithm 8 allows computing $\Pr(\omega)$, where ω is an ordering of join results, and hence applying the MCMC method to approximate the most probable ordering.

(2) Other Semantics. Computing $\Pr(t_i, r)$ allows computing ω^* under other probabilistic ranking semantics. For example, tuple's probability to appear at the top ranks only (Global Top-k [76]) is computed as $\Pr_k(t_i) = \sum_{r=1}^{k} \Pr(t_i, r)$. Similarly, pruning tuples whose probabilities to appear at the top ranks is below a given threshold (probabilistic Top-k threshold [33]) can be done by testing if $\Pr_k(t_i) < T$, for a given threshold T. A third example is finding the ordering with the minimum disagreements with other orderings in the space (Uncertain Rank Aggregation), which can be done in polynomial time using $\Pr(t_i, r)$ values as discussed in Section 5.6.

6.3.2 Incremental Ranking

The size of $\mathcal{J}_k$ can be much larger than k due to score uncertainty. In many Web application scenarios, users only inspect a small prefix of the ranked answers list (e.g., inspecting only a few top hits returned by a search engine). Computing a full ranking of all answers in advance may not thus be always required. We make use of the incremental computation of $\mathcal{J}_k$ (cf. Section 6.2) to incrementally compute an approximation of ω^*.

The main idea is computing bounds on $\Pr(t_i, r)$ for each join result t_i produced by a URankJoin plan. The bounds of $\Pr(t_i, r)$ are used to approximate a prefix of ω^*, and are progressively tightened as more tuples are produced by the URankJoin plan.

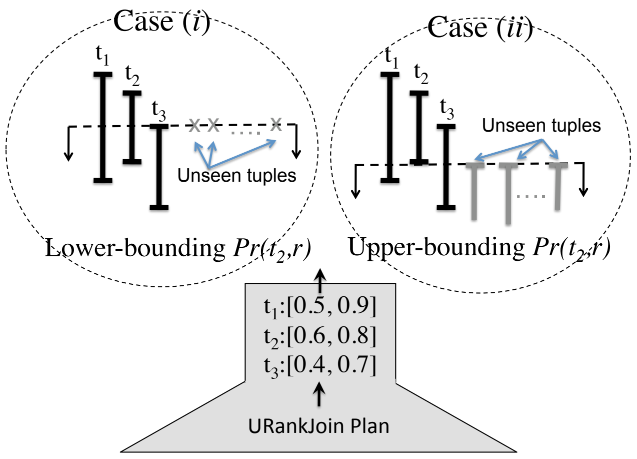


Figure 6.5: Bounding $\Pr(t_2, i)$

Figure 6.5 shows a URankJoin plan that produces tuples in $\mathcal{J}_k$ ordered on their *up* scores. The last produced tuple at this step is t_3. Assume that we need to compute $\Pr(t_2, r)$. We identify two extreme cases:

- Case (i): the scores of all non-retrieved tuples are deterministic values (shown as '$\times$' symbols in Figure 6.5) located at the largest possible unseen score (i.e., up_3).
- Case (ii): the *up* scores of all non-retrieved tuples are below lo_2 (shown as shaded intervals in Figure 6.5).

Each case gives a possible configuration of unseen tuples in $\mathcal{J}_k$. By applying Monte-Carlo sampling to each configuration, we obtain a bound on $\Pr(t_2, r)$. Specifically, Case (i) gives a lower bound on $\Pr(t_2, r)$, denoted $\underline{\Pr}(t_2, r)$, since the scores of all unseen tuples are maximized, while Case (ii) gives an upper bound on $\Pr(t_2, r)$, denoted $\overline{\Pr}(t_2, r)$, since the scores of all unseen tuples are minimized. By seeing more tuples in $\mathcal{J}_k$, computed bounds are tightened (i.e., $\underline{\Pr}(t_2, r)$ increases and $\overline{\Pr}(t_2, r)$ decreases) since the maximum score of an unseen tuple decreases. When the maximum score of an unseen tuple is below lo_2, both bounds coincide at $\Pr(t_2, r)$. Note that this bounding method is valid only if tuples in $\mathcal{J}_k$ are retrieved in *up* score order.

The bounds of $\Pr(t_i, r)$ can be used to compute rankings under multiple probabilistic ranking semantics. For example, in Global top-k [76] (rank by tuple's probability to appear within the top k ranks), we bound $\Pr_k(t_i)$ as $\underline{\Pr_k}(t_i) = \sum_{r=1}^{k} \underline{\Pr}(t_i, r)$, while $\overline{\Pr_k}(t_i) = \min(1, \sum_{r=1}^{k} \overline{\Pr}(t_i, r))$. The Global top-$k$ ranking of retrieved tuples from $\mathcal{J}_k$ can thus be approximated as follows. We set $\omega^*[t_i] < \omega^*[t_j]$ if $\underline{\Pr_k}(t_i) > \overline{\Pr_k}(t_j) - \epsilon$, where $\epsilon \in [0, 1)$ is a given acceptable error in tuples relative order. The underlying URankJoin plan is incrementally requested for new join results until the computed ω^* prefix satisfies the previous error constraint.

6.4 MashRank Research Prototype

We describe the architecture and implementation details of MashRank, a research prototype we have built to apply and experiment with our techniques in Web mashups scenarios[†].

Current mashup systems (e.g.,[1, 2, 3, 60]) allow creating data flows involving services, sources, and operators. Most systems assume data in the form of pre-computed structured feeds, with the exception of [61], which integrates text extractors into enterprise mashups. However, ranking is mostly overlooked in these works by generating non-optimized plans (e.g., materialze-sort plans) for ranked mashups. Moreover, although uncertainty is ubiquitous on the Web (e.g., missing/inexact values), current systems do not allow querying/reasoning about such uncertainty. MashRankintegrates concepts from information extraction, rank-aware processing, and probabilistic databases domains to address these problems.

6.4.1 MashRank Architecture

MashRank is a Web-accessible system (demonstrated in [68]), with client-side query processing, and server-side data retrieval and extraction. We describe the details of different components in MashRank architecture (given in Figure 6.6).

[†]Current prototype is accessible at http://prefex.cs.uwaterloo.ca/MashRank.

