CERIAS Tech Report 2011-28
Efficient Query Processing for Uncertain Data
by Yinian Qi
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance &

This is to certify that the thesis/dissertation prepared

By Yinian Qi

Entitled
Efficient Query Processing for Uncertain Data

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Sunil Prabhakar Chair	Mikhail Atallah
Jennifer Neville	Dongyan Xu

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Sunil Prabhakar

Approved by: William J. Gorman 07/22/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Efficient Query Processing for Uncertain Data

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Yinian Qi

Printed Name and Signature of Candidate

07/11/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

EFFICIENT QUERY PROCESSING FOR UNCERTAIN DATA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Yinian Qi

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2011

Purdue University

West Lafayette, Indiana

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

To my dear mom and dad, for being my source of love, support and inspiration.

ACKNOWLEDGMENTS

I owe my entire Ph.D. research to the support of two major professors I am honored to have worked with: Prof. Sunil Prabhakar and Prof. Mike Atallah. Prof. Prabhakar, my advisor, taught me to be an independent researcher while gave me great advice when I most needed it. I deeply appreciate his support during my Ph.D. years, his openness to me as an advisor, and his valuable advice on how to write good papers and present research work. Prof. Atallah, an amazing mentor to me both in academia and in life, as well as the best teacher I have ever had (whose algorithms and cryptography classes were just fascinating), has always inspired me with his passion for research and has constantly encouraged me to aspire and accomplish.

Prof. Chris Clifton and Prof. Jennifer Neville, who were on the committee of my prelim exam, gave me insightful comments and suggestions on my research. I enjoyed taking Jennifer's data mining class, and visiting Prof. Clifton's family along with other students in his wonderful home around Christmas time was undoubtedly one of the most fun memories I had at Purdue. Prof. Sonia Fahmy, whom I have neither worked with nor taken a class with, always showed her sincere support and care for me whenever we bumped into each other in the department. I would like to thank Prof. Luo Si for serving on my prelim committee, and Prof. Dongyan Xu for kindly agreeing to serve as an external member on my final exam committee when I asked him at the last minute. His operating system class, both engaging and interesting, was one of my favorite class experiences.

My dissertation could not have been written without the help and friendship from my fellow students in the database lab. I would like to especially thank Rohit Jain and Chris Mayfield, for discussing research ideas with me and for sharing ups and downs with me in both research and life. I am also grateful to Hoda Eldardiry, who offered to help me practice my prelim talk. I am extremely fortunate to have many

friends who are willing to take their time to listen to me, support and encourage me, as well as give me great suggestions throughout my years in graduate school: Emil Stefanov, Meghana Chitale, Tiancheng Li, Hao Yuan, Dongxin Zou, Xiayu Rao, Feng Yan, Rongjing Xiang, Vasil Denchev, Umang Sharan, Ashish Kundu, Nwokedi Idika are just a few among many others.

My appreciation also goes to all the staff in Computer Science Department. They provided me with useful information and facilities, and are always available for assistance. I would like to thank Dr. William J. Gorman for his dedication to students and to the department, Renate Mallus for being such a wonderful resource, Ron Constongia for being so responsive and patient to every request I had, Nicole Piegza for helping me to schedule many important meetings with Prof. Prabhakar on my dissertation work, and the late graduate secretary Amy Ingram, who used to bring me so much joy when I stopped by the graduate office.

Finally, I owe everything to my parents and my extended family. My parents have always believed in academic excellence, which eventually lead to my Ph.D. journey in the US. They have helped me to reach my goals and have supported me along the way. I also want to thank my extended family for their love and care, and for all the help they have lent me, especially my dear aunt and uncle, both chemistry professors at Fudan University in Shanghai, who treated me like their own daughter when I went to Fudan for college. My special thanks are to my dearest grandma, who is the most wise and capable woman I have ever seen. The love from my family is the most precious thing that I cherish. I hope I always make them proud.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Qi, Yinian. Ph.D., Purdue University, August 2011. Efficient Query Processing for Uncertain Data . Major Professor: Sunil Prabhakar.

Applications with uncertain data pose many challenges for data management and query processing. This dissertation advances the state of the art for efficient query processing over uncertain data. We study three types of probabilistic queries: nearest-neighbor queries, skyline queries and the general select-project-join queries, all of which could leverage a probability threshold for pruning such that only results that satisfy the query with probabilities over the given threshold are returned. For nearest-neighbor queries, we design novel indexes and data structures to monitor the pruning status and uncover pruning opportunities. For skyline queries, we propose two filtering schemes to quickly identify interesting instances whose skyline probabilities are over the threshold: i) by bounding an instance's skyline probability, and ii) by comparing the instance with others based on dominance relationship. In applications of skyline analysis where "thresholding" is not desirable, we propose the problem of computing all skyline probabilities and for the first time present two worst-case sub-quadratic algorithms to solve the problem. We further give an efficient algorithm for the online version of the problem. Finally, we study the general select-project-join (SPJ) queries under the Orion uncertainty model [1] and propose optimization rules to leverage the threshold for early pruning of unqualified tuples. We also extend our study to SPJ queries with duplicate elimination. We adopt a general tuple uncertainty model for this case and design new techniques for handling duplicate elimination. Our experiments on various data sets show that our techniques are both effective and efficient.

1. INTRODUCTION

Uncertain data arise in many important applications, such as sensor networks and location-based applications [2], where data can take different values with probabilities due to measurement errors. Other applications include scientific data management, data integration, and data cleaning, where uncertainty is caused by limited resource to analyze and understand data. Due to the importance of uncertain data for a large number of applications, there has been significant recent interest in database support for uncertain data. Existing work in this area includes new models for uncertain data, prototype implementations, and efficient query processing algorithms. In Section 1.1 below, we first give an overview of research in managing uncertain data before introducing our work on querying uncertain data in Section 1.2.

1.1 Managing Uncertain Data

Many research efforts have been dedicated to modeling uncertain data and building prototypes of the models [1, 3–9]. Two major models for uncertainty exist: Tuple uncertainty models and attribute uncertainty models. For tuple uncertainty models [3, 4], each tuple is associated with a probability of its existence. For attribute uncertainty models, a tuple always exists, but there may be one or more uncertain attributes in the tuple with *probability distribution functions* (pdfs) associated with them. Recently, the Orion database model is proposed for supporting pdf attributes. It is capable of handling both attribute and tuple uncertainty [1], where attributes can have pdfs and tuples can have existence probabilities.

A big challenge in managing uncertain data is to ensure the correctness of query processing given dependencies between uncertain data that are either inherent in data (e.g., mutual exclusivity between a set of tuples for tuple uncertainty models

and joint distribution between a set of attributes for attribute uncertainty models) or arise during query evaluation (e.g., joins). For example, MystiQ [4] queries uncertain data using *safe plans* [10], which attempts to choose an alternative query plan that results in correct confidence computation with joins and duplicate elimination of projections. However, safe plans are not always the most efficient plans and may not even exist for some queries. To overcome these drawbacks, Trio [3, 11] uses *lineage* to explicitly capture data dependencies and efficiently compute confidence. Hence their query evaluation is not restricted to safe plans and is separated from confidence computation. Other tools to capture dependencies are also proposed, such as factor tables [7], world tables [6] and history [1]. While many uncertain data models assume tuple independence or tuple dependencies, Orion [1] and MayBMS [6] are able to capture uncertainty and dependencies at attribute level. In particular, Orion handles both intra-tuple dependencies (captured by *dependency sets*) and inter-tuple dependencies (captured by *history*). It is also the first model to handle continuous uncertainty as seen in sensor networks. Recently, another model that is capable of handling continuous uncertainty is proposed in [9], which uses c-tables [12].

We first introduce the well-received *possible worlds semantics* for probabilistic data. We then give an overview of a number of uncertain data models that are widely accepted by the uncertain database research community. These uncertainty models are categorized into tuple uncertainty models and attribute uncertainty models. Finally, we introduce the Orion model [1] that captures uncertainty at both tuple level and attribute level. Our work in query processing for uncertain data is based on these uncertainty models.

1.1.1 Possible Worlds Semantics

In order to provide meaningful semantics for queries over uncertain data, a large body of recent work has adopted the well-known *Possible Worlds Semantics* [10] (PWS) over probabilistic data. As with traditional data, efficient execution is neces-

	A	B	probability
t_1	1	1	0.8
t_2	2	1	0.5
t_3	2	3	0.4

possible world	probability
$W_1 = t_1$	0.24
$W_2 = t_2$	0.06
$W_3 = t_3$	0.04
$W_4 = t_1, t_2$	0.24
$W_5 = t_2, t_3$	0.04
$W_6 = t_1, t_3$	0.16
$W_7 = t_1, t_2, t_3$	0.16
$W_8 = \emptyset \leftarrow$	0.06

Fig. 1.1.: A simple uncertain database and its possible worlds

sary for ensuring the viability of uncertain data management systems. In fact, due to the complications of ensuring correct results (with respect to PWS), and the need for CPU-intensive operations over probability distributions, it is even more critical and challenging for uncertain data.

Fig. 1.1 shows an example uncertain table with three tuples (t_1 to t_3), each associated with a probability There are two attributes A and B in the table. This simple example illustrates the *independent tuples model* [10], in which all tuples are independent from each other and each tuple is associated with a probability that it exists (i.e. *existence probability*). Due to the uncertainty of tuple existence, there are an exponential number of *possible worlds* with regards to the total number of tuples in the database. All possible worlds that can be generated from the three tuples are also shown in Fig. 1.1. The probability associated with each possible world W, denoted as $Pr(W)$, is computed as the product of the probabilities of tuples that appear in W, i.e., $Pr(W) = \prod_{t \in W} Pr(t)$. For example, $Pr(W_5) = Pr(t_2) * Pr(t_3) * (1 - Pr(t_1)) = 0.5 * 0.4 * 0.2 = 0.04$. This is because all tuples are independent from each other. Furthermore, we have$\Sigma_{W_i} Pr(W_i) = 1$. Let the table be T.

1.1.2 Tuple Uncertainty Model

For tuple uncertainty models, attributes in a tuple have exact values. The only uncertainty comes from the existence of tuples. Two most popular tuple uncertainty models are the independent tuples model and the x-tuple model.

Independent Tuples Model

The independent tuples model is the simplest model for uncertain data. As discussed earlier in Section 1.1.1, it models uncertain data as independent tuples associated with existence probabilities. There is no dependency between data in the base tables. Fig. 1.1 shows an example of the independent tuples model. This model is adopted in many papers such as [10, 11].

X-Tuple Model

The x-tuple model builds upon the independent tuples model in that each tuple is still associated with an existence probability. However, now tuples are no longer independent from each other: Some tuples are mutually exclusive among each other, i.e., only one tuple from that set (called an *x-tuple*) can exist at any time. Let the probabilities of tuples within the x-tuple add up to p: then $p \leq 1$ always holds. If $p = 1$, there must exist a "representative" tuple for the x-tuple; otherwise, there is $1 - p$ probability that no tuple from the x-tuple exists. If we model an x-tuple as an uncertain object and each tuple within the x-tuple as the object's instances, then each object has a set of values (i.e., instances) that it can take with possibly different probabilities and only one value can be taken at any time. If $p < 1$, we refer to $1 - p$ as the *missing probability* of the object – the probability that the object does not exist. This model has been used in many areas of uncertain data research, such as handling tuple dependencies [7, 11], ranking queries [13–15], skyline queries [16–19], etc.

Suppose t_2 and t_3 are mutually exclusive in Fig. 1.1.2, i.e., t_2 and t_3 belong to the same x-tuple. Then the possible worlds generated from the uncertain table subject to the above mutual exclusivity requirement are all possible worlds in Fig. 1.1.2 except W_5 and W_7.

General Tuple Uncertainty Model

While the independent tuple model assumes independence between tuples and the x-tuple model groups tuples that are mutually exclusive, we propose a general tuple uncertainty model where no assumptions are made about dependencies between tuples. In this general model, each attributes in each individual tuple has a certain value and the whole tuple is associated with a probability of its existence. The dependencies between tuples are specified using a chosen mechanism, for example, history in Orion [1], lineage in Trio [3] and world tables in MayBMS [6]. More details about this model will be discussed later in Section 5.1.1.

1.1.3 Attribute Uncertainty Model

While attributes in tuple uncertainty models have certain values, attributes in attribute uncertainty model can be uncertain. The uncertain attributes are typically associated with probability distribution functions (pdfs).

Uncertainty Region Model

One common attribute uncertainty model is the uncertainty region model, where the location of an object is associated with an uncertainty region of all possible locations. The pdf of the location is known within the uncertainty region. The cumulative probability p of an object's location within its uncertainty region is always less than or equal to 1. If $p < 1$, then the uncertainty region model also exhibits uncertainty at tuple level, e.g., the existence of the tuple (i.e., the object) itself

is uncertain, in addition to the uncertainty of its location. This model is widely adopted in location based applications, such as nearest-neighbor queries [2, 20–22], reverse nearest-neighbor queries [23, 24] and range queries [25, 26]. Generally the objects are assumed to be independent from each other.

1.1.4 Orion Model

In this section, we introduce the Orion model proposed in [1], a much more complex model compared with the simple working models introduced earlier. This is the model that we choose for the threshold SPJ query optimization problem [27] that we will address in Chapter 4. Under this model, uncertainty is represented directly in a tuple using discrete or continuous pdfs. Dependencies inherent in the data are captured in terms of joint distributions. A key aspect of the model is that it does not enumerate all possible values for an uncertain attribute or a tuple (as is the case for many other leading models). This enables the model to directly capture infinite possibilities (e.g., a Gaussian probability distribution) without necessarily resorting to an approximate representation, i.e., it handles continuous uncertainty naturally.

Example 1.1.1 *Consider an application where the speed of cars on a highway is monitored. Due to errors in measurement, the speed sensors report a range over which the actual speed is uniformly distributed. Based on the engine noise, the make and model of the car are inferred by classification programs. Often these inferences are only able to narrow down the make and model to a few options with associated confidences. For example, for a given vehicle, the make and model may be either Honda Civic, or Toyota Corolla. Note that these two fields are jointly distributed, i.e., we cannot have arbitrary combinations like Honda Corolla. This information is to be stored in a database with the following attributes:* Highway*,* speed*,* Make*, and* Model*. Table 1.1 shows the speed information for three cars stored using the Orion uncertainty model which is discussed below.*

Table 1.1: Car speed on highways

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

Under the Orion model, an uncertain relation T is represented using a *probabilistic schema*, $(\ \Sigma_T, \Delta_T)$.$\Sigma_{\ T}$ is the normal relational schema (attribute names and domain types). The set of possible domains is expanded to include new data types. These data types represent continuous uncertainty (either as a symbolic representation such as a Gaussian, or a histogram), ordered discrete (e.g., integer values) and categorical or unordered discrete (e.g., colors).$\Delta_{\ T}$ captures *dependency information*. It is a partitioning of the uncertain attributes of T. Each partition, called a *dependency set*, declares that the attributes in that partition are jointly distributed (i.e., correlated). An uncertain attribute that is independent from all the other attributes forms its own singleton dependency set. For our car example in Table 1.1, $\Delta_T = \{\{Speed\}, \{Make, Model\}\}$.

In the standard relational model, a tuple is a collection of exact values (one for each attribute in the schema). Under the Orion model, a tuple is a collection of exact values (one for each certain attribute, if any) and probability distributions (one for each dependency set, if any). For example, the first tuple in Table 1.1 consists of one certain value 101 for *Highway*, and two pdfs: Uniform(65,75) for *Speed*, and { ('Honda', 'Civic'):0.4, ('Toyota', 'Co-rolla'):0.2 } for $\{Make, Model\}$. This tuple represents a car on Highway 101 traveling with a speed that is uniformly distributed

between 65 and 75 *mph* and is either a Honda Civic with probability 0.4, or a Toyota Corolla with probability 0.2.

From this example we can see that the model allows for missing probabilities – i.e., the sum of probability values for any distribution can be less than 1 indicating partial probabilities. [1] In general each pdf may be multi-dimensional over any combination of uncertain domains. We define the probability of a dependency set S in a tuple t, denoted as $Pr(t.S)$, as the cumulative probability mass of the pdf defined on $t.S$. The overall (tuple) probability of the tuple t, denoted as $Pr(t)$, is then the product of the cumulative probability mass of each of its dependency sets, i.e. $Pr(t) = \prod_{S \in \Delta_T} Pr(t.S)$. Thus, for the first tuple in Table 1.1, the overall tuple probability is $1 \times 0.6 = 0.6$.

In addition to representation, a model must specify how queries are processed correctly (with respect to PWS). The major challenge for correct evaluation of probabilities is caused by dependencies among derived data [7]. The model explicitly tracks the original pdf from which each resulting pdf in a result tuple is derived. Thus for each tuple, the model stores a *history* Λ that handles inter-tuple dependencies that result from prior database operations. History captures dependencies between dependency sets of tuples. The function Λ maps each pdf of a dependency set $t.S$ in tuple t, to a set of pdfs that are its ancestors, i.e., from which the pdf of $t.S$ is derived. Only the top-level ancestors are stored, i.e., the base pdfs inserted in the database by the user (base tuples are assumed to be independent from each other). Two pdfs are called *historically independent* if their histories do not overlap, otherwise they are *historically dependent*.

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

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

`floor`(f, I) takes an input pdf f and reduces the probability to zero over all points in region I. It produces a partial pdf f' such that values of $f'(x) = 0$ whenever $x \in I$ and $f'(x) = f(x)$ otherwise. This `floor` operation corresponds to a selection predicate. The values in I are those which do not pass the selection criteria and hence do not exist in the resulting pdf. Multiple `floor` operations can be successively applied over a pdf in any order and the result would be `floor`$(f, I_1 \cup ...I_k)$ regardless of the order in which they are applied.

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

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

1.2 Querying Uncertain Data

Querying uncertain data has gained increasing popularity over recent years. Much work focuses on solving specific problems for uncertain data, such as the nearest-neighbor problem [2, 20, 21, 28], reverse nearest-neighbor problem [23, 24], indexing [29–31], ranking [13–15, 32–35], range queries [25, 26, 36, 37], skyline queries [19, 38], join processing [39], etc. Among these, the probabilistic threshold query is one of the most common queries over probabilistic data, which returns results satisfying the query with probabilities equal to or greater than a given threshold. Threshold queries are useful for many applications where results with low probabilities are less relevant. For example, the probability of a result is indicative of our confidence in the result being true [10]. Thus low probability results are not of interest in many cases.

Optimizations can be employed to leverage the threshold for pruning during the query evaluation so that all results that have no hope of meeting the threshold can be discarded as early as possible. For example, [20] proposed the concept of the "constrained probabilistic nearest-neighbor query" with a probability threshold and an error tolerance to save expensive computations of the exact nearest-neighbor probabilities. Other examples of threshold queries include the probabilistic threshold approach to ranking queries [14], range queries [29] and skyline queries [19, 38].

Below we first give an overview on query processing of uncertain data, then introduce three types of probabilistic queries that we have studied – probabilistic nearest-neighbor queries, probabilistic skyline queries and finally, general threshold queries for selection, projection and join (threshold SPJ queries). All three types of queries can leverage a probability threshold for efficiently pruning the search space. We then summarize our contributions to efficient evaluation of these queries in Section 1.2.5.

1.2.1 Overview

The main challenges in query processing of uncertain data lie in the following two aspects: First, we need to ensure the correctness of query results in the presence of

complicated dependencies between data. This is handled mainly by the uncertain data models, as mentioned in Section 1.1. Second, due to the probabilistic nature of uncertain data, the search space of queries is much larger compared with traditional data with no uncertainty: i) there are more potential results; ii) more data has to be considered in computing the results, especially in confidence computation, where confidence is the probability of a result satisfying the query; and iii) under the possible worlds semantics, if we take the naïve approach to querying uncertain data, there can be an exponential number of possible worlds in which the query needs to be evaluated. Specifically, given a query performed on a set of uncertain tables, the naïve approach is to first enumerate all possible worlds of the base tables, perform the query in every single possible world (treated as a set of certain tables) to obtain the results, then summarize the possible worlds to obtain the final query results in an uncertain table. This approach can be very expensive due to the large number of possible worlds, as shown in Fig. 1.1. Therefore, the more efficient approach (also the common approach) to query processing with uncertain data is to obtain the query results directly from the original tables without enumerating their possible worlds but still ensure that the results are the same as those obtained through the naïve approach.

Our contributions to querying uncertain data focus on its second aspect: improving the efficiency. We design efficient query processing algorithms that quickly reduce the search space by: i) pruning unqualified results as early as possible; ii) bounding the confidence instead of computing the exact value for pruning opportunities (the former has much more efficient algorithms than the latter); and iii) taking the efficient approach instead of the naïve approach to uncertain data query evaluation such that query results are computed directly from the original uncertain tables without explicitly considering their possible worlds.

Below we review three kinds of probabilistic queries that we worked on: probabilistic nearest-neighbor queries (Section 1.2.2), probabilistic skyline queries (Section 1.2.3) and threshold select-project-join (SPJ) queries (Section 1.2.4). Our contributions to each kind of queries are summarized in Section 1.2.5.

1.2.2 Probabilistic Nearest-neighbor Queries

The nearest-neighbor (NN) query is one of the most common database queries that finds the nearest object to a query object given some distance function. Many algorithms have been proposed for NN queries [40–42] where the value of data is certain. In probabilistic data setting, NN queries need to be re-evaluated. Take the location-based data for example. Suppose all data objects are in 2-dimensional space. The exact location of an object is unknown. However, each object is associated with a region of its possible locations and the pdf of the object's location within the region is known. Since each object has a probability (maybe 0) to be NN to a query object, we have to take probabilities into account when answering NN queries: We can either return all objects with a non-zero probability to be NN or return all objects with the NN probability greater than some threshold. We call the former *probabilistic nearest-neighbor (PNN) queries* and the latter *probabilistic nearest-neighbor threshold (PNNT) queries*. Several papers have studied the NN problem with uncertain data. For example, [2] proposed an algorithm for answering PNN queries. The algorithm returns all objects along with their non-zero NN probabilities, which requires a large number of expensive computations of the exact NN probabilities. However, most of the time we are only interested in objects with a relatively large probability to be NN, hence a probability threshold can be specified for the query to only return objects with NN probabilities that meet the threshold (i.e., PNNT queries). For such queries, the threshold can be leveraged to prune objects that cannot satisfy the probability requirement. [20] proposed the constrained probabilistic nearest-neighbor query (C-PNN) with both threshold (P) and tolerance ( ) constraints, which is equivalent to having a single threshold $P -\leftarrow$.

1.2.3 Probabilistic Skyline Queries

Skyline queries are widely used in multi-criteria decision making, where a choice that scores high in one criterion may score low in another (e.g., a hotel very close

to the beach but very expensive). The query returns all data points that are not dominated by any other point in a data set, where a point p_1 *dominates* another point p_2 if p_1 is no worse than p_2 in all dimensions and better than p_2 in at least one dimension.

Efficient algorithms are proposed to answer a variety of skyline queries [43–47], all of which deal with traditional data where no uncertainty is involved. While significant research efforts have been dedicated to modeling, managing and querying uncertain data, advanced analysis of uncertain data is still in its early stages. Recently skyline queries with uncertain data have also been studied [16, 19, 48]. [16] first introduced *probabilistic skyline queries* that answer skyline queries for data with discrete uncertainty, i.e., each uncertain object is associated with a set of instances and corresponding probabilities to take those particular instances. Instances of the same object are mutually exclusive, i.e., at most one can exist at a time. For probabilistic skyline queries, only objects with skyline probabilities greater than or equal to the threshold are returned. The skyline probability of an instance that belongs to an object is the probability that this instance occurs and is not dominated by any occurring instance of another object. The skyline probability of an object is the sum of the skyline probabilities of all its instances (because the instances are mutually exclusive).

1.2.4 Threshold SPJ Queries

Many algorithms for query processing with uncertain data leverages a probability threshold for efficient evaluation, such as probabilistic nearest-neighbor queries and skyline queries introduced earlier. Such threshold queries represent an important class of queries over uncertain data that return only those query results whose probabilities meet a given threshold. However, most of the algorithms for threshold queries are limited to a single query and do not address complex query optimization (such as an arbitrary SQL query). [8] aims at optimizing some SQL queries,

but only works for tuple-independent probabilistic databases and is focused on exact confidence computation of the query results.

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

One of the major challenges in answering threshold SPJ queries is ensuring the correctness of query results. Due to the probabilistic nature of the data, results (and base data items too) often have dependencies that must not be ignored in order to ensure correct computation of result probabilities. Consequently, the question of how a threshold query for uncertain data can be optimized is not obvious.

1.2.5 Summary of Contributions

We design algorithms to efficiently process probabilistic nearest-neighbor queries [22], skyline queries [17,18,49] and threshold SPJ queries [27] for uncertain data. Our main contributions are summarized below:

Probabilistic nearest-neighbor Queries

- We generalize the PNNT query (see Section 1.2.2) by allowing objects to have missing probabilities (i.e. the cumulative probability of an object's location in its uncertain region may be less than 1). This problem is not considered in any previous PNN paper and brings significant challenges to the design of new pruning algorithms.

- We propose an augmented R-tree index with additional probabilistic information to facilitate pruning as well as global data structures for maintaining the current pruning status.

- We propose a PNNT query processing algorithm and experimentally verify the efficiency of the algorithm in terms of pruning capabilities.

Probabilistic Skyline Queries

- We propose an instance-level probabilistic skyline problem that provides fine-grained (i.e., instance-level) information about probabilistic skylines.

- For situations in which users are only interested in instances with skyline probabilities over a certain threshold, we present two filtering schemes for efficient query processing.

- For situations where "thresholding" is not desirable – low probability events cannot be ignored when their consequences are significant, it is necessary to compute skyline probabilities of all instances. We provide the first algorithm for this problem whose worst-case time complexity is sub-quadratic, as a result of the careful balancing between a space partitioning algorithm and the existing dominance counting algorithm [50, 51]. We further propose a new algorithm to improve this sub-quadratic result, and design an efficient algorithm to compute the skyline probabilities "on the fly": Given a set of uncertain objects and a query point that is unknown until the query time, return the probability that the query point is not dominated by any instance of the given set.

Threshold SPJ Queries Under the Orion Model

- We present the first work to address the important problem of optimizing arbitrary threshold select-project-join (SPJ) queries over uncertain data under the Orion model (see Section 1.1.4).

- We formalize the notion of threshold SPJ queries using a new threshold operator, τ_θ, as an addition to the set of standard relational algebra operators.

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

- We experimentally validate (using real and synthetic data) the effectiveness of our optimization rules in the Orion uncertain database.

- We further propose the idea of increasing the threshold as we push it down the query plan for pruning, in order to quickly prune away tuples that pass the original threshold during an early stage of query evaluation but fail to do so at the end of the query.

Threshold SPJ Queries With Duplicate Elimination

- We study the optimization of threshold SPJ queries when duplicate elimination is enabled under the general tuple uncertainty model (see Section 1.1.2). We design new optimization rules for this model, which are applicable even if the dependencies between tuples are not known at the time of the query.

- We propose pruning techniques and algorithms to efficiently process queries with duplicate elimination. We also design new techniques to improve pruning for queries with joins when the tables to be joined are independent from each other.

- We give an empirical study on the performance of different optimization algorithms on various data sets and show that our techniques are both effective and efficient.

We will discuss our algorithms and approaches to the above problems in details in Chapter 2 - 5.

2. PROBABILISTIC NEAREST NEIGHBOR QUERIES

2.1 Problem Definition

Assuming that we have a database of objects with uncertain attributes as continuous random variables associated with pdfs (i.e., the uncertain data model presented in Section 1.1.3), we give two formal problem definitions for NN queries of such objects.

Definition 2.1.1 ***Probabilistic Nearest Neighbor (PNN) Query****: Given a query point q and a set of objects with uncertain attributes and their corresponding pdfs, a PNN query returns the probability $P_{nn}(U)$ that uncertain object U is NN to q for each object U.*

For PNN queries, the probability for each object to be NN must be computed unless there is evidence that the object cannot be NN (i.e., the NN probability is 0). This implies a huge number of computations if the number of objects is huge. Moreover, the computation of the probability itself is very expensive, which depends on many other objects whose uncertain regions overlap with its own [2]. The exact probability computation can involve integrations over multiple subregions that may have arbitrary pdfs, resulting in a high computational cost. However, objects having a small probability to be NN are generally less important than those with a high probability. For many applications, it is only necessary to retrieve objects with the NN probability exceeding a given threshold. The formal definition of such queries is given below.

Definition 2.1.2 ***Probabilistic Nearest Neighbor Threshold (PNNT) Query****: Given a query point q, a threshold τ and a set of objects with uncertain attributes and their pdfs, the PNNT query returns every object U with $P_{nn}(U) > \tau$.*

Since we are only concerned about object U with $P_{nn}(U) > \tau$ in PNNT queries, we do not need to compute the exact P_{nn} of the object if we can prove the probability cannot exceed τ. In this case, we can safely prune away those objects, hence reduce the computational cost.

Note that in our PNNT queries, we do not require that the probabilities of an object's region sum up to 1 (in other words, the pdf can be a partial pdf). Suppose the sum is p, then $1-p$ is the missing probability that the object does not exist at all. This is a more general case. We need to consider more when pruning objects: Unless at least one object closer to q is sure to exist, an object that is far from q still has a non-zero probability to be NN, thus cannot be pruned away immediately as in [20].

2.2 Augmented R-Tree Index

In this section, we describe our new R-tree index for the PNNT problem defined in the previous section. We propose three types of augmentation to the normal R-tree in order to answer the PNNT queries both effectively and efficiently. The following information is added to the entries in an R-tree to facilitate query processing: Absence probability (AP), maximal probability (MP) and the absence probability bounds (AP-bounds). We first introduce each augmentation separately, then show how to incorporate all of them into our index structure. In the rest of the chapter, we use τ to denote the PNNT query threshold.

2.2.1 Absence Probability (AP)

Definition 2.2.1 ***Pruning Circle****: A circle $C_{q,r}$ centered at query point q with a radius r is called a* pruning circle *if for every object U lying outside $C_{q,r}$ we have $P_{nn}(U) < \tau$.*

The reason why $C_{q,r}$ is called a pruning circle is that given $C_{q,r}$, we can safely prune away all objects lying outside it when processing PNNT queries. Our goal is to shrink

the pruning circle as much as possible so that all objects outside it can be pruned away immediately, leaving only a small portion of objects to be further examined as NN candidates. Next we introduce absence probability for our augmented R-tree index:

Definition 2.2.2 ***Absence Probability*** *AP: Given a Minimum Bounding Rectangle (MBR) M in an R-tree, $AP(M)$ is defined as the probability that none of the objects contained in M is present. Likewise, for a circle C, $AP(C)$ is the probability that no object in C is present.*

Moreover, we define *maximum distance* $d_{max}(q, M)$ from query point q to MBR M to be the maximum distance of all distances from q to M and similarly *minimum distance* $d_{min}(q, M)$ is the minimum distance of all distances from q to M. We propose the following theorem that leverages $AP(M)$ and $d_{max}(q, M)$ to prune away MBRs whose objects cannot be NN candidates.

Theorem 2.2.1 *If $AP(M_i) < \tau$ for MBR M_i, then a circle $C_{q,r}$ centered at query point q with radius $r = d_{max}(q, M_i)$ is a pruning circle.*

Proof Since there may be objects inside $C_{q,r}$ that are contained in MBRs other than M_i (denoted as M_j, as shown in Fig. 2.1), we can infer that

$$AP(C_{q,r}) \leq AP(M_i) \cdot \left(\prod_{M_j, j \neq i} AP(M_j) \right) \leq AP(M_i) < \tau$$

For any object U in any MBR M_k outside $C_{q,r}$ ($d_{min}(q, M_k) \geq r$) to be NN to q, there should be no object inside $C_{q,r}$, i.e., $P_{nn}(U) \leq AP(C_{q,r}) < \tau$. From Definition 2.2.1 we conclude that $C_{q,r}$ is a pruning circle. ■

Fig. 2.1 illustrates the pruning circle $C_{q,r}$ when $AP(M_i) < \tau$. The MBR M_k outside the circle thus can be pruned away immediately. This pruning strategy with respect to AP will be referenced later as the ***first-level pruning***. We will see in Section 2.2.3 a variation of it that is finer-grained.