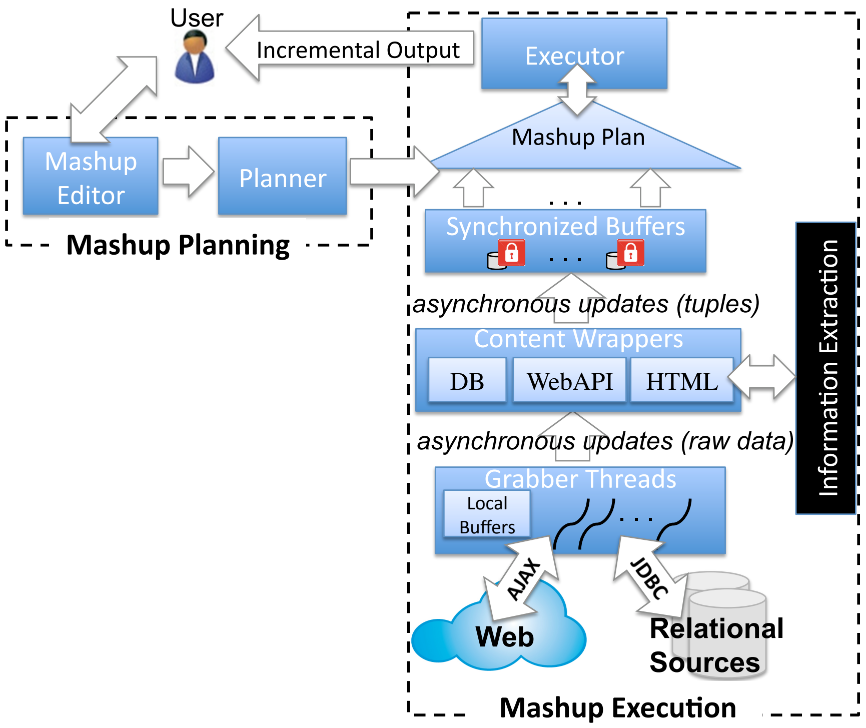


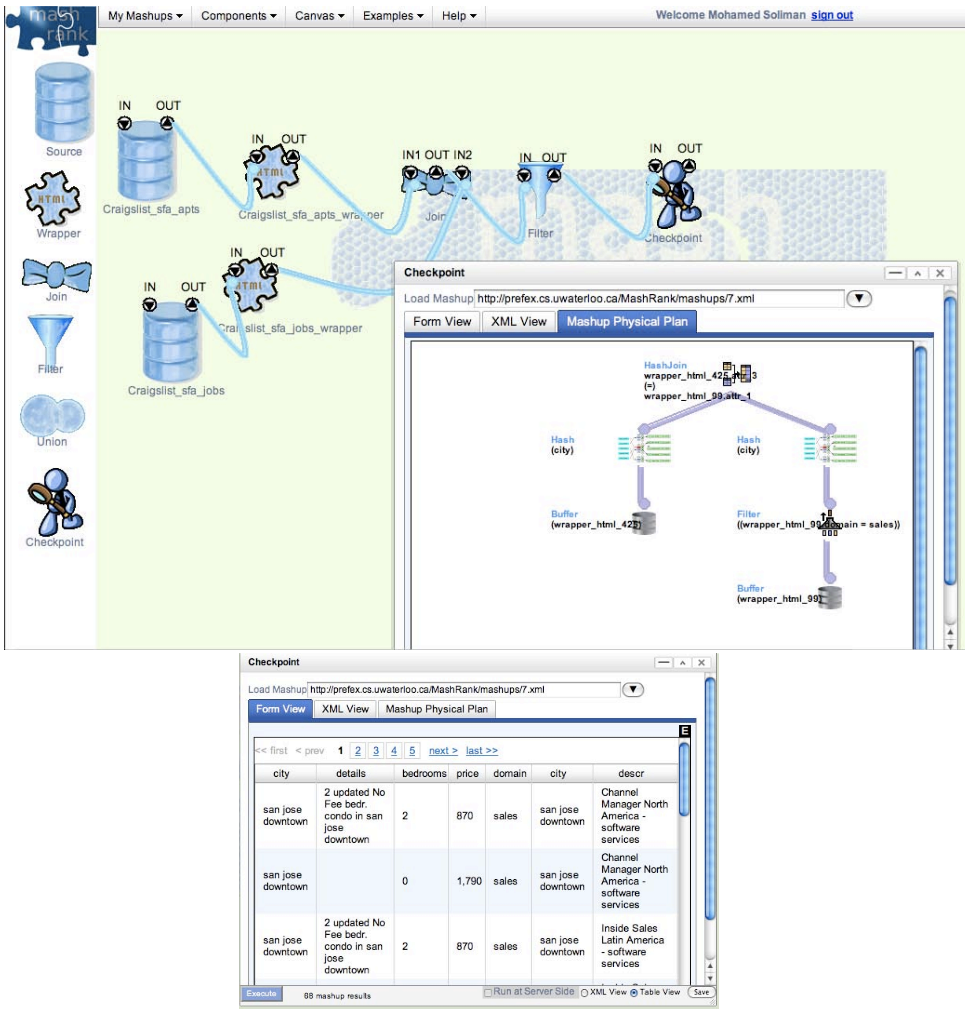
Figure 6.6: MashRank architecture

Mashup Editor builds a mashup data flow, by interacting with the user, to identify source schemas, join/filter conditions, and scoring function. A mashup data flow is a tree whose leaves are the sources, and internal nodes are three primary logical operators: extractors, joins, and filters[†]. The edges between tree nodes are pipes indicating the flow of tuples from one logical operator to its parent.

Mashup Planner maps the mashup data flow into a rank-aware and uncertainty-aware physical plan. A mashup physical plan needs to be rank-aware if the user provides a scoring function to order mashup results. The physical plan needs to be uncertainty-aware if the scoring function involves at least one uncertain attribute.

Mashup Planner exploits sorting capabilities of input sources to offload sort to source side. For example, if the scoring function involves an attribute that has sorted

[†]Our framework is extensible, which allows other logical operators such as union and intersection.



city	details	bedrooms	price	domain	city	descr
san jose downtown	2 updated No Fee bedr. condo in san jose downtown	2	870	sales	san jose downtown	Channel Manager North America - software services
san jose downtown		0	1,790	sales	san jose downtown	Channel Manager North America - software services
san jose downtown	2 updated No Fee bedr. condo in san jose downtown	2	870	sales	san jose downtown	Inside Sales Latin America - software services

Figure 6.7: A Screenshot for MashRank prototype

access (as provided by its corresponding source), the created mashup plan pushes source records directly (i.e., without a sorting phase) into rank-aware mashup execution. This also allows limiting extraction on pages that are not required to compute the top-ranked results (by upper bounding the scores of records not yet extracted).

Figure 6.7 shows a mashup data flow, constructed using MashRank editor, a corresponding mashup physical join plan, and a listing of computed mashup results. We elaborate on planning and sort offloading in Section 6.4.3.

Content Wrappers bridge different data models to the relational model.

Mashups may involve data sources of different models (e.g., XML generated by Web API's, raw HTML, and relational data). MashRank adopts a simple relational model in which mashup (intermediate) results are represented as tuples.

Wrapping HTML into relational tuples is complicated by the lacking of schema information. We build HTML wrappers based on the concepts of *wrapper induction* [42] in information extraction. The idea is to request user to provide a number of labeled examples of different attributes in a sample of source pages. A wrapper inductor learns a rule that correctly extracts all of the given examples. For example, the rule can be maximal strings in HTML source that delimit all of the examples. Applying the rule to other source pages, with the same structure of labeled pages, produces the required tuples. We elaborate on wrapper induction details in Section 6.4.2

Grabber Threads grab data from mashup sources. When sources are accessible through URLs, MashRank initiates a thread for each URL to grab contents. Each thread forwards source response, once ready, to the appropriate wrapper. This allows MashRank to process multiple requests in parallel, and avoid getting blocked on slow sources. When sources are relational, a database connection is used to retrieve tuples from remote database.

Mashup Executor executes the physical plan generated by the Planner against live data sources, and incrementally reports mashup results to the user. We build on the iterator model (OPEN-GETNEXT-CLOSE), used in most DBMSs, for mashup execution. OPENING the root operator in a query plan tree recursively initializes all tree operators. Processing the query is done by calling the GETNEXT method of the root operator repeatedly until it returns an empty result. Finally, CLOSING the root operator recursively shuts down all operators in the tree.

However, in contrast to relational plans that read data from tables with known sizes residing on disks, a mashup plan may read data from remote sources of unknown sizes filling up asynchronously as more tuples are extracted. Hence, mashup execution needs to be (1) *synchronized*: to guarantee correct reads/updates, sources are locked when wrappers attempt to append new extracted tuples, or when parent nodes in the mashup plan attempt to read next tuple; and (2) *push-based*: an operator requesting tuples waits if no new tuples are currently available, and wrappers are not done processing source contents. Once new records are available, they are pushed into plan execution by notifying all waiting requesters.

Each mashup source has a dedicated *synchronized buffer* satisfying these two requirements. Synchronized buffer owns a monitor (lock) to prevent concurrent reads and updates. Wrappers writing to the buffer, as well as mashup plan nodes, reading from the buffer, must obtain buffer's lock before accessing its data. If a read/write request is being served, all other requests are forced to wait until the request being served completes. As soon as the request completes, a notification message is issued to wake up all waiting requests to re-attempt accessing the buffer.

MashRank executor interleaves extraction with query processing such that none of the two tasks blocks the other. Moreover, variance in source response times is tolerated by allowing asynchronous updates as soon as source responds with contents, as opposed to blocking until source responds. Hence, the execution is geared toward early-out of mashup results, if possible, while extraction and query processing are in progress.

6.4.2 Information Extraction

Information extraction techniques approach unstructured data from different perspectives. Supervised learning methods (e.g., [6, 42]), learn extraction rules from a set of user-specified examples by generalizing common properties in these examples. On the other hand, unsupervised learning methods (e.g.,[19, 59]) focus on learning a grammar/template describing the schema of the underlying source by exploiting repeated structure and domain knowledge. The learned template can be used to populate relational tables out of the unstructured sources. MashRank provides a mashup authoring tool that builds on supervised extraction methods, namely wrapper induction. We allow users to annotate and refine examples, during mashup data flow creation, and use these examples to learn extraction rules. In the following, we present our adaptation of wrapper induction techniques.

MashRank uses wrapper induction techniques to transform unstructured sources into relational (structured) sources. The details of the wrapper induction algorithm are orthogonal to mashup planning and processing in MashRank. We assume an interface to the wrapper inductor with three main functions: (1) *addExample*: adds a new training example (e.g., a text node representing the value of some attribute); (2) *learn*: processes the training examples using the induction algorithm to compute an extraction rule; and (3) *extract*: applies the learned extraction rule to a given page, and returns a set of extracted records.

The previous interface is generic, and applies to multiple wrapper induction proposals (e.g.,[6, 42]). We elaborate on the implementation of the interface in our adaptation of [42]. We emphasize, however, that information extraction is a black-box in MashRank, and hence other techniques can be integrated with MashRank to conduct more sophisticated extraction.

The inductor in [42] treats each HTML page as a sequence of characters, and learns extraction rules in the form of string patterns. The learned rule extracts attributes from the page source in a round-robin fashion, and binds them into records. This method can generate erroneous records when some attribute values are missing. Since missing values are common on the Web, we adapt this method by learning extraction rules on attribute level, and then bind extracted values into records based on their proximity in the HTML source. We describe our technique in the following.

For a schema $\langle a_1, \ldots, a_n \rangle$ of n attributes, the function *addExample* receives as input a triple (a_i, s, e), where a_i is a schema attribute, while s and e are the start and end character positions of one example value of a_i in the HTML source. In MashRank editor, this is enabled by allowing the user to highlight pieces of text inside the page as examples for each required attribute.

The function *learn* computes an extraction rule for each attribute a_i in the form of a pair of strings (l_i, r_i). The rule is interpreted as follows: all values of attribute a_i appearing in the underlying page are enclosed between two strings l_i and r_i. For example, one possible extraction rule for hotel name could simply be ("$<b>$", "$</b>$"). The strings l_i and r_i are computed by scanning the characters appearing before and after all training examples, and appending these characters to l_i and r_i, respectively, as long as all examples agree on the scanned character. We stop when finding maximal patterns in the sense that by appending more characters to any of l_i and r_i, at least one training example is not matched.

The function *extract* applies the extraction rule of each attribute to extract a set of attribute values. We align extracted values to form records based on their proximity in the page. We process attributes in the order in which they appear in the HTML source (e.g., *name* appears before *price*), and within each attribute, we process extracted values in the order of their appearance in the HTML source. We start by assigning each extracted value in the first attribute to a new record. For each subsequent attribute a_i, we assign attribute value v to the record that has an attribute a_j, with $j < i$, whose value is the closest value preceding v. If such record cannot be found, v is assigned to a new record with empty values in all attributes a_j for $j < i$, and value v in attribute a_i.

In our experimental study we evaluate extraction accuracy by counting as an error any extracted record with wrong information (e.g., missing values that should be non-missing, or wrongly linked values of different attributes). We manually computed a ground truth of all correct records to be extracted, and compared the output of our extraction technique to ground truth. Figure 6.8 shows extraction quality in precision, recall, and F1 measures, computed on a sample of 50 pages of each Web source used in our experiments (cf. Figure 6.11). We achieve perfect extraction in almost half of the sources, and very high accuracy on the rest. We note that the extraction method we adapt depends on regularity in HTML source, which may limit its applicability to some sources. However, as we discuss in Section 6.4.1, we treat information extraction as a blackbox, and hence other variants of extraction methods can also be plugged in MashRank framework.

Source	Precision	Recall	F1
Vianet	1.0	1.0	1.0
TvTrip	1.0	0.92	0.95
Menus	0.97	0.96	0.97
Epinion	1.0	1.0	1.0
Flickr	0.92	1.0	0.96
Pubs	1.0	1.0	1.0
GScholar	1.0	0.94	0.97
Apartments	1.0	1.0	1.0
Restaurants	0.975	0.97	0.972

Figure 6.8: Information extraction precision/recall

6.4.3 Mashup Planning

Query optimizers use statistics collected on queried relations, and query predicates, to prune query plans that are expected to perform poorly. In our settings, we usually have no prior knowledge about data sources, as they may be remote sources given by the user in an ad-hoc fashion. We thus resort to exploiting the configuration of mashup data flow to build a feasible mashup physical plan. Nevertheless, building mashup planning on a cost model can be quite important in many other scenarios (e.g., mashing up sources that the system has prior knowledge on, asking for user input to characterize cost factors of mashed up sources, or sampling the sources to compute estimates on their cost factors). We leave cost modeling as an important future extension of this work.

Given a ranked mashup with a scoring dunction $\mathcal{F}$, MashRank Planner starts by labeling each node in the mashup data flow with its corresponding ranking attributes (attributes that appear in $\mathcal{F}$). The labeling starts with leaves (data sources), where each source is labeled with the ranking attributes it covers. Then, moving up in the data flow tree, the union of the ranking attributes of all children of a node p gives the ranking attributes of p.

After labeling is done, the Planner processes the labeled data flow starting from the root, mapping each node to one or more physical operators, and then recursing on nodes' children.

A source node is mapped to a *synchronized buffer* (cf. Section 6.4.1). An extractor node with empty ranking attributes is mapped to a *scan* operator. An extractor node with non-empty ranking attributes is mapped to *sort* operator, on top of a *scan* operator, so that all source tuples are sorted based on the scoring function (ranking attributes not belonging to the source assume the largest possible score). Since we assume monotone scoring functions, using such sort expression guarantees tuples flowing out of the source in the right order. A join node with empty ranking attributes is mapped to either a *nested-loops join* operator, or a *hash join* operator if the join condition is non-equality or equality, respectively. Similarly, a join node with non-empty ranking attributes is mapped to a *nested-loops rank join* operator, or a *hash rank join* operator[†] if the join condition is non-equality or equality, respectively. Finally, a filter node is mapped to a *filter* operator with the node's Boolean condition. We also implemented techniques to push down filtering operations to their relevant sources, as typically done in relational query optimizers.

When the scoring function includes one or more uncertain attributes, the Planner generates a URankJoin plan (cf. Section 6.2). The above procedure is followed to generate two identical rank join plans, where one plan rank-joins tuples on their *lo* scores, while the other plan rank-jons tuples on their *up* scores. A *ulimit* operator is used as the parent operator of the two plans, and a *probranker* operator (implementing our MC-based sampling methods) is added as the parent of *ulimit*.