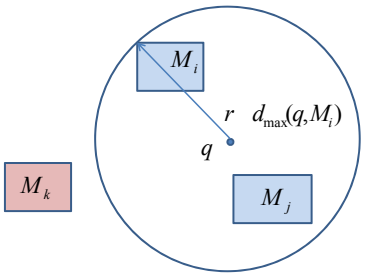

Fig. 2.1.: Pruning circle $C_{q,r}$ $(AP(M_i) < \tau)$

2.2.2 Maximal Probability (MP)

Definition 2.2.3 ***Maximal Probability*** *$MP(M)$ for MBR M is defined as: $\max_{U \in M} p_o$, where U is an object contained in M and p_o is the probability that U is present.*

The maximal probability MP is introduced for two purposes. Firstly, it can be used for ***pre-pruning*** to prune away MBRs with $MP < \tau$. Consider an MBR M with $MP(M) < \tau$. By definitions of MP and p_o, we know that $p_o \leq MP(M) < \tau$. Since the probability for U to be NN to q is at most the probability of its presence, we have $P_{nn}(U) \leq p_o < \tau$ for any object U in M. Hence we can safely prune away the entire M. Secondly, $MP(M)$ can also be used for further pruning beyond the capability of the first-level pruning, which we call the ***second-level pruning***, which is supported by the theorem below:

Theorem 2.2.2 *Let M_i be an MBR within a circle C. Let M_k be an MBR outside C. If $MP(M_k) \cdot AP(M_i) < \tau$, then for any object U in M_k, $P_{nn}(U) < \tau$.*

Proof For any object U in M_k, we have $P_{nn}(U) \leq p_o \cdot AP(C) \leq MP(M_k) \cdot AP(M_i) < \tau$, where p_o is the probability that U is present. ■

We have proved above the probability that any object U in M_k is NN to q is less than τ, so we can safely prune away the entire MBR M_k. This is called *second-level pruning*. In Fig. 2.1, if $AP(M_i)$ τ instead, we cannot use the first-level pruning to

prune away M_k. However, if $MP(M_k) \cdot AP(M_i) < \tau$ holds, we can use the second-level pruning to prune M_k away.

2.2.3 *AP*-Bounds

Unlike AP introduced in Section 2.2.1 that stores the absence probability of an entire MBR, *AP*-bounds store the absence probabilities of regions in the MBR specified by the bounds. The goal of *AP*-bounds is to shrink the size of the pruning circle as much as possible so that more MBRs outside the circle can be pruned away. This is a fine-grained version of the first-level pruning in Section 2.2.1. Both methods require that we have a pruning circle in which the absence probability is below τ.

The idea of probability bounds (e.g. x-bounds) is first proposed in [39] for range queries with probability thresholds. In this study, we use probability bounds for PNNT queries.

Definition 2.2.4 *AP-**bounds** $AP^l_M(x)$ (left AP-bound) and $AP^r_M(x)$ (right AP-bound) for MBR M are defined as a pair of lines intersecting with M such that the absence probability of the region to the left of $AP^l_M(x)$ and to the right of $AP^r_M(x)$ is no greater than the bounding probability x ($0 \leq x \leq 1$).*

Furthermore, we define *AP-distances* $d^l_M(x)$ and $d^r_M(x)$ to be distances from the left and right edges of MBR M to $AP^l_M(x)$ and $AP^r_M(x)$ respectively. We require that *AP*-bounds be tight — they are pushed towards the left or right edges of the MBR as much as possible while still satisfying Definition 2.2.4. This ensures that *AP*-bounds are unique. *AP*-bounds can be represented using *AP*-distances and the bounding probability x. For example, $AP^l_M(x)$ is represented using distance $d^l_M(x)$ and x itself. Suppose U_1, U_2, and U_3 are three objects in MBR M, as shown in Fig. 2.2. Let the bounding probability be x, then the *AP*-bounds of M ensures that the probability that none of the three objects is present within the *AP*-bounds is no more than x. Note that the bounding probability becomes larger as *AP*-bounds are pushed towards the edges, i.e., $0.5 > 0.2$ in Fig. 2.2.

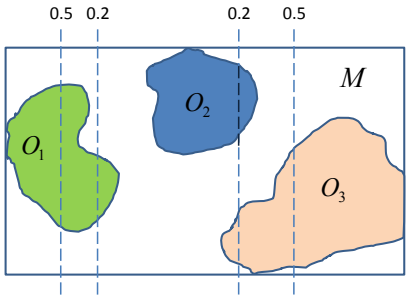

Fig. 2.2.: AP-bounds in MBR M

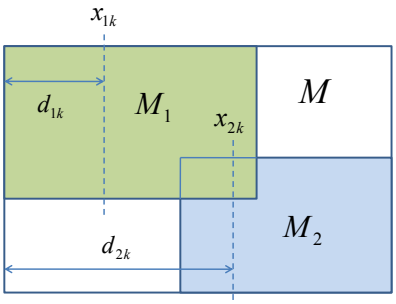

Fig. 2.3.: Construct left-AP-bounds of M

Pruning with AP-bounds: We first find the set of AP-bounds with bounding probability $x < \tau$ and as close to the edges of the MBR as possible. We let the new radius of the pruning circle be the minimal distance from the query point q to the AP-bound. Then all MBRs outside of this circle can be pruned away. Pruning using AP-bounds instead of AP of an entire MBR has the advantage that the resulting pruning radius is smaller, indicating that more MBRs are likely to be outside of the pruning circle and thus can be discarded immediately without further evaluation.

2.2.4 The Index Structure

Now that we have introduced all three kinds of information that we want to leverage in PNNT query processing, we redesign the R-tree index structure by adding all the information to the entries of the R-tree internal nodes. The construction of the augmented R-tree index is also discussed in details.

There are multiple entries in an R-tree internal node, each of which has an MBR (M) and a pointer (p) to a child node that stores information about all smaller MBRs contained in M. We augment R-tree by adding the following additional items to each entry of an internal node: i) AP ii) MP iii) left AP bounds and right AP bounds. Note that we keep a set of left and right AP bounds with different bounding probabilities x to suit queries of various thresholds. The list of x's is stored globally, each of which corresponds to a left and right pair of AP bounds in the MBR entry.

When constructing the new index, we propagate the additional information in MBR entries in a bottom-up fashion: The AP of an MBR at a higher level of the R-tree can be obtained by simply multiplying APs of all its child MBRs. Let $M_1, M_2, ..., M_m$ be the child MBRs of M. Then $AP(M) = \prod_{k=1}^{m} AP(M_k)$. In contrast, the MP of MBR M is obtained by finding the maximum MP among its child MBRs, i.e., $MP(M) = \max_{k=1}^{m} MP(M_k)$. To compute the left-$AP$-bounds $AP_M^l(x)$ of M, we compute the AP distance $d_M^l(x) = \max_{k=1}^{m} d_{M_k}^l(x)$ for each bounding probability x. The right-AP-bounds are computed in the same way. We call this method "Coarse Estimation Method" (CEM). Alternatively, we have "Fine Estimation Method" (FEM), which leverages the *AP hop function* to obtain a much finer estimation of AP-bounds. We compute AP hop functions for all of M's child MBRs and deduce the hop function of M from them.

Definition 2.2.5 ***AP hop function*** *is a function from AP-distance d to bounding probability x, denoted as $x = h(d)$. A hop function is with regard to an MBR M if d is the distance from AP-bounds to M's bounds.*

Note that for both left and right AP-bounds, we have a corresponding hop function. Suppose M_1 and M_2 are two MBRs contained in M, as shown in Fig. 2.3. $x_{11} \cdots x_{1m}$ are the bounding probabilities of left AP-bounds ($AP_{M_1}^l$) of M_1. Likewise, $x_{21} \cdots x_{2m}$ are the bounding probabilities of left AP-bounds ($AP_{M_2}^l$) of M_2. Let h_1 be the hop function of M_1 and h_2 for M_2. Let (d_{jk}, x_{jk}) be the points on h_j, where $j \in \{1, 2\}$, $1 \leq k \leq m$, and d_{jk} is the distance from M's left edge to M_j's AP-bound $AP_{M_j}^l(x_{jk})$. Moreover, the AP-bounds for both MBRs are ordered such that $d_{jk} < d_{jk+1} (d_{jm+1} = +\infty, d_{j0} = 0)$. Then we write function h_j as follows:

$$h_j(d) = x_{jk}, \quad \text{if} \quad d_{jk} \leq d < d_{jk+1} \tag{2.1}$$

Our goal is to compute M's hop function h from h_1 and h_2. The absence probability of the region within the AP-bound $AP_M^l(x)$ with AP-distance at least $\max(d_{1k_1}, d_{2k_2})$ is at most the product of absence probabilities within AP-bounds $AP_{M_1}^l(x_{1k_1})$ and $AP_{M_2}^l(x_{2k_2})$, that is, AP_M^l's bounding probability $x \leq x_{1k_1} \cdot x_{2k_2}$.

Having observed this property, we can obtain h from h_1 and h_2 as follows ($1 \leq \leftarrow k_1, k_2 \leq m$):

$$h(d) = x_{1k_1-1} \cdot x_{2k_2-1} \text{ if } \quad d \in [d_{1k_1-1}, d_{1k_1}) \text{ and } d \in [d_{2k_2-1}, d_{2k_2}) \tag{2.2}$$

Note that more than m AP-bounds for M can be computed from Equation 2.2. However, to be consistent with M_1 and M_2, we need to normalize function h so that it has only m AP-bounds. This can be done in a number of ways. One naïve solution is to keep the first m bounds and throw the others away. With the help of hop functions, we get tighter AP-bounds and thus more MBRs could be pruned away using first-level pruning.

2.3 Query Processing

Before presenting our PNNT query processing algorithm, we first introduce the Global AP (GAP) function that is essential for pruning.

2.3.1 GAP Function

GAP function maintains the global AP information for the query point q. Let the distance to q be d. The definition of GAP function is as follows:

Definition 2.3.1 *GAP **function** $GAP(d)$ is the probability that no object exists inside the circle $C_{q,d}$.*

GAP is used to find and shrink the pruning circle as much as possible so that all MBRs outside of the circle can be pruned away. The radius R of the current pruning circle is maintained globally and decreases as more MBRs are seen during the query processing. GAP is updated whenever a new MBR is retrieved, whose absence probability contributes to GAP to make it more accurate. The algorithm *updateGAP* has the details. We use M to denote an MBR and $M.AP(q.threshold)$ to denote the AP-bound of M with bounding probability no greater than the query

Algorithm 1 Update GAP

Require: The current GAP, the query point (q), the newly-seen MBR (M)

Ensure: The updated GAP

```
if M.ap < q.threshold then
    set currentPoint.d to be the distance between q and M.AP(q.threshold)
    if currentPoint.d == d_max(query, M) then
        currentPoint.ap = M.ap
    else
        currentPoint.ap = q.threshold
    end if
end if        // choosing a GAP point given M ends here
if GAP is empty then
    add currentPoint to GAP
else
    savedAP = currentPoint.ap
    insert currentPoint into GAP according to d, let the point before it be prevPoint
    if currentPoint is not the first point of GAP then
        currentPoint.ap = savedAp * prevPoint.ap
    end if
    if there are points after currentPoint in GAP then
        set their new ap to be the old ap times savedAp
    end if
end if
find the first point (boundaryPoint) in GAP with its ap ≤ q.threshold
set the pruning radius R = boundaryPoint.d
discard all points in GAP with d > R
```

Algorithm 2 PNNT Query Processing

Require: The augmented R-tree (*tree*) for all data items, the query point (*query*)

Ensure: All data items with NN probability greater than *query.threshold* (*results*)

```
prune(tree.root, query)
for each node in non-discarded leaf-level nodes after pruning do
  for each data item in node do
    if item is marked as 'c' (candidate) or 'k' (non-candidate to be kept) then
      add item to remains (non-discarded data items)
      if item is marked as 'c' then
        add item to candidates (NN candidates)
      end if
    end if
  end for
end for        // pruning stage ends here
for each item in candidates do
  P_nn = computeNNProbability(item, remains, query)
  if P_nn > query.threshold then
    add item to results
  end if
end for
return results        // refining stage ends here
```

threshold. For each point on GAP function, we use d to denote the distance to query q and ap to denote $GAP(d)$, the absence probability of the circle $C_{q,d}$.

2.3.2 PNNT Query Processing Algorithm

Our PNNT query processing algorithm (shown in Algorithm 2) has two stages: Pruning stage and refining stage. In the pruning stage, the algorithm prunes away nodes in the augmented R-tree with the help of the GAP function. The goal is to dynamically update GAP as we see more MBRs so that we can shrink the pruning circle accordingly. The refining stage then decides whether a NN candidate is indeed a query result by checking whether its exact NN probability is greater than the threshold.

The details of the pruning algorithm (i.e. `prune`) are in Algorithm 3. The input is the query point and the node in the tree where the pruning starts. Note that we update the GAP function whenever we see a new MBR using algorithm `updateGAP` introduced in Section 2.3.1. `MarkMBRs` (Algorithm 4) marks all MBRs in the node as 'c', 'k' or 'd' according to the latest GAP function, where 'c' means NN candidates, 'k' means non-candidates that we need to keep for the refining stage and 'd' means others to be discarded.

The nodes in the augmented R-tree are visited in a depth-first manner. The function `PickMBRtoExplore` picks an MBR in the node from all that are marked 'c'. The corresponding child of the node will then be explored. The criteria for picking is to choose the MBR that is furthest from the query point, in the hope that its children will be discarded soon.

2.4 Experimental Evaluation

We performed our experiments on 1-dimensional data represented as intervals. Each interval is the uncertain region of the data and its pdf is represented using histograms. The total probability p over the interval is either in (0, 1] or in (0.5,

Algorithm 3 Prune

Require: A node in *tree* (*node*) to start pruning, *query*

Ensure: All non-discarded nodes with marked MBRs

```
for each MBR M in node do
    updateGAP(M, query)
end for
markMBRs(node, query)
if node is a leaf then
    return
end if
next = pickMBRtoExplore(node, query)
while next != NULL do
    prune(next, query)
    markMBRs(node, query)
    next = pickMBRtoExplore(node, query)
end while
```

Algorithm 4 Mark MBRs

Require: A node in *tree* (*node*) to mark its MBRs, *query*

Ensure: All MBRs in *node* are marked

```
for each MBR M in node do
  if M is outside of the current pruning circle C_R centered at query with radius R
  then
    mark M as 'd'          // first-level pruning
  else
    mark M as 'c'
    if M.mp < query.threshold then
      mark M as 'k'        // pre-pruning
    else
      search in GAP for the last point satisfying GAP.d ≤ d_min(query, M)
      if M.mp * GAP.ap < query.threshold then
        mark M as 'k'      //second-level pruning
      end if
    end if
  end if
end for
```

1]. The threshold τ of the PNNT query is at least 0.1. All intervals are randomly generated within the range (0, 10000] and the size of the interval is in [1, 10]. For each experiment, we average the statistics over 10 randomly generated query points in (0, 10000]. The default values/ranges for data size n, threshold τ and total probability p is 100000 (100K), 0.3 and (0, 1]. We generated data with either uniform pdf (default) or Gaussian pdf. Default values of parameters are used unless otherwise specified. We ran our experiments (written in C++) on a PC with Intel T2500 2.00GHz CPU and 2.00GB main memory.

Effect of Data Size: We evaluated our algorithm by varying the data set size n from 10000 to 100000. We computed the pruning percentage of our algorithm by dividing the number of NN candidates (`candidateCount`) by n. We compared the pruning percentage when the total probability $p \in (0, 1]$ and $p \in (0.5, 1]$. Fig. 2.4 shows the result. Over 99.7% data items are pruned away for both cases while a random $p \in (0, 1]$ generally has a higher pruning percentage than $p \in (0.5, 1]$. We also evaluated the time cost of our algorithm with regard to the pruning and refining stages in Fig. 2.5. The total time cost (pruning and refining) is also shown in Fig. 2.5.

Effect of Threshold: We repeated the previous experiments with the data size fixed at 100000 and the threshold varying from 0.1 to 0.9 in Fig. 2.6 and Fig. 2.7. We further compared the three pruning techniques (`Prune0, Prune1, Prune2`) with the varying threshold in Fig. 2.8. The result shows that `Prune1` contributes the most of all three techniques with a pruning percentage around 99.8%, followed by `Prune0` and `Prune2`.

Data with Gaussian pdf: Our algorithm performs well for data with Gaussian pdf too. Fig. 2.9 shows the pruning percentage of the three pruning techniques over different thresholds. Compared with Fig. 2.8, we observe the similar results: `prune1` prunes most, followed by `prune0` and `prune2`. The pruning percentages of the three techniques are all above 94%.