We describe our plan generation algorithm using the following rank join query:

```
 SELECT *
FROM vianet, tvtrip
WHERE vianet.HotelName ∼ tvtrip.HotelName
```

[†]The Hash Rank Join (HRJN) algorithm [37] iteratively selects an input relation to read its next tuple. Each new tuple is hashed on its join attribute in a per-relation hash table to facilitate creating join results. The join results are created by finding, for each new tuple, the joinable tuples currently read from other relations. Join results are stored in a priority queue ordered on score. The scores of non-materialized join results are upper-bounded by assuming best-case joins, where tuples with the highest scores in all inputs, but one, join with the last retrieved tuple from the excluded input.

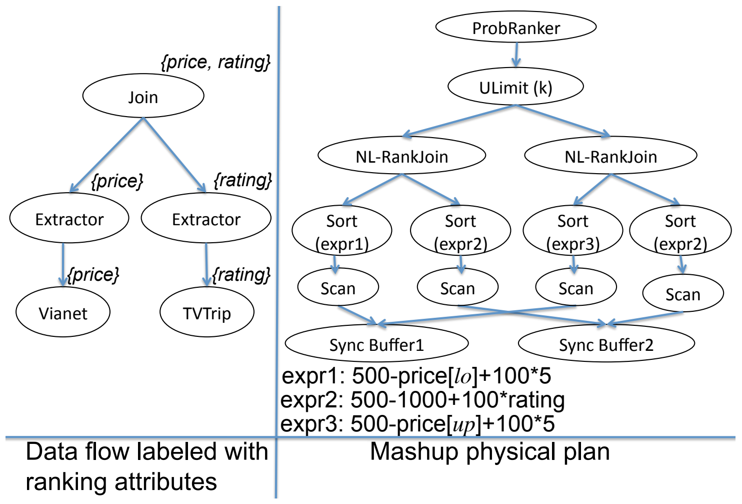


Figure 6.9: Generating mashup plan

```
ORDER BY 500-vianet.Price+ 100* tvtrip.Rating
LIMIT k
```

The scoring function includes two attributes *price*, and *rating*, where *price* is an uncertain attribute. The join condition is approximate equality of hotel names (implemented in MashRank as a thresholded edit distance similarity function). Figure 6.9 shows the data flow nodes after being labeled with ranking attributes. The generated physical plan is a nested-loops rank join plan (since join condition is non-equality). The Planner adds a *ulimit* operator to drive the execution of the *lo* and *up* rank join plans, and a *probranker* operator to conduct probabilistic ranking. The sort expressions ($expr_1$, $expr_2$, and $expr_3$) are created by replacing ranking attributes not covered by the underlying source with their largest possible scores.

Offloading Sort to Web Sources. In rank-aware query processing, the existence of sorted access methods on ranking attributes is crucial for pipelining ranked results efficiently. Implementing such access methods as a sort operator per input (e.g., as in Figure 6.9) introduces a bottleneck in query execution due to the blocking nature of sort. When an index on a ranking attribute already exists, a rank-aware plan can benefit from such cheap sorted access method to pipeline ranked query results efficiently.

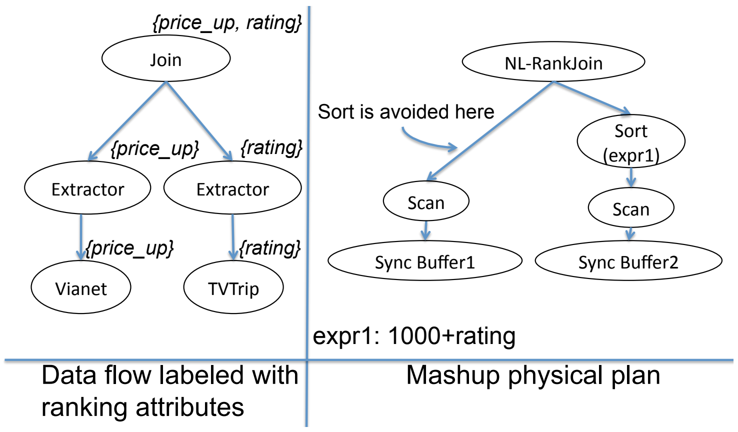


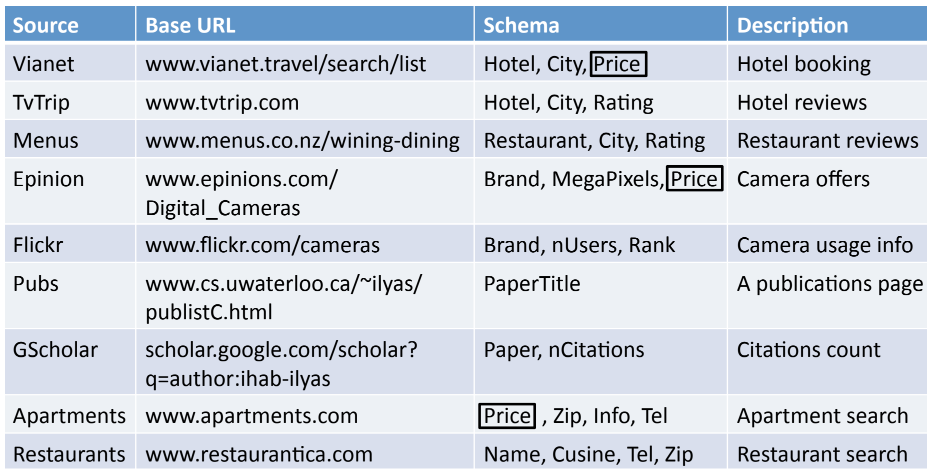
Figure 6.10: Generating mashup plan with offloaded sorting

In our settings, we build mashups on arbitrary sources selected by the user, and hence we cannot generally assume the existence of indexes on these sources. However, many Web sources provide sorting capabilities to view query results ordered on some attribute. Such information is obtained from the user in the form of a special sorting parameter that can be appended to page URLs. By offloading sort to source side, we allow rank-aware mashup plan to pipeline sorted results, as they are extracted from the sources.

For example, assume the following mashup query, where Vianet is declared by the user as a source that can produce records ordered on Price_up (the highest prices):

```
 SELECT *
FROM vianet, tvtrip
WHERE vianet.HotelName ∼ tvtrip.HotelName
ORDER BY vianet.Price_up+ tvtrip.Rating
LIMIT k
```

Fiigure 6.10 shows the mashup data flow labeled with ranking attributes, and the corresponding physical plan generated by MashRank Planner. Note that the Planner did not add a sort on top of the scan of Vianet, since it leveraged the fact that records of Vianet are pre-sorted, and hence they can be directly pipelined into the NL-RankJoin operator.



Source	Base URL	Schema	Description
Vianet	www.vianet.travel/search/list	Hotel, City, $\boxed{\text{Price}}$	Hotel booking
TvTrip	www.tvtrip.com	Hotel, City, Rating	Hotel reviews
Menus	www.menus.co.nz/wining-dining	Restaurant, City, Rating	Restaurant reviews
Epinion	www.epinions.com/ Digital_Cameras	Brand, MegaPixels, $\boxed{\text{Price}}$	Camera offers
Flickr	www.flickr.com/cameras	Brand, nUsers, Rank	Camera usage info
Pubs	www.cs.uwaterloo.ca/~ilyas/ publistC.html	PaperTitle	A publications page
GScholar	scholar.google.com/scholar? q=author:ihab-ilyas	Paper, nCitations	Citations count
Apartments	www.apartments.com	$\boxed{\text{Price}}$, Zip, Info, Tel	Apartment search
Restaurants	www.restaurantica.com	Name, Cusine, Tel, Zip	Restaurant search

Figure 6.11: Web sources used in experiments (boxes indicate uncertain attributes)

We discuss how we modify our data grabbing module (cf. Section 6.4.1) to exploit sorted access methods. Our multi-threaded architecture spawns a grabber thread per page to avoid blocking on slow sources. When each thread grabs data from one of the pages in a sorted retrieval, some of the pages can be ready for extraction before others, due to differences in source response time. We thus need to an maintain a page-level order to guarantee pipelining records into mashup execution in the right order. This is done by associating each thread with an order reflecting the position of the thread's page in the ordered retrieval of source pages. Tuple requests are answered while respecting such page-level order. That is, a tuple is not reported from page p unless all tuples in pages with orders preceding p have been already reported.

6.5 Experiments

All experiments are conducted on a 2.2GHz client machine with 2GB RAM, and 80GB hard disk. The techniques we described in this chapter are implemented within

M1

```
SELECT *
FROM Vianet v, TvTrip t,
        Menus m
WHERE v.Hotel ≈ t.Hotel
         AND v.City=m.City
ORDER BY 500-v.Price+ 100*
(t.Rating+m.Rating)
LIMIT k
```

M2

```
SELECT *
FROM Epinion e, Flickr f
WHERE e.Brand contains
f.Brand
ORDER BY e.Price+
            (100-f.Rank)
LIMIT k
```

M3

```
SELECT *
FROM Pubs p, GScholar g
WHERE p.PaperTitle ≈ g.Paper
ORDER BY nCitations
LIMIT k
```

M4

```
SELECT *
FROM Apartments a,
Restaurants r
WHERE a.Zip = r.Zip
ORDER BY a.Price
LIMIT k
```

Figure 6.12: Mashup examples used in experiments

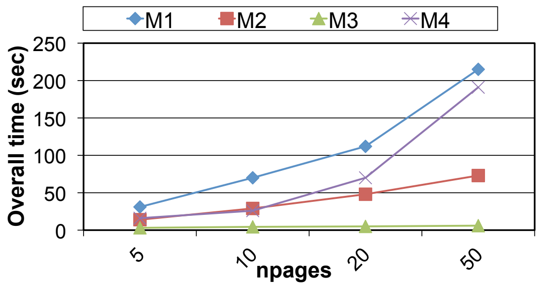


Figure 6.13: Overall execution time

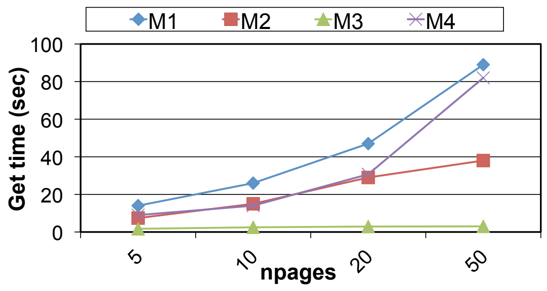


Figure 6.14: Avg source grabbing time

MashRank, a mashup authoring and processing tool that integrates information extraction techniques with ranking under uncertainty. MashRank allows users to define extraction rules to compute structured records out of unstructured HTML sources. Users can then formulate mashups among live Web sources as relational join queries, possibly with ranking requirements to order mashup results, and uncertainty in the extracted attribute values. We thus use mashups as practical examples of rank join queries under the attribute level uncertainty model. We give an overview of the architecture and implementation details of MashRank prototype in Section 6.4.

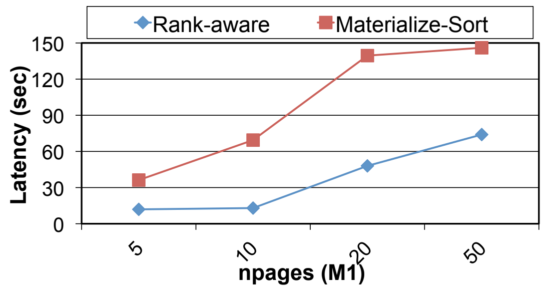


Figure 6.15: Sort offloading

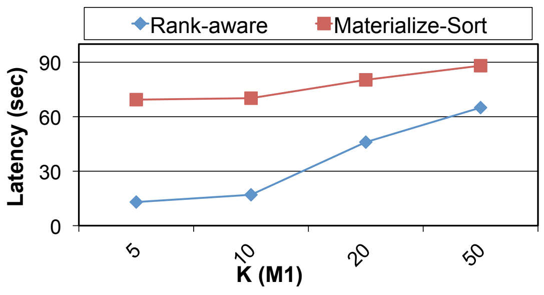


Figure 6.16: Sort offloading with k

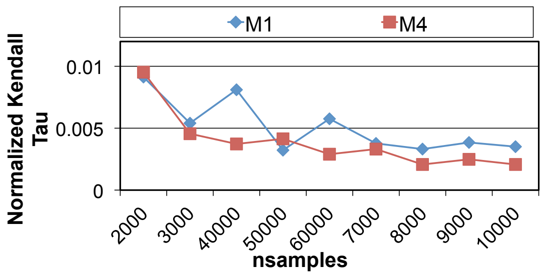


Figure 6.17: MC Convergence

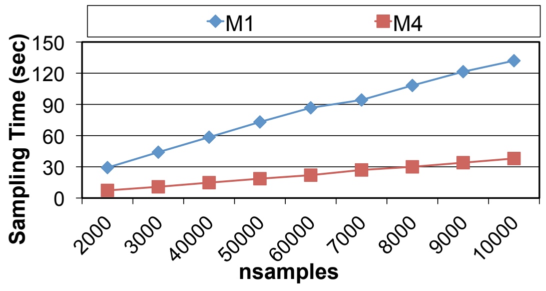


Figure 6.18: MC processing time

While many mashup examples are implementable using MashRank, we use 4 examples to show effectiveness and scalability of our solutions. Figure 6.11 gives the details of involved Web sources, while Figure 6.12 shows (in SQL-like syntax) the mashup examples we build. Note that Price attribute in many of the shown sources follows the attribute level uncertainty model, since it was given as a range of possible values. Join conditions and scoring functions are selected based on source schemas. We assume uniform score distributions for all uncertain scores (in general, uncertain join scores can be non-uniform even if base scores are uniform).

Our main performance metrics are (1) *latency*: time before returning first result, and (2) *overall time*. We control values of three parameters: (1) k: number of tuples dominating any produced result is $< k$ (default is 1), (2) *npages*: maximum number of pages per source, (default is 10), (3) *nsamples*: number of Monte-Carlo samples, (default is 10,000). When changing one parameter value, we keep other parameters at their defaults.

Scalability with respect to Source Size

We evaluate the performance of query evaluation as *npages* increases. In general, processing time increases sub-linearly with *npages*. Figure 6.13 shows that *overall time* has increased by an average of 5.5 times, as *npages* increased by 10 times. Since query evaluation is conducted on records extracted on-the-fly, most of the processing time is consumed in grabbing data from online sources. Figure 6.14 shows the average *get time* consumed in grabbing a source page. Grabbing data from multiple source pages is parallelized in MashRank using a threaded implementation. Due to the added overhead of grabber threads synchronization, *get time* increased by an average of 4.5 times as *npages* increased by 10 times. We also measured the average extraction time per page, which had a value below 0.5 seconds in all sources.