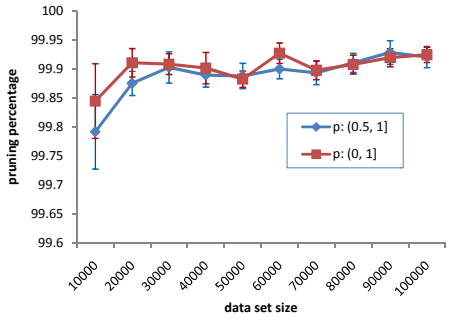

Fig. 2.4.: Effect of data size on pruning

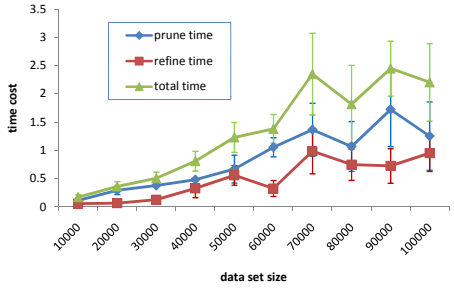

Fig. 2.5.: Effect of data size on time

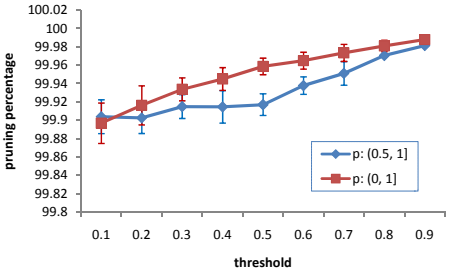

Fig. 2.6.: Effect of threshold on pruning

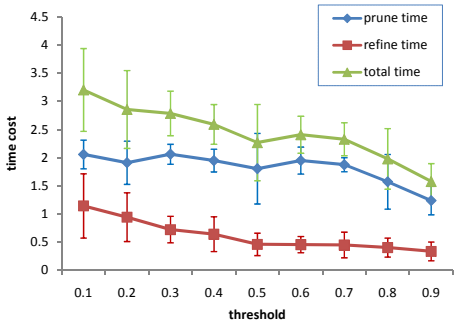

Fig. 2.7.: Effect of threshold on time

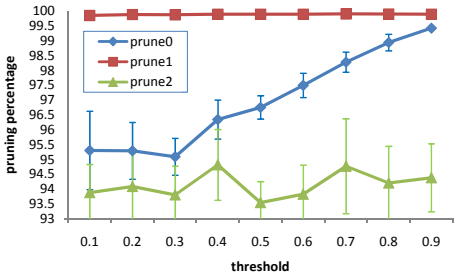

Fig. 2.8.: Pruning techniques

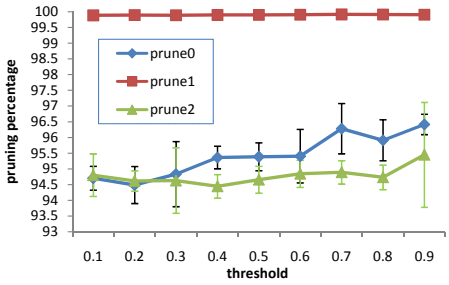

Fig. 2.9.: Gaussian pdf

3. PROBABILISTIC SKYLINE QUERIES

In this chapter, we present our instance-level probabilistic skyline problem and discuss two main variations of the problem: one with a probability threshold that can be used in pruning instances whose skyline probabilities fail the threshold (Section 3.2); the other with no threshold given, resulting in the computation of skyline probabilities of all instances (Section 3.3, Section 3.4). We further study an online version of computing skyline probabilities and present our results in Section 3.5.

3.1 Problem Definitions

We first introduce some preliminaries before presenting our probabilistic skyline problems. The uncertainty model that we adopt for both problems (with or without thresholds) is the one introduced in Section 1.1.2: An uncertain object can have multiple instances, each associated with a probability that the instance occurs. Our model is more general than the previous work on probabilistic skylines [16, 19], which assumes that instance probabilities of the same object always add up to 1. This assumption significantly simplifies the problem by enabling the pruning of all objects that are completely dominated by at least one object – since the existence of the dominating object is certain, the object being dominated is guaranteed to have a zero skyline probability and hence can be pruned right away. We remove this assumption in our model by allowing objects to have *missing probabilities*: the probabilities of an object's instances may add up to $x < 1$, indicating that the object does not exist with probability $1 - x$ (the missing probability). Under this new model, pruning becomes less straightforward: Even if one object is completely dominated by another, the former can still have a non-zero skyline probability as long as the latter does not exist.

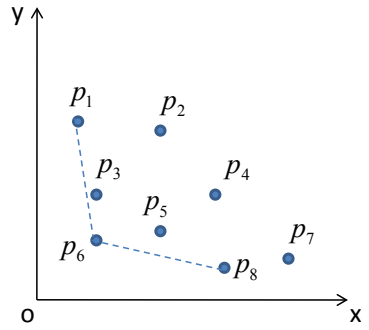

Fig. 3.1.: Skyline computation without uncertainty

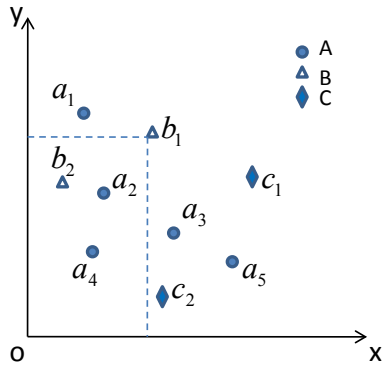

Fig. 3.2.: Skyline computation with uncertainty

Table 3.1: Instance probabilities in Fig. 3.2

A					**B**		**C**	
a_1	a_2	a_3	a_4	a_5	b_1	b_2	c_1	c_2
0.3	0.2	0.1	0.1	0.2	0.2	0.4	0.5	0.5

3.1.1 Dominance and Skyline

Given a data set S of n certain points: $p_1, \cdots, p_n$ in the data space $\mathcal{D}$ with d dimensions: $\mathcal{D}_1, \cdots, \mathcal{D}_d$, point p_i is said to *dominate* point p_j if $\forall k \in [1, d], p_i.\mathcal{D}_k \leq\leftarrow p_j.\mathcal{D}_k$ and $\exists l \in [1, d], p_i.\mathcal{D}_l < p_j.\mathcal{D}_l$. A point p_i is a *skyline point* or *in the skyline* if it is not dominated by any other point in S. A *skyline query* for certain data returns all skyline points.

Example 3.1.1 *In Fig. 3.1,* $p_1, p_2, \ldots, p_{10}$ *are 10 points in the two-dimensional space* $\mathcal{D}$*. The skyline points consist of* p_1, p_6 *and* p_8*. All the other points are dominated by at least one point, e.g.,* p_7 *is dominated by* p_8*, and* p_5 *is dominated by* p_6*.*

3.1.2 Skyline Probabilities

Fig. 3.2 shows an example of three uncertain objects with their corresponding instances in two-dimensional space. Note that both objects A and B have missing probabilities (0.1 and 0.4 respectively), since the probabilities of their instances sum up to less than 1.

Generally, we consider each instance as a d-dimensional point in the data space $\mathcal{D}$. The dominance relationship "$\prec$" between such points (i.e. instances) is the same as the dominance relationship between points for certain data as defined in Section 3.1.1. Following the convention in the database community, we assume that smaller values in each dimension are preferred over larger ones. We hence use p to refer to an instance, i.e. a data point in $\mathcal{D}$. An uncertain object U with k instances can be denoted as $U = \{p_1, ..., p_k\}$ where $p_i (1 \leq i \leq k)$ is an instance of U. The transitivity of the dominance relationship holds between instances [16], i.e. if $p_1 \prec p_2$, $p_2 \prec p_3$, then $p_1 \prec p_3$.

Definition 3.1.1 *The* skyline probability of an instance p, *denoted as* $Pr_{sky}(p)$, *is the probability that* p *exists and no instance of other uncertain objects that dominates* p *exists. Let* m *be the total number of uncertain objects and let* $p \in U_k$. *The skyline probability is defined as:*

$$Pr_{sky}(p) \stackrel{\text{def}}{=} Pr(p) \cdot \prod_{i=1, i \neq k}^{m} (1 - \sum_{q \in U_i, q \prec p} Pr(q)) \tag{3.1}$$

$$= Pr(p) \cdot \prod_{i=1, i \neq k}^{m} (1 - \sum_{q \in D_{S,i}(p)} Pr(q)) \tag{3.2}$$

where $D_{S,i}(p)$ *denotes the set of instances of object* U_i *in* S *that dominate* p. *We call it* dominance set. *The* skyline probability of an uncertain object $U = \{p_1, ..., p_k\}$ *is the sum of the skyline probabilities of all its* k *instances:*

$$Pr_{sky}(U) = \sum_{i=1}^{k} Pr_{sky}(p_i) \tag{3.3}$$

Example 3.1.2 *In Fig. 3.2, we have three uncertain objects A, B, C with multiple instances. The probabilities of instances are listed in Table 3.1. Each object has a skyline probability. For example, B has two instances b_1 and b_2. b_2 is not dominated by any point, so its skyline probability is simply its own probability 0.2. For b_1 to be a skyline point, none of the points that dominate b_1 (i.e., a_2, a_4, b_2, points in the rectangle) should exist. Hence its skyline probability is 0.4 * (1 - 0.2 - 0.1) = 0.28. The skyline probability of B is 0.48.*

Probabilistic skylines proposed in [16] are at the object level: only uncertain objects with a skyline probability over a certain threshold are returned. However, there are applications where the user is more interested in the instances, such as applying for job positions offered by different companies [17], where job positions are modeled as instances and the company that offers them as the object (the detailed example is in Section 3.1.4). In this case, a fine-grained look at probabilistic skylines is desired by the user. Below we propose two kinds of instance-level probabilistic skyline problems: One with thresholds available for pruning and the other without thresholds.

3.1.3 Probabilistic Skylines with Thresholds

In situations where users are interested only in instances with skyline probabilities over a certain threshold, we can bound the skyline probability of an instance p and use the threshold as well as the bounds (both upper and lower bounds) to quickly decide if p is worth evaluating or not (i.e., whether p has the potential to be an interesting instance). We denote the upper bound of $Pr_{sky}(p)$ as $Pr^{+}_{sky}(p)$, and the lower bound as $Pr^{-}_{sky}(p)$. Similarly, we also have $Pr^{+}_{sky}(U)$ and $Pr^{-}_{sky}(U)$ for an uncertain object U.

We define our instance-level probabilistic skyline problem as follows:

Definition 3.1.2 *Given a data set S of n instances that belong to m uncertain objects and a probability threshold θ, the instance-level probabilistic skyline analysis*

returns all instances with skyline probabilities at least θ. i.e. return the skyline set S_{sky} such that:

$$S_{sky} = \{p \in S | Pr_{sky}(p) \quad \theta\}$$

We also refer to S_{sky} (the skyline results) as probabilistic skylines for threshold θ.

We call the instances with skyline probabilities below the given threshold "uninteresting" and the others "interesting". The goal of our probabilistic skyline analysis is to quickly identify all interesting instances by leveraging the threshold for pruning. We discuss our algorithms in Section 3.2.

3.1.4 Probabilistic Skylines without Thresholds

Existing probabilistic skyline queries use a threshold to filter out results whose skyline probabilities are below the threshold [16, 19, 49]. We propose a different approach that abandons the use of thresholds in pruning and instead computes the skyline probabilities for all instances. This allows more flexibility for users to utilize the skyline results according to their own utilities rather than focus on probabilities alone. Moreover, with this new approach, we only need to compute the skyline probabilities once for all users, leaving users the largest flexibility to make their own decisions based on their current utilities and the skyline probabilities returned by the algorithm. We give two examples below to further illustrate the motivation behind this approach.

Motivation

Example 3.1.3 *The provision of a service often involves a number of different subservices: The quality of a patient's experience at an "urgent health care" facility depends on which of the doctors is on duty, which nurse is assigned to the patient, which staff member handles the billing and insurance paperwork, etc. In effect, there is not*

a single patient experience at that facility, but a collection of possible experiences, one for each k-tuple of team members that the patient interacts with during a visit. Similarly, the quality of the dining experience at a restaurant depends on which of the waiters the customer gets, and which of the cooks prepares the meal. Each possible k-tuple of team members has a probability, and determining which service providers dominate is easily seen to be a skyline computation problem (one in which a service provider is an object and a k-tuple of team members is an instance of that object). But why would one want the detailed skyline probabilities of the instances, when the skyline probability of an object would seem to suffice, which allows the efficient elimination (through thresholding) of dominated objects? The reason is that a customer's valuation function of an instance is variable from customer to customer (and can vary over time for the same customer): Thresholding may eliminate a low-probability object but whose instances (or a subset thereof) are peculiarly appealing to some customers. In other words, probabilities are not all that matters, consequences matter too: Computing all instance probabilities allows for subsequent customized valuation of objects according to different sets of valuation functions, some of which may have been unforeseen at the time the instance probabilities were first computed (there is no need to recompute them when a new valuation function is used).

Example 3.1.4 *Alice just got an MBA degree and is looking for jobs at various companies. Each company has multiple jobs that can be offered to MBAs. These jobs vary in titles, work units (departments) and geographic locations, which are not criteria in her decision making. The two criteria in deciding which company she wants to work for are the annual salary and the job security (shown in Fig. 4.2), both the bigger the better. The salaries vary among the available jobs in the company, and the job security of each is quantified by a numerical score (e.g., as reported by credit-rating agencies or financial analysts). A company can be considered as an uncertain object with its job openings as instances. At any time, an MBA can only take one job from a company, and the job offers between different companies are independent from each other. Each job is associated with a probability that the particular job will*

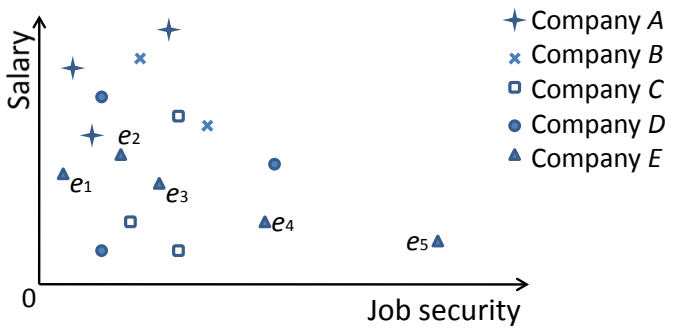

Fig. 3.3.: Companies and job openings.

be offered to an MBA by the company, which can be estimated using past statistics on success of applications. This probability may differ from one job to another, because some positions are easier to get than others or because there are more positions of a certain kind than others. Furthermore, the probabilities over all job instances of a company may add up to less than 1, as a result of some job that might be offered by the company but is unknown to the MBA at the time of her job hunting. The relative importance of salary and job security varies among different MBAs. For some, salary is most important while others prefer steady jobs. Even for the same MBA like Alice, her interest might change from focusing on salaries to preferring secure jobs over time.

Computing all skyline probabilities of instances instead of leveraging a threshold on skyline probabilities for pruning has several advantages:

First, the relative importance among the criteria of the skyline computation for a specific user is usually unknown to the system, hence for skyline queries with certain data, typically all skyline points are returned to the user. We use the term "*utility*" to refer to the satisfaction of a user when given a point. It is the responsibility of the user to identify the interesting points from the skyline set according to his/her own utility function. In case of uncertain objects with multiple instances, each instance of an object has a skyline probability from which the object's skyline probability is computed. If we threshold out uncertain objects (thus all its instances) based solely

on their skyline probabilities like in [16], we are making assumptions on the utility of a user in the sense that the user will not be interested in any uncertain object whose skyline probability is below a certain threshold. However, it is possible that for some user, an uncertain object with a relatively low skyline probability (below the threshold) has instances whose utilities are so huge to the user as to make them non-negligible.

Example 3.1.5 *In Fig. 4.2, Company E is very appealing to Alice for its best average job security. However, its skyline probability may be low due to the fact that most of its jobs are dominated by multiple jobs from other companies, hence could have been discarded if we had followed the thresholding approach in [16]. As a result, Alice could have missed a good opportunity to apply for Company E that has the most secure job* e_5.

As indicated in Examples 3.1.3 and 3.1.4, utility is user-defined, may change over time for the same user, or simply be unknown to the system at the time of the skyline analysis. Therefore, it is unfeasible to replace the skyline probability by (skyline probability)*utility and threshold on this new quantity.

Besides the main problem above, the thresholding approach suffers from the inherent problem of selecting a suitable threshold in search for interesting uncertain objects: A high threshold may lead to empty results, and hence the query needs to be restarted with a lower threshold; a low threshold may produce too many results and increase the query response time [21]. Moreover, the performance of heuristic pruning methods that leverage a given thresholddepends heavily on the characteristics of the data set being used and the value of the probability threshold. For some data and threshold, the skyline probability computation may have to be done for a large number (if not all) of the instances.

The problems with the thresholding approach have motivated the study of new probabilistic skyline analysis: computing skyline probabilities of all instances [17,18]. The outcome of such analysis is useful to all users in their decision making, despite their different utilities.

The Offline Problem

The first problem that we address is to compute the skyline probabilities of all instances, from which the skyline probabilities of all objects can be computed. We propose several algorithms to solve the problem. The input and the output of the algorithms are given below.

Input: m independent uncertain objects denoted as $U_1, \ldots, U_m$: For each object U_i, a set of n_i instances of that object is given (denote the set by U_i), where the set consists of d-dimensional points with probabilities that add up to at most 1. The sets of instances for different objects are disjoint, but two instances are allowed to have the same coordinates if they are from different objects. We assume that any two instances within the same object can not have the same coordinates. We use S to denote $\cup_{i=1}^{m} U_i$ and n to denote $|S| = \sum_{i=1}^{m} n_i$ where $n_i = |U_i|$. For each point $p \in S$, we use $\Pr(p)$ to denote its instance probability.