Sort Offloading in Rank-aware Processing

We evaluate the performance impact of sort offloading, where we use available sorted access methods in a Web source to pipeline its extracted records to rank-aware query evaluation (as opposed to blocking until all records are extracted, and then applying a sort operation). We compare the generated plan to the conventional materialize-sort plan (i.e., compute all mashup results then sort) that does not exploit pre-sorted records. Figure 6.15 shows that sort offloading and rank-aware processing improved *latency* by an average of 66% over different *npages* values, while Figure 6.16 shows that the average *latency* improvement is 56% over different k values.

Monte-Carlo Sampling

Figure 6.17 illustrates convergence of ω^* under expected ranks semantics (cf. Section 6.3). We use normalized Kendall tau distance as our convergence measure. Kendall tau distance between two orderings is the number of pairs of items with disagreeing relative orders in the two orderings. We measure Kendall tau distance between each two orderings produced using two consecutive sample sizes. As *nsamples* increases, the distance decreases indicating that ω^* approaches stability. Figure 6.18 shows the time consumed in generating MC samples of join tuples. The sampling algorithm shows linear increase in time with respect to *nsamples*.

6.6 Summary and Lessons Learned

In this chapter, we present evaluation techniques for the novel problem of rank join under uncertainty given in Section 6.1. Our proposal initiates a study of the integration of relational optimizations (e.g., rank join), and the concepts of probabilistic databases towards building a relational+probabilistic query engine. We give an implementation of probability and rank-aware join operators, and an infrastructure for uncertain rank-join under multiple probabilistic ranking semantics using Monte-Carlo simulation.

We design a system architecture that integrates information extraction with query processing within a push-based execution model. We also present the technical details of MashRank, a novel research prototype that addresses integrating information extraction with joining and ranking under uncertainty in the context of Web mashups.

Lessons Learned. Based on the study and the experiments presented in this chapter, we make the following high level observations:

- Building an integrated uncertainty-aware and rank-aware join operator trigger changes in the input/output interfaces of relational operators. For example, our logical operator design is based on consuming input tuples in two different orders (based on the smallest and largest possible tuple scores). Physical operator implementation can be realized as two parallel rank join operators, or as an integrated operator with modified input/output interfaces.

- Ranking with uncertain scores using sampling is a blocking process that needs to be completed before ranked answers are reported. Our incremental ranking algorithm proposes a possibility for pipelining the output of the sampling process, with accuracy guarantees, in order to avoid the blocking nature of the sampler.

- Integrating sampling methods with the execution of relational operations creates new factors for costing a relational ranking query plan such the sampling budget and its interaction with the accuracy of reported results.

- When processing ranked mashups against live remote Web sources, network latency is a dominating performance factor. In general, minimizing the number of needed-to-see tuples/pages is an important optimization that all of our techniques build on. This optimization is highly effective for online computation of ad-hoc mashups. In this context, the availability of sorted access methods to Web pages in remote sources greatly improves the performance by allowing only the needed pages to be downloaded and processed by information extraction techniques. Indexing or caching Web sources is another important optimization that we did not experiment with in our study. One limitation of this optimization is the potential staleness of cached/indexed Web data. Building an integrated solution that closely maintains the freshness of (uncertain) extracted data, and efficiently supports the computation of ranking queries is an interesting research problem.

Chapter 7

Conclusions, Limitations, and Future Work

In this chapter we conclude this dissertation, discuss the limitations of our proposal, and present future research directions.

7.1 Conclusions

This dissertation presents our work in supporting ranked retrieval in uncertain and probabilistic databases. Our study provides insights on the interplay between ranking and uncertainty models, and introduces efficient mechanisms to compute ranking queries under different types of data uncertainty.

We introduced new probabilistic formulations for top-k queries under possible worlds semantics. Under tuple level uncertainty, we modeled the problem as a state space search, and described several query processing algorithms with guarantees on the number of accessed tuples and the size of the materialized search space. Our processing methods integrate tuple retrieval, ranking, and uncertainty management in the same framework, while leveraging existing indexing and query processing facilities in RDBMSs.

We proposed a new probabilistic model based on partial orders to encode uncertainty in tuple scores. We formulated several ranking queries under this model, and designed a set of efficient query evaluation techniques. We also showed that our methods can be used to compute rank aggregation queries in polynomial time under some classes of partial orders.

Finally, we formulated the novel problem of rank join with uncertain scores. We proposed a pipelined operator implementation that integrates joining, ranking, and uncertainty handling in one operator that can be used in relational query plans. We also designed and implemented MashRank, a functional research prototype that realizes our query formulations and processing techniques in the context of data mashups on the Web.

7.2 Limitations

We believe that this dissertation has initiated a new line of research for understanding and formulating ranking problems in the context of uncertain and probabilistic databases. However, similar to any other work within a time frame, our proposal has some limitations that we discuss in this section. Some of these limitations can be handled by further analysis of the given methods or extending the experimental study, while others are intrinsic in the design of our models and techniques, and may be addressed by adopting different approaches and/or technical tools.

Some limitations of our query semantics have been raised by [17], which gives a number of plausible properties of ranking semantics, and show that they are not completely covered by our proposed query semantics (cf. Section 2.4). For example, under some configurations of tuple dependencies, it can be shown that UTop-Prefix query answers do not satisfy the *containment* property (i.e., the most probable top-$(k+1)$ vector does not necessarily include the most probable top-k vector). However, it is also possible to formulate other plausible properties that are not maintained under the query semantics proposed in other works. For example, ranking based on *expected* *ranks* [17] can produce answers that are easily thrown off with the existence of outliers. This means that we might not be able to report tuples with very high scores and probabilities if the majority of other tuples have low scores and/or probabilities. Hence, we believe that, at this point of time, there is no one-fits-all proposal for formulating and evaluating ranking queries on uncertain data. An important challenge to be tackled is inferring the appropriate semantics and evaluation mechanisms from the characteristics of the underlying data and the requirements of target applications.

A related limitation is due to adopting the concept of finding the most probable answer of a ranking query. The same concept is adopted in related AI inference problems, where finding the Maximum A Posteriori (MAP) estimation or the Most Probable Explanation (MPE) are highly popular questions. However, one problem with adopting this concept is that when the distribution of possible answers is large, and the probabilities of these answers are close to each other, finding the most probable answer may not be as informative as adopting other aggregation-based semantics (e.g., the optimal rank aggregation). Finding indicative features in the underlying data, to be used for adaptively formulating the answers of ranking queries, is a related challenging problem.

Another limitation is related to the theoretical analysis of the error involved in computing numerical integrals using the Monte-Carlo method. Our proposal makes the assumption that the number of Monte-Carlo samples is selected so that the approximation error of computed numerical integrals is negligible. In order to understand the effect of such approximation error, we need to conduct deeper analysis of the behavior of our algorithms. For example, let the estimated value of $\Pr(t,i)$ (the probability of a tuple t to appear at rank i) using Monte-Carlo integration be $\hat{\Pr}(t,i)$. Then, based on the guarantees of the Monte-Carlo method (cf. Section 2.3), we know that if the number of samples is in $\Omega(\frac{1}{\Pr(t,i)\cdot\epsilon^2} ln(\frac{1}{\delta}))$, for a given $\epsilon \in (0,1]$ and $\delta \in (0,1]$, we have $\Pr\ (\ |\Pr(t,i) - \hat{\Pr}(t,i)| \ \leq\ \epsilon\ \cdot\ \Pr(t,i))\ \geq\ (1-\delta)$. A crucial challenge for finding the required number of samples that achieves the previous guarantee is to efficiently compute a tight lower bound on $\Pr(t,i)$.

One final limitation is related to our experimental methodology. Our experiments mainly measure the efficiency and scalability of our methods in various contexts. However, an orthogonal and important experimental dimension that we did not explore is user's satisfaction with reported results. This dimension raises multiple questions that are interesting to answer through user studies. For example, which query semantics make more sense from user's perspective? what classes of applications in which users can sacrifice the guarantees of reported answers in favor of more efficient processing? how much users can appreciate uncertainty-aware ranking compared to a traditional ranking that eliminates uncertainty from the underlying data?

7.3 Future Work

Our future work includes studying ranking queries under more general uncertainty models, solving the problem of aggregating multiple partial orders, and mapping the requirements of probabilistic ranking queries into low-level modifications in query engines. In the following, we give a high level overview on each of these extensions.

7.3.1 Ranking Under More General Uncertainty Models

The study presented in this dissertation treats tuple level and attribute level uncertainty models in isolation. While our proposed query semantics are generic and apply to any uncertainty model that can be interpreted under possible worlds semantics, the processing techniques we introduced are dependent on the underlying type of uncertainty. For tuple level uncertainty, we exploit the possibility of ordering tuples on score values to incrementally maintain a state space representing possible query answers, while for attribute level uncertainty, we build on Monte-Carlo methods to find the probabilities of possible orderings by sampling from the space of score combinations.

In general, our techniques can be extended to support both types of uncertainty conjointly. Given an uncertain database with tuples having both uncertain attributes and membership probabilities, where the domains of uncertain attributes are discrete, we can adopt a tuple-level uncertainty model to capture both types of uncertainty, as we show in Section 3.1. Specifically, each tuple is inflated into a group of exclusive instances representing possible combinations of uncertain attributes' values. A top-k query in this setting is mapped to a top-k groups query, similar to our proposal in [65].

The previous approach is inapplicable when uncertain attributes have continuous domains, e.g., ranges of values. One possible approach to handle ranking queries in this setting is to extend the procedure of sampling from the space of score combinations to reflect probabilistic tuples' membership. Specifically, a tuple's possible score is included in a sample score combination with probability equal to tuple's membership probability. Exploiting other optimization opportunities, like k-dominance, is more complicated since tuples' probabilities need to be taken into account in these settings.

7.3.2 Aggregation of Partial Orders

Consider a multi-agent setting where each agent specifies preferences on a set of objects. A pair of objects can be either deterministically ordered or incomparable. Incomparability originates from different sources including unknown relationship between objects, e.g., comparing an apartment to a house, incomplete object description, e.g., apartments whose rent amounts are given in the form of intervals rather than single values, or adopting multiple criteria to order the objects. Such preferences can be often formulated as partial orders.

When multiple partial orders are available on the same set of objects, each object has a set of different rank intervals. Our goal is to collapse the rank intervals of each object into one rank interval that best conforms to the individual partial orders, and to compactly encode the resulting rank intervals into a new partial order that acts as an aggregation of the individual partial orders.

The problem of aggregating partial orders has been considered in other contexts, where an aggregation criterion (e.g., prioritization or Pareto aggregation) is used to compute an aggregate order. One problem with these approaches is non-closure, (i.e., the outcome order is not guaranteed to be a proper partial order). Our goal is different, since we would like to compute a mean partial order that has the minimum average distance to all input partial orders. Hence, there is no explicit aggregation criterion in our settings, and we restrict the result to be a proper partial order.

7.3.3 Low-Level Modifications in Query Engines

We plan at integrating our algorithms with query engines by embedding the algorithms in specialized query operators. A key problem that we need to address is defining the proper inputs and outputs for such operators, which has a direct impact on operator design and implementation. For example, under tuple level uncertainty, since our query semantics integrate scores and probabilities as two interacting ranking dimensions, one possibility is to feed the new operators with two tuple streams of the same input relation(s) ordered based on scores and probabilities. We need then to design special score aggregation methods to integrate both orders.

Regarding operators' output, the quality requirements of query output have to affect the order in which query results are propagated. Operators' implementation need to satisfy this requirement by pipelining high-quality query answers early in the output streams. The integration of the new operators with other conventional query operators would also induce several changes in the implementation of conventional query operators and their input/output interfaces.

Permissions

Part of the material in Chapters 3 and 4 has appeared in [65], copyrights ACM. The use of material in author's thesis is granted under ACM license number 2591420431237:

Mohamed A. Soliman, Ihab F. Ilyas, Kevin Chen–Chuan Chang. "Probabilistic top-k and ranking-aggregate queries", ACM Transactions on Database Systems (TODS), Vol. 33: 3, © 2008 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/1386118.1386119.

Part of the material in Chapter 3 and 5 has appeared in [63], copyrights Springer. The use of material in author's thesis is granted under Springer license number 2591410018657:

With kind permission from Springer Science+Business Media: <The VLDB Journal, 'Supporting ranking queries on uncertain and incomplete data', volume 19, year 2010, pp. 477 - 501, Mohamed A. Soliman, Ihab F. Ilyas, and Shalev Ben-David. DOI: 10.1007/s00778-009-0176-8 >.

Part of the material in Chapter 6 has appeared in [67], copyrights VLDB Endowment: Mohamed A. Soliman, Ihab F. Ilyas, Mina Saleeb. 'Building ranked mashups of unstructured sources with uncertain information.' PVLDB 3(1): 826-837 (2010). The use of material in author's thesis is granted under the following copyright notice:

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed

Bibliography

[1] Google mashup editor: http://code.google.com/gme/. 140

[2] Microsoft Popfly: http://www.popfly.com/. 140

[3] Yahoo! Pipes: http://pipes.yahoo.com/. 140

[4] Serge Abiteboul, Paris Kanellakis, and Gosta Grahne. On the Representation and Querying of Sets of Possible Worlds. *SIGMOD Rec.*, 16(3):34–48, 1987. 16, 19

[5] Periklis Andritsos, Ariel Fuxman, and Renee J. Miller. Clean Answers over Dirty Databases: A Probabilistic Approach. In *Proceedings of the 22nd ICDE*, page 30, 2006. 16, 21

[6] Tobias Anton. Xpath-wrapper induction by generalizing tree traversal patterns. *LWA 2005 - Workshopwoche der GI-Fachgruppen/Arbeitskreise*, 2005. 144, 145

[7] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, 2008. 21

[8] Lyublena Antova, Christoph Koch, and Dan Olteanu. 10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information. In *ICDE*, pages 606–615, 2007. 19, 21

[9] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proceedings of the 32nd VLDB*, pages 953–964, 2006. 1, 16, 21, 37, 55, 56

[10] Indrajit Bhattacharya, Lise Getoor, and Louis Licamele. Query-Time Entity Resolution. In *Proceedings of the 12th ACM SIGKDD*, pages 529–534, 2006. 17, 22

[11] Graham Brightwell and Peter Winkler. Counting linear extensions is #p-complete. In *STOC*, 1991. 7

[12] Russ Bubley and Martin Dyer. Faster random generation of linear extensions. In *SODA*, 1998.