Output: For all $p \in S$, the skyline probability of $p : \Pr_{sky}(p)$

Note that we do not count the "effect" of instances that dominate p and come from the same object as p (i.e., instances in $D_{S,j}(p)$). The existence of p (with probability $\Pr(p)$) already ensures that none of the other instances of U_j exists. We use $D_S(p)$ to denote $\cup_{i=1}^{m} D_{S,i}(p)$ $(i \neq j)$, i.e., all instances in S that dominate p and are not from the same object as p.

From Equation 3.2, we can see that the skyline probability of p $(p \in U_j)$ consists of two parts: p's own probability $\Pr(p)$ and the probability that p is not dominated by any instance from other objects, which is computed as the product of probabilities that none of instances from other objects that dominate p exist. We denote the second part as $\hat{\beta}(p)$ (to be distinguished from $\beta(p)$ in Equation 3.6), i.e.,

$$\hat{\beta}(p) \stackrel{\text{def}}{=} \prod_{i=1, i\neq j}^{m} \left(1 - \sum_{p' \in D_{S,i}(p)} \Pr(p')\right) \tag{3.4}$$

Example 3.1.6 *In Fig. 3.2, instance b_1 is dominated by instances a_2, a_4 and b_2. Therefore, $D_S(b_1) = D_{S,A}(b_1) = \{a_2, a_4\}$ and $\hat{\beta}(b_1) = 1 - (\Pr(a_2) + \Pr(a_4)) = 0.7$.*

Since we can always compute $\Pr_{Sky}(p)$ from $\hat{\beta}(p)$ in constant time, we henceforth focus on computing $\hat{\beta}(p)$ rather than the actual skyline probability of an instance. Section 3.3 gives a sub-quadratic algorithm for computing all skyline probabilities of two-dimensional instances and Section 3.4 presents a new algorithm that further improves the sub-quadratic time complexity.

The Online Problem

The second problem that we address naturally extends from the first one: Now instead of computing the skyline probabilities for a fixed set of points (i.e., instances), we want to compute the probability that no instance from the fixed set dominates a point "on the fly", where no query point is known in advance. The input and the output of an algorithm that solves the online problem are as follows:

Input: Same as that for the offline problem except that now we also have an arbitrary query point q in the data space $\mathcal{D}$ that is not part of the input data set S.

Output: For the query point q, the probability that q is not dominated by any instance in S:

$$\prod_{i=1}^{m}(1 - \sum_{p \in D_{S,i}(q)} \Pr(p)). \tag{3.5}$$

Note that q is simply a point in $\mathcal{D}$ – We can treat it as the only instance of an extra online object (i.e., the $(m+1)_{\text{th}}$ object), and it has an instance probability of 1. When it is clear from context, we refer to the above probability in Equation 3.5 as the *online skyline probability* of query point q, which is different from the skyline probability of an instance in S defined in Equation 3.2. The algorithm for computing the online skyline probability of a query point is given in Section 3.5.

The major notations for the above probabilistic skyline problems are summarized in Table 3.2.

Table 3.2: Summary of notations

Notation	**Meaning**
m, n	number of all uncertain objects, number of all instances
d	number of dimensions
U_i	the ith uncertain object; the set of instances of U_i ($n_i = \|U_i\|$)
n_i	number of instances of U_i
S	the set of all instances ($n = \|S\|$)
p	point/instance in S
θ	probability threshold
$\Pr_{\text{sky}}(\cdot)$	skyline probability
$D_{S,i}(p)$	instances of U_i in S that dominate p
$D_S(p)$	instances of non-U_j objects in S that dominate p ($p \in U_j$)
$\sigma_i(p)$	sum of probabilities of U_i's instances that dominate p
$\hat{\beta}(p)$	probability that p is not dominated by instances of other objects
$F, \overline{F}$	the set of frequent objects, the set of infrequent objects
G_i	a group of infrequent objects
I_i	all the instances in G_i
μ	cutoff value for determining if an object is frequent or infrequent
$\alpha(p)$	probability that p is not dominated by any instance of other frequent objects (effect of frequent objects on p)
$\gamma(p)$	probability that p is not dominated by any instance of other infrequent objects (effect of infrequent objects on p)
$\gamma_i(p)$	probability that p is not dominated by any instance of other infrequent objects in group G_i (p can be an instance of objects outside of G_i)
$\gamma_i'(p)$	for an instance outside I_i, the probability that no instance $q \in I_i$ exists such that $q \preceq p$

3.2 Identifying Interesting Instances for Probabilistic Skylines

In this section, we present the algorithms for identifying interesting instances for probabilistic skylines, defined as instances with skyline probabilities greater than or equal to a given threshold (Definition 3.1.2).

We present two filtering schemes for efficient query processing:

1. Preliminary filtering scheme: techniques for avoiding the expensive computation of exact skyline probabilities by bounding them with easier-to-compute values for comparing to the threshold.

2. Elaborate filtering scheme: techniques for massive filtering through inter-instance comparisons that leverage one instance's bounds to filter other instances based on the dominance relationship.

3.2.1 Probabilistic Range Trees

We propose two indexing structures based on the range tree [52] to facilitate bounding and computing skyline probabilities. We augment the original range trees with additional probabilistic information stored at the internal nodes, which can be leveraged when querying the trees to quickly bound the skyline probability of a given instance p (the query instance). We call such trees *probabilistic range trees* (PRT). Section 3.2.1 introduces a general PRT built upon all n instances with probabilistic information. A similar indexing structure is described in Section 3.2.1, which is built for every uncertain object and has different probabilistic information. A total of m such trees are needed for all m objects. Our algorithms for the preliminary filtering use both trees, as we will see later in Section 5.1.2.

Overview

We explain the construction of the PRT on n d-dimensional points (representing all instances in the data set S). We begin with the base case of $d = 2$, and follow the

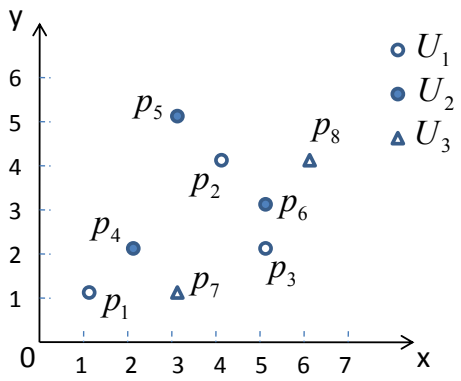

Fig. 3.4.: Probabilistic skylines with three objects and eight instances

Table 3.3: Instances in Fig. 3.4 and their skyline probabilities

Object	O_1			O_2			O_3	
Instance	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
Value	(1, 1)	(4, 4)	(5, 2)	(2, 2)	(3, 5)	(5, 3)	(3, 1)	(6, 4)
Probability	0.2	0.3	0.5	0.4	0.2	0.2	0.2	0.8
Skyline Probability	0.2	0.144	0.24	0.32	0.128	0.048	0.16	0

presentation of [52] modified to accommodate the probabilities. A complete binary tree T is built on top of the points sorted according to dimension D_1. Each internal node v of T points to an *info-list* L_v that contains the points at the leaves of the subtree of T rooted at v, sorted according to their D_2 dimension. Therefore, if v has children u and w, then L_v is the merge of L_u and L_w; we assume that every element of L_v stores its rank in each of the lists L_u and L_w (which implies that once a search item's position has been located in L_v it can be located in L_u and L_w in constant time). The space is obviously $O(n \log n)$. We also assume a derived probability (defined later in Section 3.2.1 and Section 3.2.1 respectively for the two kinds of PRTs) is associated with every element of L_v.

Fig. 3.5 illustrates a two-dimensional PRT built on top of the eight instances in the example of Fig. 3.4 and Table 3.3. The leaves of the PRT are the instances by

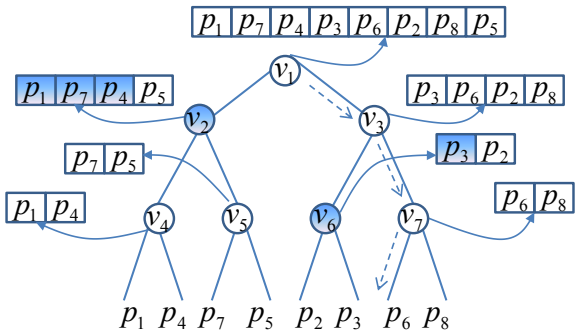

Fig. 3.5.: A 2-dimensional probabilistic range tree

the first dimension. Each of the internal nodes v_1 to v_7 points to an info-list that contains instances sorted by the second dimension. For example, v_2's info-list (L_{v_2}) has four instances p_1, p_7, p_4, p_5 with ascending values in the second dimension. They are instances at the leaf level of the subtree rooted at v_2. Since v_2 has two children: v_4 and v_5, L_{v_2} can be obtained by simply merging L_{v_4} and L_{v_5}.

A d-dimensional PRT is built inductively using $d-1$ dimensional PRTs: A complete tree T is built whose leaves are the n points sorted according to dimension D_1, and each internal node v of T points to a $d-1$ dimensional PRT that contains the elements at the leaves of the subtree of T rooted at v, organized according to the remaining $d-1$ dimensions (i.e., ignoring D_1). The space complexity is $O(n(\log n)^{d-1})$. Note that our construction ensures that the points in the info-lists are always sorted according to the last dimension. Fig. 3.6 illustrates such a d-dimensional PRT with $d = 3$. A node u in this PRT points to a two-dimensional PRT where a node v points to an info-list.

General Probabilistic Range Tree

To compute probabilistic skylines, we build the *general probabilistic range tree* (*general-PRT*) on all n instances in the data set S.

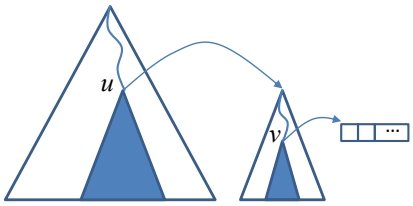

Fig. 3.6.: A 3-dimensional probabilistic range tree

Probabilistic Information

In the overview of probabilistic range trees, we did not explain what probabilistic information an info-list contains. Here we take a closer look at the info-lists in the general-PRT. The probabilities stored with info-lists are defined as follows:

Definition 3.2.1 *Let p be the k-th instance in an info-list L. Let p belong to an uncertain object $U_i (1 \leq i \leq m)$, let $\hat{L}$ be the list of the first k instances in L, then the probability associated with p (denoted as $_p$) in L is computed as:*

$$_p = \prod_{i=1}^{m} (1 - \sum_{q \in \hat{L}, q \in U_i} Pr(q)) \tag{3.6}$$

In other words, the probability $_p$ is the probability that no instance in $\hat{L}$ exists, i.e., the probability that p does not exist and no instance before p in the info-list L exists.

Creating Info-Lists

Given a set of instances, we can create an info-list L by adding each instance to L and then sort all instances by their d-th dimension values. After this, we need to compute the probabilistic information associated with each instance in the info-list. Based on Equation 3.6, we compute $_p$ for each p in L as shown in Algorithm 5. We use s_i to record the current probability sum for object U_i that has appeared in L. As we go through the instances in L, we update the corresponding probability sum (line

Algorithm 5 Computing 's for an info-list

Input: Info-list L

Output: Updated L with $_p$ computed for each instance p in L

1: **for each** $p \in L$ **do**
2: **if** $p \in U_i$ **then**
3: $s_i = 0$ // initialize probability sums
4: **end if**
5: **end for each**
6: $= 1$ // initialize the current
7: **for each** $p \in L$ **do**
8: **if** $p \in U_i$ **then**
9: $s = s_i$ // back up the old s_i
10: $s_i = s_i + Pr(p)$ // update s_i
11: $_p =$ $/(1 - s) * (1 - s_i)$ // compute $_p$
12: $=$ $_p$ // update the current
13: **end if**
14: **end for each**

10), and compute $_p$ based on the of the instance immediately before p in L (line 11).

As a concrete example on computing 's, let us look back at Fig. 3.5 where info-lists of a PRT is shown. Since the instances in this figure are from the example in Fig. 3.4, they are from different objects. Hence the PRT is actually the general-PRT. Therefore, for the info-list of the node v_2 (L_{v_2}), we can compute the probabilities associated with each instance using Algorithm 5. For example, $_{p_1}$ in L_{v_2} is $1 - Pr(p_1) = 1 - 0.2 = 0.8$, and $_{p_7}$ is $0.8 * (1 - Pr(p_7)) = 0.8 * (1 - 0.2) = 0.64$.

The time needed to compute $'s$ for an info-list L is $O(|L|)$, since we only scan the list twice. Note that the list of probability sums is only used for computing $'s$

of the instances when creating the info-list. It is not stored along with the info-list. Therefore, it will not bring an additional worst-case $O(m)$ space complexity for each info-list in the general-PRT.

Colored Probabilistic Range Trees

Besides the general-PRT built upon all n instances in S, we also have m specific PRTs, each built upon the instances of the corresponding object. If we render each instance with a color that indicates the source of the instance and match color i to object U_i, then each of these specific PRTs has only one color. Hence we call these trees *colored-PRTs* as opposed to the general-PRT. For the rest of the section, whenever we say "instance p of color i", we mean "instance p that belongs to object U_i".

Unlike the info-lists of the general-PRT, an info-list of a colored-PRT is associated with probability sums for each instance in the list. For the k-th instance p in an info-list L of a colored-PRT, its probability sum σ_p is computed as follows:

$$\sigma_p = \sum_{q \in \hat{L}} Pr(q) \tag{3.7}$$

where $\hat{L}$ is the list of first k instances in L. That is, σ_p is the sum of probabilities of all instances up until p in L. For computing all σ's, we simply go through each instance in L and accumulate the probability sum. The time complexity is $O(|L|)$. As an example, if we build a colored-PRT upon the instances of U_3 in Fig. 3.4, the colored-PRT has p_7 and p_8 as the leaves and an internal node that is also the root. The info-list pointed to by the root contain the two instances: p_7 and p_8 sorted by the second dimension values. We compute their σ's as follows: $\sigma_{p_7} = Pr(p_7) = 0.2$, $\sigma_{p_8} = \sigma_{p_7} + Pr(p_8) = 0.2 + 0.8 = 1$.

3.2.2 A Preliminary Filtering Scheme

Now that we have introduced both the general-PRT and the m colored-PRTs, we can use them to compute the bounds of the skyline probabilities that are used for filtering. The scheme presented in this section works at the individual instance level without comparing instances; the next section will present a more refined scheme where more savings are achieved through a mechanism of inter-instance comparisons whereby one instance's elimination (it is not a skyline result) implies a mass extinction of other instances that dominate it, and one instance's survival (it is a skyline result) implies a mass survival of other instances that are dominated by it.

Obtaining an Upper Bound

Given a query instance p, we can obtain $Pr^{+}_{sky}(p)$ by querying the general-PRT T_g as follows:

We begin with the base case of $d = 2$, as shown in Algorithm 6. Given the two-dimensional query $p = (p.D_1, p.D_2)$, we first locate the search path (call it $\mathcal{P}$) in T_g from the root to the position of the value $p.D_1$ among the leaves, then do one binary search for $p.D_2$ in the info-list L_{root} of the root of T_g. We record the position (rank) k of $p.D_2$ in L_{root} and call it the *search position* in L_{root}. We use $L_v[k]$ to denote the k-th instance (let it be q) in the info-list of the node v and $L_v[k].rankL$, $L_v[k].rankR$ to denote the rank of q in the info-lists of v's left child and right child respectively. These ranks are stored so that given the position of q in L_v, we can locate its position in info-lists of v's children in constant time. Since v is initially the root and k is initially the search position in L_{root}, we can obtain the search positions in the successive nodes as we walk down the search path $\mathcal{P}$.

We define the *left fringe nodes* of the PRT given the query instance p as the left children of the nodes on the search path $\mathcal{P}$ and are not nodes on $\mathcal{P}$ themselves. For example, in Fig. 3.5, the search path $\mathcal{P}$ for p_6 is $v_1 -> v_3 -> v_7 -> p_6$ (rendered with dashed arrows). The corresponding left fringe nodes are v_2 and v_6 (highlighted),

who are left children of v_1 and v_3 respectively. The leaf p_6 is on $\mathcal{P}$, so it is not a left fringe node despite the fact that it is a left child of v_7 on $\mathcal{P}$.

We use $\hat{L}_v$ to denote the truncated info-list of node v (L_v) with instances up till the search position in L_v. If v is a left fringe node, we call such $\hat{L}_v$ a *qualified info-list* . Fig. 3.5 highlights two qualified info-lists for the query p_6: One contains the first three instances of L_{v_2} and the other contains the first instance of L_{v_6} (refer to Fig. 3.4 for the values of instances).