[13] Chee-Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, 2006. 6

[14] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Evaluation of Probabilistic Queries over Imprecise Data in Constantly-Evolving Environments. *Inf. Syst.*, 32(1):104–130, 2007. 19, 21

[15] Reynold Cheng, Sunil Prabhakar, and Dmitri V. Kalashnikov. Querying imprecise data in moving object environments. In *ICDE*, 2003. 7

[16] Jan Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4), 2003. 6

[17] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *ICDE*, 2009. 31, 32, 34, 130, 158, 159

[18] Mary Kathryn Cowles and Bradley P. Carlin. Markov chain Monte Carlo convergence diagnostics: A comparative review. *Journal of the American Statistical Association*, 91(434), 1996. 107

[19] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001. 144

[20] Nilesh Dalvi and Dan Suciu. Efficient Query Evaluation on Probabilistic Databases. *The VLDB Journal*, 16(4):523–544, 2007. 1, 16, 19, 21, 27, 56, 58

[21] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-based approximate querying in sensor networks. *VLDB J.*, 14(4), 2005. 40

[22] Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009. 24

[23] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622, 2001. 53, 109, 110, 111, 112

[24] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003. 13, 84

[25] P. C. Fishburn. Probabilistic social choice based on simple voting comparisons. *The Review of Economic Studies*, 51(4), 1984. 113

[26] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning Probabilistic Relational Models. In *IJCAI*, 1999. 17, 22

[27] Ariel Fuxman, Elham Fazli, and Renée J. Miller. ConQuer: Efficient Management of Inconsistent Databases. In *Proceedings of the 2005 ACM SIGMOD*, pages 155–166, 2005. 19, 21

[28] Andrew Gelman and Donald B. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4), 1992. 106, 107

[29] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity Estimation using Probabilistic Models. *SIGMOD Rec.*, 30(2):461–472, 2001. 27, 28

[30] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans.*, 4(2):100–107, 1968. 68, 69, 72, 74

[31] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1), 1970. 30, 31, 106

[32] Mauricio A. Hernandez and Salvatore J. Stolfo. Real-world Data is Dirty: Data Cleansing and the Merge/Purge Problem. *J. Data Mining and Knowledge Discovery*, 2(1):9–37, 1998. 19

[33] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *SIGMOD*, 2008. 31, 33, 34, 138

[34] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting Top-K Join Queries in Relational Databases. In *Proceedings of the 29th VLDB*, pages 754–765, 2003. 55, 61

[35] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. *VLDB Journal*, 13(3):207–221, 2004. 14, 128

[36] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008. 12, 131

[37] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-Aware Query Optimization. In *Proceedings of the 2004 ACM SIGMOD*, pages 203–214, 2004. 131, 148

[38] Tomasz Imieliński and Jr. Witold Lipski. Incomplete Information in Relational Databases. *J. ACM*, 31(4):761–791, 1984. 16, 18

[39] Michael D. Intriligator. A probabilistic model of social choice. *The Review of Economic Studies*, 40(4), 1973. 113

[40] Mark Jerrum and Alistair Sinclair. *The Markov chain Monte-Carlo method: an approach to approximate counting and integration.* PWS Publishing Co., Boston, MA, USA, 1997. 29, 30

[41] Claire Kenyon-Mathieu and Warren Schudy. How to rank with few errors. In *STOC*, 2007. 110, 112

[42] Nicholas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *IJCAI (1)*, 1997. 143, 144, 145

[43] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov chains and mixing times.* American Mathematical Society, 2006. 29, 30

[44] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting Ad-hoc Ranking Aggregates. In *Proceedings of the 2006 ACM SIGMOD*, pages 61–72, 2006. 55

[45] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *Proceedings of the 2005 ACM SIGMOD*, pages 131–142, 2005. 61, 85, 134

[46] Jian Li and Amol Deshpande. Consensus answers for queries over probabilistic databases. In *PODS*, 2009. 33, 38

[47] Jian Li and Amol Deshpande. Ranking continuous probabilistic datasets. In *VLDB*, 2010. 33, 34

[48] Jian Li, Barna Saha, and Amol Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1), 2009. 32, 33

[49] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of the 2003 ACM SIGMOD*, pages 491–502, 2003. 19

[50] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1997. 29

[51] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *Proceedings of the 27th VLDB*, pages 281–290, 2001. 128, 131

[52] Dianne P. O'Leary. Multidimensional integration: Partition and conquer. *Computing in Science and Engineering*, 6(6), 2004. 29

[53] R-project. The R Project for Statistical Computing: www.r-project.org. 86

[54] Christopher Ré, Nilesh N. Dalvi, and Dan Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In *Proceedings of the 23rd ICDE*, pages 886–895, 2007. 21, 29

[55] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62, 2006. 24

[56] Anish Das Sarma, Omar Benjelloun, Alon Halevy, and Jennifer Widom. Working Models for Uncertain Data. In *Proceedings of the 22nd ICDE*, page 7, 2006. 1, 16, 19, 21, 37, 56

[57] Karl Schnaitter and Neoklis Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008. 131

[58] Prithviraj Sen and Amol Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *Proceedings of the 23rd ICDE*, pages 596–605, 2007. 23, 27, 34, 39, 56

[59] Pierre Senellart, Avin Mittal, Daniel Muschick, Rémi Gilleron, and Marc Tommasi. Automatic wrapper induction from hidden-web sources with domain knowledge. In *WIDM*, 2008. 144

[60] David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, and Ashutosh Singh. Damia: data mashups for intranet applications. In *SIGMOD*, 2008. 140

[61] David E. Simmen, Frederick Reiss, Yunyao Li, and Suresh Thalamati. Enabling enterprise mashups over unstructured text feeds with infosphere mashuphub and systemt. In *SIGMOD*, 2009. 140

[62] Mohamed A. Soliman and Ihab F. Ilyas. Ranking with uncertain scores. In *Proceedings of the 25th ICDE*, 2009. 33, 34, 35, 92, 130

[63] Mohamed A. Soliman, Ihab F. Ilyas, and Shalev Ben-David. Supporting ranking queries on uncertain and incomplete data. *The VLDB Journal*, 19(4), 2010. 33, 34, 35, 92, 163

[64] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin C. Chang. Top-k Query Processing in Uncertain Databases. In *Proceedings of the 23rd ICDE*, pages 896–905, 2007. 31, 34, 35, 54, 130

[65] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin C. Chang. Probabilistic Top-k and Ranking-Aggregate Queries. *ACM Trans. Database Syst.*, 33(3):1–54, 2008. 31, 35, 54, 57, 58, 161, 163

[66] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. URank: Formulation and Efficient Evaluation of Top-k Queries in Uncertain Databases. In *Proceedings of the 2007 ACM SIGMOD*, pages 1082–1084, 2007. 56, 90

[67] Mohamed A. Soliman, Ihab F. Ilyas, and Mina Saleeb. Building ranked mashups of unstructured sources with uncertain information. *PVLDB*, 3(1):826–837, 2010. 33, 127, 163

[68] Mohamed A. Soliman, Mina Saleeb, and Ihab F. Ilyas. Mashrank: Towards uncertainty-aware and rank-aware mashups. In *ICDE*, 2010. 140

[69] Yufei Tao, Xiaokui Xiao, and Jian Pei. Efficient skyline and top-k retrieval in subspaces. *TKDE*, 19(8), 2007. 6

[70] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. 27

[71] P. van Acker. Transitivity revisited. *Ann. Oper. Res.*, 23(1-4), 1990. 113

[72] Michael L. Wick, Andrew McCallum, and Gerome Miklau. Scalable probabilistic databases with factor graphs and mcmc. *PVLDB*, 2010. 23, 39

[73] Jennifer Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Proceedings of the Second Biennial CIDR*, pages 262–276, 2005. 16, 21, 55

[74] Garrett Wolf, Hemal Khatri, Bhaumik Chokshi, Jianchun Fan, Yi Chen, and Subbarao Kambhampati. Query processing over incomplete autonomous databases. In *VLDB*, 2007. 7, 40

[75] Xintao Wu and Daniel Barbará. Learning missing values from summary constraints. *SIGKDD Explorations*, 4(1), 2002. 7

[76] Xi Zhang and Jan Chomicki. On the semantics and evaluation of top-k queries in probabilistic databases. In *ICDE Workshops*, 2008. 31, 33, 130, 138, 140