When we reach the leaf at the end of the query, the variable *upperBound* in Algorithm 6 is the product of all 's we read along $\mathcal{P}$. It is indeed an upper bound of $Pr_{sky}(p)$, as we will see shortly. The time complexity for such a query is $O(\log n)$. In the example of Fig. 3.5, the upper bound that we get for $Pr_{sky}(p_6)$ is $_{p_4} *\!\!\leftarrow_{p3}$, where p_4 and p_3 are the last instances of the two qualified info-lists.

When $d > 2$, we obtain the upper bound $Pr^{+}_{sky}(p)$ by querying inductively on $d-1$ dimensional PRTs: Given the query $p = (p.D_1, \cdots, p.D_d)$, we first locate the path $\mathcal{P}$ in T_g from the root to the position of the value $p.D_1$ among the leaves. Then we walk along $\mathcal{P}$ and issue the query of $p' = (p.D_2, \cdots, p.D_d)$ for every $d-1$ dimensional PRT associated with every qualified node. The final $Pr^{+}_{sky}(p)$ is obtained by multiplying the values returned by all sub-queries. Such a query takes altogether $O((\log n)^{d-1})$ time, as we cannot avoid doing at most $O(\log n)$ binary searches in PRTs of the left fringe nodes for the first $d-1$ dimensions in order to find the qualified info-lists for reading 's.

We can obtain the qualified info-lists ($\hat{L}$'s) by modifying Algorithm 6: Instead of reading 's in line 9 and multiplying them along the path in line 10, we create an info-list $\hat{L}_u$ containing the first k' instances of L_u and add it to the result. The lemma below states that the set of all instances in $\hat{L}$'s is the set of all instances in S that dominate p, which can be easily proved from the search process and the definition of the general-PRT. Note that the notation $\hat{L}_i$ in the lemma is the i-th qualified info-list, not the qualified info-list at node i.

Algorithm 6 Compute an upper bound of $Pr_{sky}(p)$

Input: the general PRT T_g and a query instance p

Output: an upper bound of $Pr_{sky}(p)$

```
binary search for p.D_1 to find the search path 𝒫
binary search for p.D_2 in L_root, let the position be k
upperBound = 1
v = root          //walk along 𝒫 starting from root
while current node v is not a leaf do
  if the next node w ∈ 𝒫 is v.rightChild then
    k' = L_v[k].rankL
    u = v.leftChild
       = L_u[k'].beta      // read    from v's left child
     upperBound = upperBound *←
     k = L_v[k].rankR    // locate the position in L_w
   end if
   else                  // w is a left child of v
     k = L_v[k].rankL   // locate the position in L_w
   v = w                 // go one level down
 end while
 return upperBound
```

Lemma 3.2.1 *Let $\hat{L}_1 \cdots \hat{L}_t$ be qualified info-lists for query p. Let $S_{\hat{L}} = \cup_{i=1}^{t} S_{\hat{L}_i}$, where $S_{\hat{L}_i}$ is the set of instances in $\hat{L}_i$. Then we have: 1) $\forall q \in\!\leftarrow\!S_{\hat{L}}, q \prec\!\leftarrow\!p$; 2) $\forall q' \in S - S_{\hat{L}}, q' \not\prec p$.*

For every $\hat{L}_i$, let $_i$ be the associated with the last instance in $\hat{L}_i$, i.e. $_i$ is the probability that none of the instances in $\hat{L}_i$ exists. The next lemma (can be easily proved by induction) and theorem show that although we cannot compute $Pr_{sky}(p)$

directly from $_i$'s, we can compute $Pr^+_{sky}(p)$ to help prune p if this upper bound falls below the threshold θ.

Lemma 3.2.2 $(1-a_1)\cdots(1-a_t)$ $1-(a_1+\cdots+a_t)$, *where* $0 \le a_i \le 1, 1 \le i \le t$, *and* t 1

Following the notations in Lemma 3.2.1, we have:

Theorem 3.2.3 *Let* $_i$ *be the probability associated with the last instance in* $\hat{L}_i (1 \le \leftarrow i \le t)$ *where* L_i *is a qualified info-list for query* p, *then* $\prod_{i=1}^{t}$ $_i$ *is an upper bound of* $Pr_{sky}(p)$, *i.e.*

$$\prod_{i=1}^{t}\left(_i \quad \prod_{j=1}^{m}\left(1-\leftarrow\sum_{q\in U_j, q\in S_{\hat{L}}}\left(Pr(q)\right)\right)\right. \quad Pr_{sky}(p) \tag{3.8}$$

Proof We know from Definition 3.2.1 that $_i = \prod_{j=1}^{m}(1-s_{ij})$ where s_{ij} is the sum of probabilities of instances that belong to O_j and at the same time are instances in $\hat{L}_i$. We expand $_1, \cdots,$ $_t$ as follows:

$$\begin{aligned}
_1 &= (1-s_{11})\cdot(1-s_{12})\cdots(1-s_{1m}) \\
_2 &= (1-s_{21})\cdot(1-s_{22})\cdots(1-s_{2m}) \\
&\vdots \qquad \vdots \qquad \vdots \qquad \vdots \\
_t &= (1-s_{t1})\cdot(1-s_{t2})\cdots(1-s_{tm})
\end{aligned}$$

We multiply the above t equations together and obtain:

$$\prod_{i=1}^{t}\left(_i = \prod_{i=1}^{t}(1-s_{i1})\prod_{i=1}^{t}(1-s_{i2})\cdots\prod_{i=1}^{t}(1-s_{im})\right.$$

Each product $\prod_{i=1}^{t}(1-s_{ij}), 1 \le j \le m$ on the right hand side (RHS) is for the same uncertain object O_j. Since $0 \le s_{ij} \le 1$, apply Lemma 3.2.2 m times and we have:

$$\prod_{i=1}^{t}\left(_i \quad \prod_{j=1}^{m} \quad 1-\sum_{i=1}^{t}s_{ij}\right)^{\#} = \prod_{j=1}^{m}\left(1-\leftarrow\sum_{q\in O_j, q\in S_{\hat{L}}}Pr(q)\right)^{\#}$$

From Lemma 3.2.1, we further conclude that

$$\prod_{i=1}^{t}\Bigg(_i \quad \prod_{j=1}^{m}\left(1 - \overleftarrow{\sum_{q\in O_j, q\prec p}}\Bigg(Pr(q)\right)$$

Let $p \in O_k$. Since $\sum_{q\in O_k, q\prec p} Pr(q) + Pr(p) \leq 1$, the following holds for the RHS of the above inequality:

$$RHS = \quad 1 - \overleftarrow{\sum_{q\in O_k, q\prec p}}^{\#} Pr(q)\Bigg) \cdot \prod_{j=1, j\neq k}^{m}\left(1 - \overleftarrow{\sum_{q\in O_j, q\prec p}} Pr(q)\right)^{\#}$$

$$Pr(p) \cdot \prod_{j=1, j\neq k}^{m}\Bigg(\left(1 - \overleftarrow{\sum_{q\in O_j, q\prec p}} Pr(q)\right) = Pr_{sky}(p)$$

Therefore, Inequality 3.8 holds. ■

Theorem 3.2.3 shows that $\prod_{i=1}^{t}{}_i$ is an upper bound of the desired $Pr_{sky}(p)$, which proves that the value returned by $T_g.getUpperBound(p)$ is indeed a $Pr^{+}_{sky}(p)$. This directly points out a way of pruning the query instance: Given a threshold θ, as soon as we see the current product of 's (which is a $Pr^{+}_{sky}(p)$) fall below θ, we can stop and declare that p is not in the skyline, since $Pr_{sky}(p) < \theta$ must also hold.

Obtaining a Tighter Upper Bound

While using the general-PRT alone gives us an upper bound of the skyline probability, a tighter upper bound can be achieved by using both the general-PRT and the colored-PRT.

First, let us review Theorem 3.2.3: We have proved that

$$\prod_{i=1}^{t}\Bigg(_i \qquad 1 - \overleftarrow{\sum_{q\in U_k, q\prec p}}^{\#} Pr(q)\Bigg) \cdot \prod_{j=1, j\neq k}^{m}\left(1 - \overleftarrow{\sum_{q\in U_j, q\prec p}} Pr(q)\right)^{\#}$$

By dividing $\left(1 - \sum_{q\in U_k, q\prec p} Pr(q)\right)$ and multiplying $Pr(p)$ at both sides, we have

$$\frac{\prod_{i=1}^{t}{}_i \cdot Pr(p)}{1 - \sum_{q\in U_k, q\prec p} Pr(q)} \qquad Pr(p) \cdot \prod_{j=1, j\neq k}^{m}\Bigg(\left(1 - \overleftarrow{\sum_{q\in U_j, q\prec p}} Pr(q)\right)^{\#}$$

$$= Pr_{sky}(p)$$

We further observe that $Pr(p) \leq 1 - \sum_{q \in U_k, q \prec p} Pr(q)$ (remember $p \in U_k$), hence

$$\prod_{i=1}^{t} \left(i \quad \frac{\prod_{i=1}^{t} \quad _i \cdot Pr(p)}{1 - \sum_{q \in U_k, q \prec p} Pr(q)} \right. \quad Pr_{sky}(p)$$

i.e. $\prod_{i=1}^{t} \quad _i \cdot Pr(p) / \left(1 - \sum_{q \in U_k, q \prec p} Pr(q)\right)$ is a tighter upper bound of $Pr_{sky}(p)$ than $\prod_{i=1}^{t} \quad _i$.

We know $Pr(p)$ and $\prod_{i=1}^{t} \quad _i$ from querying the general-PRT, to obtain this tighter upper bound, the only part we need to know is $\sum_{q \in U_k, q \prec p} Pr(q)$, which is a probability sum that can be obtained by querying the PRT of color k, denoted as T_{c_k}. The algorithm for computing this sum given a query instance p is the same as computing the upper bound with the general-PRT in Algorithm 6 except this time we carry a sum instead of a product along the search path: Whenever a new probability is read from a qualified info-list (remember that the probability now is σ instead of , see Section 3.2.1), we add it to the current sum (initialized to 0). The final sum is then the sum of all σ's we read as we walk along the path. The algorithm to get all qualified info-lists in a colored-PRT given query p is exactly the same as that in the general-PRT described in Section 3.2.2.

The corollary below for querying colored-PRTs can be derived immediately from Lemma 3.2.1:

Corollary 3.2.4 *The set of instances of all qualified info-lists by querying T_{c_k} is the set of all instances of color k in S that dominate p.*

Therefore, the probability sum returned by querying T_{c_k} is indeed $\sum_{q \in U_k, q \prec p} Pr(q)$, i.e., the sum of probabilities of all instances that dominate p and belong to U_k at the same time. The algorithm to compute the tighter upper bound is summarized in Algorithm 7.

Obtaining a Lower Bound

We start with $d = 2$. For every instance (x, y), we define $s_{iL}(x)$ (resp., $s_{iB}(y)$) to be the sum of the probabilities of instances of color i that are to the left of x (resp.,

Algorithm 7 Compute a tighter upper bound

Input: query instance p, the general-PRT T_g and m colored-PRTs $T_{c_1}, \cdots, T_{c_m}$

Output: a tighter upper bound for $Pr_{sky}(p)$

1: obtain $Pr^{+}_{sky}(p)$ (*oldBound*) by querying T_g // Algorithm 6
2: **if** $p \in U_k$ **then** //Section 3.2.2
3: obtain the probability sum (*sum*) by querying T_{c_k}
4: **end if**
5: *newBound* = *oldBound* * $Pr(p)$ / (1 - *sum*)
6: return *newBound*

below y). It is straightforward to preprocess the n instances so that a query that asks for $s_{iL}(x)$ or $s_{iB}(y)$ can be processed in $O(\log n_i)$ time: Simply x-sort (resp., y-sort) the instances of color i and store in that sorted list the prefix sums of the probabilities: For each instance p in the list, the prefix sum of p is the sum of probabilities of all instances in the list up till p. Then we process a $s_{iL}(x)$ (resp., $s_{iB}(y)$) query by locating x (resp., y) in that list and reading the relevant prefix sum. Doing such preprocessing for all m colors takes $O(\sum_{i=1}^{m} n_i \log n_i) = O(n \log n)$, where n_i is the number of instances of object U_i. We assume this has been done.

The following lower bound (whose proof we omit) $Pr^{-}_{sky}(p)$ holds for any instance $p = (p.D_1, p.D_2)$ and $p \in U_k$.

$$Pr(p) \cdot \prod_{i=1, i \neq k}^{m} (1 - \min\{s_{iL}(p.D_1), s_{iB}(p.D_2)\})$$

The above lower bound for all n instances can be computed in time $O(m^2 + n \log m)$ (due to the space limit, we omit the details here). While this is good if m is much smaller than n (i.e., if each object has many instances), it is not satisfactory if m is close to n. In such a case we can compute the n lower bounds given below in total

time $O(n \log n)$ (they are thus easier to compute, but also less sharp than the above lower bound).

$$Pr(p) \cdot \max \left\{ \prod_{i=1, i \neq k}^{m} \left((1 - s_{iL}(p.D_1)) \right), \prod_{i=1, i \neq k}^{m} \left((1 - s_{iB}(p.D_2)) \right) \right\}$$

The above lower bounds can be easily extended to $d > 2$ by computing the sums of probabilities for each dimension, as we did for the first and second dimension in case $d = 2$.

3.2.3 An Elaborate Filtering Scheme

Recall that the preliminary filtering tries to filter out instances by bounding their respective skyline probabilities. The improved filtering scheme of the present section adds inter-instance comparisons to achieve wholesale filtering (positive or negative), i.e., it considers the impact of one instance's elimination or survival on other instances related to it by the dominance relationship. Therefore, the order in which instances are processed (individually, by bounding skyline probabilities as in the preliminary scheme) becomes crucial.

Filtering Rationale

Before presenting our elaborate filtering scheme, we first define a ratio called the "*key ratio*" for an instance p:

Definition 3.2.2 *For any instance $p \in U_k$, p's* key ratio *r is:*

$$r_p = \frac{Pr(p)}{1 - \sum_{p' \in U_k, p' \prec p} Pr(p')} \tag{3.9}$$

If $r_p \quad \frac{1}{2}$, we call p a "target instance"*.*

r_p can be easily computed in $O(\log n)$ by querying T_{c_k} to get the probability sum $\sum_{p' \in U_k, p' \prec p} Pr(p')$.

The following theorem states the conditions for negative filtering:

Theorem 3.2.5 *Let instance $p \in U_k$, instance $q \in U_l$. If $Pr_{sky}(p) < \theta$ and $p \prec q$, then:*

1) $k \neq l$: If p is a target instance, then $Pr_{sky}(q) < \theta$.

2) $k = l$: If p is a target instance or if $Pr(p)$ $Pr(q)$, then $Pr_{sky}(q) < \theta$.

Proof 1) Since p is a target instance, r_p $\frac{1}{2}$. We can deduce that

$$Pr(p) \quad 1 - \leftarrow \sum_{p' \prec p, p' \in O_k} \Big(Pr(p') - Pr(p) \tag{3.10}$$

Due to the transitivity of the dominance relationship, any instance that dominates p must dominate q. Hence

$$(3.10) \quad 1 - \leftarrow \sum_{p' \prec q, p' \in O_k} \Big(Pr(p')$$

Using the above inequality and the transitivity of the dominance relationship as well as the fact that $Pr(q) \leq 1 - \sum_{p' \prec q, p' \in O_l} Pr(p')$ (because both p' and q belong to O_l), we have

$$\begin{aligned} Pr_{sky}(p) = Pr(p) \cdot \prod_{i=1, i \neq k}^{m} \Big(1 - \leftarrow \sum_{p' \prec p, p' \in O_i} \Big(Pr(p') \Big) \Big(\\ 1 - \leftarrow \sum_{p' \prec q, p' \in O_k} \Big(Pr(p') \Big) \cdot \prod_{i=1, i \neq k}^{m} \Big(1 - \leftarrow \sum_{p' \prec q, p' \in O_i} \Big(Pr(p') \Big) \Big(\\ = \quad 1 - \leftarrow \sum_{p' \prec q, p' \in O_l} \Big(Pr(p') \Big) \cdot \prod_{i=1, i \neq l}^{m} \Big(1 - \leftarrow \sum_{p' \prec q, p' \in O_i} \Big(Pr(p') \Big) \Big(\\ Pr(q) \cdot \prod_{i=1, i \neq l}^{m} \Big(1 - \leftarrow \sum_{p' \prec q, p' \in O_i} \Big(Pr(p') \Big) \Big(= Pr_{sky}(q) \end{aligned}$$

Since $Pr_{sky}(p) < \theta$, $Pr_{sky}(q) < \theta$ also holds.

2) If p is a target instance and $k = l$, the proof in 1) still holds; if $Pr(p)$ $Pr(q)$, since $k = l$, we have:

$$\begin{aligned} \theta > Pr_{sky}(p) = Pr(p) \cdot \prod_{i=1, i \neq k}^{m} \Big(1 - \leftarrow \sum_{p' \prec p, p' \in O_i} \Big(Pr(p') \Big) \Big(\\ Pr(q) \cdot \prod_{i=1, i \neq k}^{m} \quad 1 - \leftarrow \sum_{p' \prec q, p' \in O_i} \Big(Pr(p') \Big) \Big(= Pr_{sky}(q) \end{aligned}$$

■

We call instances satisfying the above conditions "killers" – the elimination of themselves causes the massive extinction of others from the skyline result set. In contrast, the corollary below states the conditions for instances to be "saviors" – the survival of themselves causes the survival of others in the final skyline result. The proof of this corollary depends on the proof of $Pr_{sky}(p) \quad Pr_{sky}(q)$, which is exactly the same as the proof in Theorem 3.2.5.

Corollary 3.2.6 *Let instance $p \in U_k$, instance $q \in U_l$. If $Pr_{sky}(q) \quad \theta$ and $p \prec q$, then:*
1) $k \neq l$: If p is a target instance, then $Pr_{sky}(p) \quad \theta$.
2) $k = l$: If p is a target instance or if $Pr(p) \quad Pr(q)$, then $Pr_{sky}(p) \quad \theta$.

Schedule Instances

The theorem and corollary in the previous section together point out a way of filtering instances massively based on a single instance's skyline probability. As we have mentioned earlier, the order in which instances are processed is crucial. The goal of our refined filtering scheme is to maximize both negative filtering ("*killing*") and positive filtering ("*saving*") as we process the candidates list so that the number of the PRT queries (either for bounding or computing the exact skyline probability) is minimized. We propose the following heuristic for scheduling instances to achieve this goal:

Using the standard dominance counting techniques [50], we preprocess all n instances in $O(n \log n)$ time to compute two quantities $count^+(p)$ and $count^-(p)$ for every instance p, where $count^+(p)$ is the number of instances dominated by p and $count^-(p)$ is the number of instances that dominate p. We first sort the instances according to $count^+$ in the descending order. The list then becomes our initial candidate list for computing the skyline results.

Algorithm

The algorithm for the elaborate filtering consists of two parts: first the negative filtering, then the positive filtering. After scheduling all n instances to form the initial candidate list, we process each instance p in the candidate list in order by upper bounding $Pr_{sky}(p)$ (using the techniques in the preliminary filtering scheme). Then we do the negative filtering as shown in Algorithm 8. In line 6, we obtain the set of instances that are dominated by p by querying a mirror of our general-PRT (i.e. instead of returning instances that dominate p, it returns instances that are dominated by p). The order that we process instances guarantees that the current instance, if turned out to be a killer, can kill the largest number of instances (because its $count^+$ is the biggest among the unprocessed candidates).

After the candidate list has been exhausted, i.e. all killings have been done, we sort the remaining instances in the list by their $count^-$ in the descending order. We then process each instance q in this new candidate list in order by computing $Pr^-_{sky}(q)$ and compare it with θ to see whether q survives as a skyline result. If it survives, we move it from the candidate list to the skyline result S_{sky}. The rest of the algorithm is similar to the one in Algorithm 8.

Notice that we do negative filtering (killing) first, followed by positive filtering (saving). This is due to the asymmetry of killing and saving: A killer p kills all instances dominated by p, whereas a savior q only saves a portion of all instances that dominate q — only the target instances among them can be saved. Hence killing filters more than saving. It should come before saving to minimize the number of instances that need to be processed or further evaluated.

If an instance cannot be pruned by the preliminary or the elaborate filtering, we need to further evaluate it to decide whether the instance is really a skyline result by computing the exact skyline probabilities. This can be done by querying either the general-PRT or the colored-PRTs.

Algorithm 8 Algorithm for negative filtering in the elaborate filtering scheme

Input: data set S, threshold θ

Output: the candidate list $Cand$ after filtering

```
1: create the initial Cand from S //Section 3.2.3
2: for each instance p in Cand do
3:    compute Pr^+_sky(p) // Algorithm 6 and 7
4:    if Pr^+_sky(p) < θ then
5:     remove p from Cand
6:     get the set of instances dominated by p
7:     for each instance q in the set do
8:      if p is a target instance then
9:       remove q from Cand
10:     else
11:      if p, q are of the same color and Pr(p)    Pr(q) do
12:       remove q from Cand
13:      end if
14:     end if
15:    end for each
16:   end if
17: end for each
18: return Cand
```

Using General-PRT

From qualified info-lists ($\hat{L}$'s) for query p (see Section 3.2.2), we can get all instances in $\hat{L}$'s. By Theorem 3.2.1, these instances are all instances that dominate the query instance p in S. Therefore, we can go through all such instances to compute the exact $Pr_{sky}(p)$ according to Equation 3.2 and add p to the skyline result if

$Pr_{sky}(p)$ θ. The time complexity for computing $Pr_{sky}(p)$ is $O((\log n)^{d-1}+t)$, where t is the number of all instances in $\hat{L}$'s.

Using Colored-PRTs

For the instance p that belongs to object U_k, we can compute $Pr_{sky}(p)$ by querying all colored-PRTs except the one with color k. For each colored-PRT T_{c_i} $(1 \leq i \leq \leftarrow m, i \neq k)$, we obtain the sum of probabilities of all instances of color i that dominate p (see Section 3.2.2). We denote this sum as s_i. Then $Pr_{sky}(p) = Pr(p)\prod_{i=1,i\neq k}^{m}(1-s_i)$. The time complexity for computing $Pr_{sky}(p)$ is $O(m(\log n)^{d-1})$.

3.2.4 Probabilistic Skyline Algorithm

Now that we have presented our preliminary and the more elaborate filtering schemes and our algorithm for computing the exact skyline probabilities, we can propose our final algorithm for computing the instance-level probabilistic skylines given a threshold θ.

Two-Stage Algorithm

We propose a two-stage scheme for our instance-level probabilistic skyline algorithm, given the threshold θ:

1. **Filtering stage**:

   1) Initialize the skyline result S_{sky} to an empty set

   2) Initialize the candidate list to be all n instances in S

   3) Use the elaborate filtering scheme to reorder the candidate list, eliminate instances with skyline probabilities below θ, and move those with skyline probabilities at least θ to S_{sky}

2. **Refining stage**:

1) For each remaining instance p in the candidate list, compute the exact $Pr_{sky}(p)$ by querying the PRTs

2) Add p to S_{sky} if $Pr_{sky}(p)$ θ

3) Return the final S_{sky} as the set of skyline results to our probabilistic skyline problem

The filtering stage uses the elaborate filtering scheme we proposed in Section 3.2.3, which includes the usage of the preliminary filtering scheme in Section 5.1.2. We can also use the preliminary filtering scheme alone in the above algorithm, by changing Step 3 of the filtering stage. The remaining instances in the candidate list after the filtering stage are instances that can neither be eliminated nor guaranteed to belong to S_{sky}. We then query either the general-PRT or the colored-PRTs to compute the exact skyline probabilities of the instances and add those with skyline probabilities at least θ to S_{sky}.

Probabilistic Skylines at Object Level

While [16] computes all uncertain objects whose skyline probabilities meet a given probability threshold, our probabilistic skyline algorithms return all instances in the data set S whose skyline probabilities meet the threshold. The granularity of our skyline results is at the instance level, which is finer compared with the object level in [16]. Moreover, our instance-level algorithms can be adapted for obtaining the skyline results at the object level as follows:

For each object U_i, we compute the lower and the upper bounds of all its instances by using the preliminary filtering scheme. The sum of the lower bounds (resp. upper bounds) of U_i's instances becomes $Pr^-_{sky}(U_i)$ (resp. $Pr^+_{sky}(U_i)$). Let the threshold for the object-level probabilistic skylines be θ_o. We then check whether $Pr^-_{sky}(U_i)$ θ_o (i.e. U_i is a skyline result) and whether $Pr^+_{sky}(U_i) < \theta_o$ (i.e. U_i is not a skyline result). If U_i can neither be put to the skyline result set nor be discarded, we further compute

the exact skyline probabilities of its instances, sum them up to obtain $Pr_{sky}(U_i)$ and compare it with θ_o.

3.2.5 Experimental Evaluation

To evaluate the effectiveness of our filtering schemes and the scalability of our algorithms, we used two data sets, one real data set and one synthetic data set. The experiments were performed on a PC with Intel Core 2 Duo T9600 2.8GHz CPU and 6GB main memory running Ubuntu Linux operating system. All algorithms were implemented in C++. Currently our probabilistic range trees are stored in memory. Our future work will be to store the data structures on the disk to support efficient query processing at a larger scale.

Data Sets

In our experiments, we used the real data set: the NBA data set as in [16], kindly provided to us by the authors of [16]. The NBA data set contains 339,721 records about 1,313 players. Like in [16], we treat each player as an uncertain object and the records of the player as the instances of the object. Each record has three attributes: number of points, number of assists, and number of rebounds (large values are favored over small ones), i.e., the dimension $d = 3$. We assign random probabilities to instances of the same object such that the probabilities sum up to 1 (later for the synthetic data set, we allow missing probabilities of objects). This is different from [16], which assigns equal probabilities to instances of an object. Allowing different records of a player to have different probabilities captures the fact that the physical condition of a player usually changes from game to game (e.g., the player could be in great physical condition in some games, and have suffered from small injuries prior to other games).

Besides the NBA data set, we also used a synthetic data set generated similarly to [16, 17, 43] as follows: We first randomly generated the centers c of each uncertain

object. The value at each dimension of an instance has a domain [1, 1000] and was randomly generated in the hyper-rectangular region centered at c with the edge size uniformly distributed in the range [1, 200]. The default number of uncertain objects m is 20,000. The number of instances for an object is uniformly distributed in the range [1, 30] by default. Therefore, if $m = 20,000$ the expected total number of instances n is around 300,000. The default threshold θ is 0.01 for the instance-level skyline probabilities. Although the absolute value of the threshold seems small, this threshold is already very selective among skyline probabilities of all instances, as we can see later in the experiments below. This is mainly due to the fact that an uncertain object may have many instances, resulting in small occurrence probabilities for these instances to begin with before bounding/computing their skyline probabilities.

Effectiveness of Filtering

We evaluated the effectiveness of our two filtering schemes: we computed the percentage of instances filtered by the upper bounds and the lower bounds in the preliminary scheme, as well as the percentage of instances filtered by massive killing and saving in the elaborate scheme. During the evaluation, we also varied several parameters of the data set to test the scalability of our algorithms as well as to see how the parameters affect the filtering gain. Such parameters include: the data set size (number of objects/instances), the threshold, the average number of instances per object, the number of dimensions.

Effectiveness of the Preliminary Scheme

We evaluated the percentage of instances filtered by the upper bounds and the lower bounds in our preliminary scheme on the synthetic data set with $m = 2000$. We also evaluated the respective filtering capabilities of the upper bounds (Section 3.2.2) and the corresponding tighter upper bounds (Section 3.2.2). The result is shown in the first chart of Fig. 3.7. We always use the "normal" upper bounds (as opposed to

"tighter" upper bounds) first to filter instances, as they are easier to compute than the tighter ones. If the normal upper bound is above the threshold (i.e., cannot filter), then we further compute the tighter upper bound to see if the tighter one will help us filter the instance. Notice that if the upper bound is below the threshold, then the tighter upper bound must also fall below the threshold, indicating that both bounds can filter the instance.

Similarly, we show the effect of the lower bound on filtering in the second chart of Fig. 3.7. While the filtering percentage of both upper bounds increases as the threshold increases, the trend is reversed for the lower bound. This is because with higher thresholds, it is easier for an upper bound to fall below the threshold but harder for a lower bound to exceed it. Furthermore, we can see that the two upper bounds filter much more than the lower bound (over 97% of instances are filtered by the upper bounds), although these two kinds of bounds are computed independently.

Effectiveness of the Elaborate Scheme

The elaborate scheme uses the upper and lower bounds from the preliminary scheme, and further exploits the dominance relationship between instances for massive filtering (negative or positive). We evaluated the effectiveness of our elaborate filtering on both the real NBA data set and the synthetic data sets. The percentage of instances filtered after "killing" (i.e., negative filtering) and that after "saving" (i.e., positive filtering) are shown in the third chart of Fig. 3.7 for the NBA data with a varying threshold. The same plot is drawn for the synthetic data in the last chart of Fig. 3.7. For both data sets, killing filters instances massively while saving contributes an additional 0.1% or less to the final filtering percentage, as most of the instances (above 99.5% for a threshold over 0.002) have already been identified as uninteresting. This demonstrates our earlier statement that negative filtering filters much more than positive filtering in the elaborate filtering scheme. We also plotted the filtering percentage against the data set size (i.e., different m's) on the synthetic data in the first chart of Fig. 3.8. As the number of objects increases, the filtering percentage

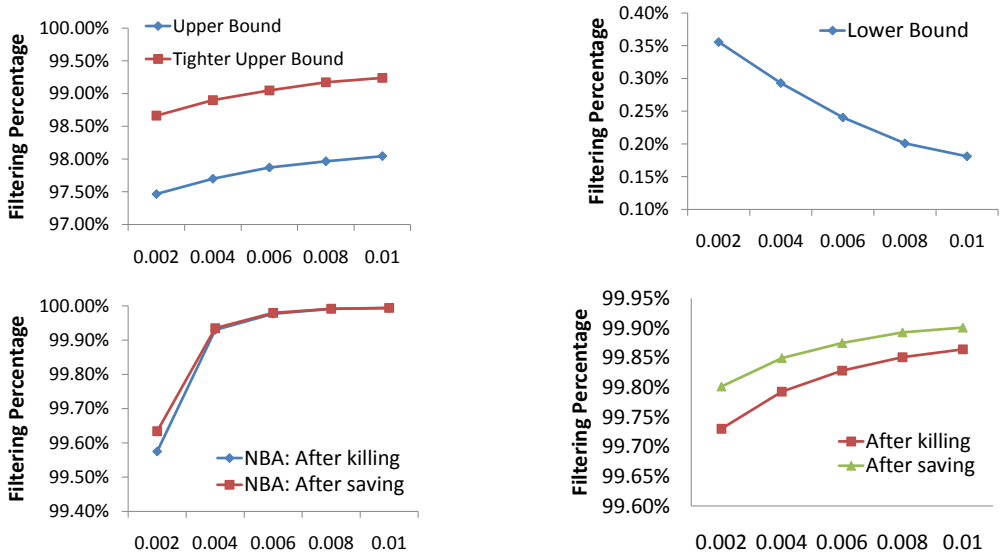

Fig. 3.7.: Effectiveness of preliminary filtering and elaborate filtering with respect to threshold

increases in general (because more instances are present to compete with each other and more instances are likely to be dominated by others) with an exception from $m = 12k$ to $m = 16k$, which might due to their particular distributions of instances in the data space.

As we know from Section 3.2.4, the final skyline result set consists of two parts: the instances that are saved during the elaborate filtering, and the instances whose exact skyline probabilities are verified to be above the threshold during the refining stage. We call the former "saved" ones; and the latter "refined" ones. The second and the third charts in Fig. 3.8 both compare "saved" and "refined", and display them in stacked columns, since the sum of the two is the actual skyline result size. The former is plotted from the real NBA data set while the latter is drawn from the synthetic data set. The two charts also differ in the x-axes: the former plots the skyline set against the threshold while the latter plots it against the data size m with a fixed threshold at 0.01. We observe that the threshold seems to have a much more significant effect than m on the size of the skyline results – the higher the threshold,

Fig. 3.8.: Effect of data set size (m), threshold and instance count per object

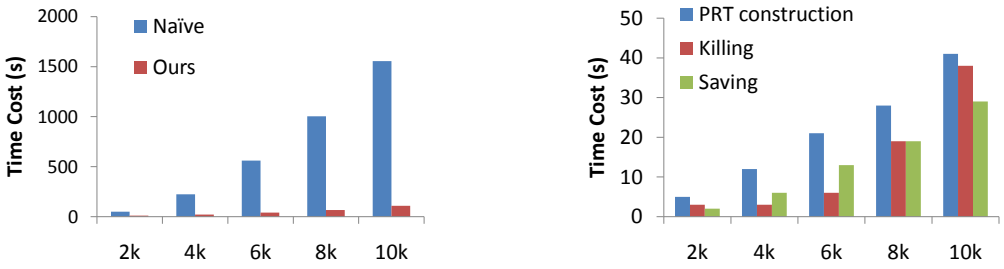

Fig. 3.9.: Comparison between our algorithm and the naïve algorithm

the smaller the set. The effect of m on the skyline size is not as obvious, though we can still presume that the bigger the data set, the bigger the final skyline set. It may depend heavily on the threshold in use: a smaller threshold may yield a clearer trend of this, since there are more instances that are likely to be above the threshold with a larger data set.

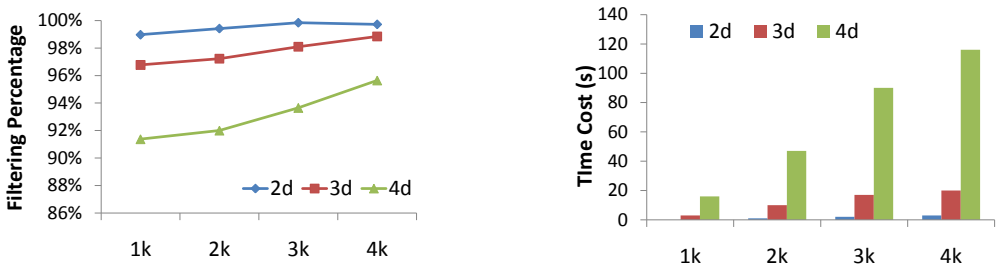

Fig. 3.10.: Effect of dimensionality

Comparison with the Naive Approach

We implemented the naïve approach to the instance-level probabilistic skylines for benchmarking, which uses a nested loop ($O(n^2)$) to compute the exact skyline probability of an instance by looking at all other instances. The time cost of our algorithm using the elaborate filtering scheme and that of the naïve algorithm are shown in the first chart of Fig. 3.9. The dimensionality is 3 and the threshold is 0.01. Our algorithm performs significantly better than the naïve one, and the advantage of our algorithm becomes even bigger as the data set size grows. The second chart of Fig. 3.9 provides a detailed view on how the time cost of our algorithm breaks down to three parts: the time cost for constructing PRT's (the general-PRT and the colored-PRT's), the time cost for negative filtering (killing) and that for positive filtering (saving). We can see that constructing the indices is actually the most expensive of the three: This is due to the fact that when we construct the trees, we also need to compute the probability information stored with each node for later use. Time cost for killing and saving also increases as m grows. More optimizations can be done for saving to reduce the time cost, which involves designing strategies to efficiently compute the lower bound. Due to the space limit, we will not discuss the details here.

Since our algorithms are designed specifically for the instance-level filtering with a more general uncertain model, while [16] focuses on the object-level filtering for probabilistic skylines, we do not think a comparison of the two will yield convincing

results when either their algorithms or our algorithms have to be specifically modified and optimized in order to suit the other's case and become comparable.

Effect of Other Parameters

We have mentioned the effect of the threshold and m on filtering in the previous sections. Now we will discuss the effect of the dimensions as well as the effect of the number of instances per object.

The two charts in Fig. 3.10 show how filtering percentage and time cost change with different dimensions and different data set size. Here the filtering percentage is computed as the total number of instances killed and saved divided by the total number of instances in the data set. We observe that given a data set, the filtering percentage decreases as the dimension increases. This is because an instance p is less likely to be dominated by another instance q that has values better than or equal to p's own values in every dimension. In addition, for all dimensions, the filtering percentage increases as m increases. For the time cost, increasing dimensions bring increasing overhead in constructing and querying PRT's, as seen in the second chart of Fig. 3.10: the time cost of the 4d data is over ten times more than that of the 2d data for the same number of objects.

Finally, the last chart of Fig. 3.8 shows how filtering changes with respect to the number of instances per object. The x-axis represents the maximum number of instances an uncertain object can have. For example, 90 means the instance count per object is generated uniformly between 1 and 90. We fixed the total number of instances to around 300,000 while changing the range of the instance count per object from [1, 30] to [1, 150]. Intuitively, more instances per object implies fewer objects given a fixed total number of all instances. It also suggests that each instance now has smaller probabilities to occur in the first place (the sum of the probabilities over all instances of an uncertain object cannot exceed 1), which means that the skyline probabilities of these instances may also be smaller because they cannot exceed their

own occurrence probabilities. Therefore, the filtering percentage increases, as the number of interesting instances has decreased.

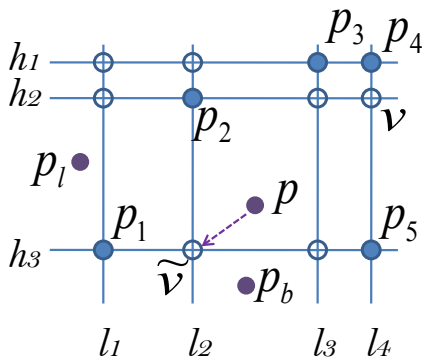

Fig. 3.11.: Space partitioning using a grid

3.3 Computing All Skyline Probabilities

Recall that the key in computing the skyline probability of an instance is computing $\hat{}(p)$ (Equation 3.4, Section 3.1.4). We first review two basic algorithms for computing $\hat{}(p)$ in Section 3.3.1 and Section 3.3.2, then present our offline algorithm in Section 3.3.3 that balances the use of the two basic algorithms to achieve subquadratic time complexity for two-dimensional data.

3.3.1 The Grid Method

We first present this space partitioning method for uncertain data with two dimensions: $d = 2$. Let C_x (C_y) be the set of x (y) coordinates of instances in S. Then $|C_x| \leq n, |C_y| \leq n$, as there might be instances with the same x or y coordinates. The elements of $C_x \times C_y$ form a grid of at most n^2 vertices, consisting of the original n instances (along with their probabilities), and the rest of the vertices that do not correspond to original instances. We therefore consider the latter as *"virtual"* instances of a non-existent object U_0 and assign to each a probability of zero.

Example 3.3.1 *Fig. 3.11 shows such a grid with five instances $p_1, ..., p_5$ marked as solid points. All the other vertices on the grid (7 in total), which are virtual instances, are marked as hollow points (e.g., vertex v).*

We seek to compute $\hat{}(p)$ for every grid vertex p (even the virtual ones). The reason that we also compute for virtual instances will be explained later in Section 3.3.3. Now we focus on the algorithm for doing this in $O(mn^2)$ time.

First we observe that it suffices to compute, for each vertex p, the m sums $\sigma_1(p), \ldots, \sigma_m(p)$ where

$$\sigma_i(p) = \sum_{p' \in D_{S,i}(p)} Pr(p'), \qquad i = 1, \ldots, m \tag{3.11}$$

because once we have those sums we can compute the desired $\hat{}(p)$ values for all p's (let p's object be U_j) in the grid in $O(mn^2)$ time, following the equation below:

$$\hat{}(p) = \prod_{i=1, i \neq j}^{m} (1 - \sigma_i(p)) \tag{3.12}$$

So we focus on the computation of all the $\sigma_i(p)$'s. The algorithm for computing them is given next.

1. **Process the horizontal grid lines:** For each horizontal line of $O(n)$ vertices, from left to right, we compute for every vertex p of that line the m *horizontal summations* $\sigma_i^*(p)$'s which are similar to the $\sigma_i(p)$'s except that they are defined one-dimensionally and relative only to the horizontal line that contains p (i.e., as if nothing exists other than what is on that horizontal line). Formally, let $p \in h$ where h is the horizontal line that contains p, let $p' <_h p$ denote the relationship "p' is to the left of p and is on the same horizontal line h as p", then we have

$$\sigma_i^*(p) = \sum_{p' \in S_i, p' <_h p} Pr(p') \tag{3.13}$$

We compute $\sigma_i^*(p)$'s in the following way: If p is the first vertex on h, set all m $\sigma_i^*(p)$'s to zero. Otherwise, let the left neighbor of p on h be $\hat{p}$. Copy all m $\sigma_i^*(\hat{p})$'s to $\sigma_i^*(p)$'s. If $\hat{p}$ is an original (not virtual) instance and $\hat{p} \in S_j$, add $Pr(\hat{p})$ to $\sigma_j^*(p)$.

The above takes $O(m)$ for each p on the horizontal line, hence we can compute $\sigma_i^*(p)$'s for all p's in time $O(mn)$ per horizontal line.

Example 3.3.2 *In Fig. 3.11, p_1 is an instance of U_1 with probability 0.8, p_2 and p_4 are instances of U_2 with probability 0.5 each, p_3 and p_5 are instances of U_3 with respective probabilities 0.6 and 0.1. Then for p_4 on the horizontal line h_1, $\sigma_1^*(p_4) = \sigma_2^*(p_4) = 0$ while $\sigma_3^*(p_4) = 0.6$.*

2. **Process the vertical grid lines:** We compute $\sigma_i(p)$'s for the vertices of each vertical grid line in bottom to top order:

$$\sigma_i(p) = \sigma_i^*(p) + \sigma_i(p') + \begin{cases} Pr(p') & \text{if } p' \in S_i \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

where p' is the grid vertex immediately below p on the vertical line l that contains p (hence its $\sigma_i(p')$ is already available because it has already been processed in the bottom-up order for l). If p is the very bottom vertex on l, then $\sigma_i(p) = \sigma_i^*(p)$. Note here we add $Pr(p')$ to $\sigma_i(p)$ to take into account probabilities of original instances on l that dominate p, which are not captured by either $\sigma_i^*(p)$ or $\sigma_i(p')$.

Example 3.3.3 *To compute $\sigma_i(p_4)$'s from $\sigma_i^*(p_4)$'s computed in Example 3.3.2, we follow Equation 4.2.3 (take $i = 3$ for example):*

$$\sigma_3(p_4) = \sigma_3^*(p_4) + \sigma_3(v) + 0 \tag{3.15}$$

$$= Pr(p_3) + \sigma_3^*(v) + \sigma_3(p_5) + Pr(p_5) \tag{3.16}$$

$$= 0.6 + 0 + \sigma_3^*(p_5) + 0.1 = 0.7 \tag{3.17}$$

Similarly, we compute $\sigma_1(p_4) = 0.8$, $\sigma_2(p_4) = 0.5$.

The next theorem states that $\sigma_i(p)$ is correctly computed by the algorithm above. The proof is straightforward, hence omitted.

Theorem 3.3.4 *For any vertex p on the grid, $\sigma_i(p)$ computed by the grid method is the sum of probabilities of instances of U_i that dominate p, i.e. Equation 3.11 can be deduced from Equation 3.13 and 4.2.3.*

Step 1 of the algorithm takes $O(mn)$ time per horizontal line, thus $O(mn^2)$ total time for all horizontal lines. Step 2 takes $O(mn)$ time per vertical line, thus $O(mn^2)$ total time for all vertical lines. Since it takes an additional $O(mn^2)$ to compute $\hat{}(p)$'s from $\sigma_i(p)$'s for all p's in the grid, the overall time complexity for computing $\hat{}(p)$'s is also $O(mn^2)$.

The above algorithm easily generalizes to dimensions $d > 2$, with a rather daunting time complexity of $O(mn^d)$. In the worst case m is proportional to n and the time complexity is then $O(n^{d+1})$. It is interesting that such an algorithm with a discouragingly bad time complexity, will actually play a critical part in the sub-quadratic time solution that we will provide later in Section 3.3.3.

3.3.2 Weighted Dominance Counting

The algorithm reviewed in this section, although inefficient if used as the sole method for solving the problem, will play a useful role as one of the two building blocks for the more efficient solution we give later.

The *weighted dominance counting* (WDC) problem is: Given a set S of n weighted points, compute for each point p of S the sum of the weights of all points in S that dominate p. If all weights are equal to 1, the problem is the same as counting points that dominate p. To be consistent, we use the same dominance concept here as in Section 3.1.1.

It is well known that the WDC problem can be solved in $O(n(\log n)^{d-1})$ time for d-dimensional points ([50, 51]). This immediately gives an $O(mn(\log n)^{d-1})$ time solution to our problem, by using the WDC algorithm m times (once for each object) on all n instances with their probabilities as weights: For object U_i, we assign the n_i instances of object U_i weights that are equal to their probabilities, and assign the other $n - n_i$ instances weight zero.

These m successive runs of the WDC algorithm give, for every instance $p \in S$, the m sums $\sum_{p' \in D_{S,i}(p)} \Pr(p')$, i.e., $\sigma_i(p)$'s (for $i = 1, \ldots, m$). From these, it is easy to use an additional $O(m)$ time per instance p in S to compute

$$\hat{}(p) \stackrel{\text{def}}{=} \prod_{i=1, i \neq j}^{m} (1 - \sum_{p' \in D_{S,i}(p)} \Pr(p')),$$

because each summation within the product is already available from one of the m runs of the WDC algorithm.

In the worst case, m is proportional to n and the time complexity is $O(n^2(\log n)^{d-1})$. The algorithm in Section 3.3.3 (developed in [17]) improved this worst-case time complexity down to $\tilde{O}(n^{2-\frac{1}{d+1}})$. In this work, we further improve the worst-case time complexity down to $\tilde{O}(n^{2-\frac{1}{d}})$.

3.3.3 The Combined Algorithm

Before presenting our sub-quadratic algorithm for computing all skyline probabilities and later its extension to high dimensions, we first give an intuitive overview on how our scheme works in the two-dimensional case.

Our algorithm works by balancing the use of two inefficient methods: One uses weighted dominance counting (WDC), the other is based on partitioning space into grids. Using the former alone to compute skyline probabilities for all instances would result in an $O(n^2 \log n)$ time complexity, whereas using the latter alone would take $O(n^2)$ time without computing for virtual instances. The schema of our algorithm is illustrated in Fig. 3.12, which shows the interplay between different subsets of uncertain objects. We use a dashed arrow from one set of objects S_A to another set S_B to denote the effect of instances in S_A on the skyline probabilities of instances in S_B. Specifically, by "effect of $p \in S_A$ on $p' \in S_B$" we mean the contribution of p to the summation $\sigma_i(p')$. We use ε and e to denote the effect computed during WDC and the effect computed during the grid method.

The main ideas of our algorithm are as follows:

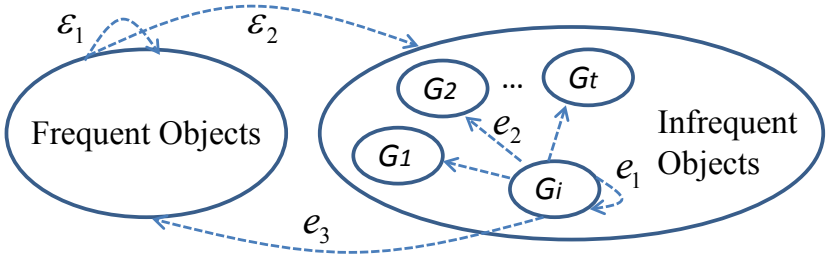

Fig. 3.12.: Schema of our algorithm

1. We use WDC for objects that are "frequent" with a number of instances that exceeds a special value μ. Objects that are not frequent are called "infrequent" objects. Specifically, we compute the effect of each frequent object on the skyline probabilities of all n instances in time $O(n \log n)$ using the WDC algorithm. This is captured by ε_1 and ε_2 in Fig. 3.12.

2. We merge the infrequent objects into groups such that each group contains between μ and 2μ instances (except that the last group could contain fewer than μ instances). In Fig. 3.12, the set of infrequent objects is divided into t groups: $G_1, \ldots, G_t$.

3. We use the grid method for each group of the infrequent objects. It is crucial here that we compute $\hat{}(p)$'s for virtual instances on the grid as well as original instances. Although this takes cubic time with regard to the number of instances in the group, it lends itself to the efficient computation of the effects of the group on all the other instances outside of the group through the use of a "bucketing" technique that processes together all the non-group instances that fall within a cell of the grid (a "bucket"); the details are given later in Section 3.3.3. In Fig. 3.12, e_1 captures the effect of a group G_i on its own instances while e_2 and e_3 capture the effect of G_i on all the other instances.

We use the case $d = 2$ to present our algorithm, and later extend it to the case $d > 2$. The case $d = 1$ is trivial to solve in $O(n)$ time and hence we omit it.

Algorithm 9 shows our algorithm for computing all skyline probabilities that follows the schema in Fig. 3.12. A more detailed explanation for two-dimensional data is given below.

Partitioning Objects

Partition the m objects into two sets: a set of *frequent* objects (F), consisting of every object U_i for which $n_i > (n \log n)^{\frac{1}{3}}$; the other objects (in $\overline{F}$) are said to be *infrequent*. We shall process the frequent objects differently from the infrequent ones. The value $(n \log n)^{\frac{1}{3}}$ is the partitioning point μ we mentioned earlier.

Handling Frequent Objects

Line 3 to 10 in Algorithm 9 shows how to handle frequent objects. For every frequent object U_i, we use WDC once (as explained in Section 3.3.2) to obtain $\sigma_i(p)$'s for every $p \in S$. Let p be an instance of object U_j. We compute the quantity

$$\alpha(p) = \prod_{U_i \in F, U_i \neq U_j} (1 - \sigma_i(p)) \tag{3.18}$$

as the effect of frequent objects on any instance p, which is illustrated by ε_1 and ε_2 in Fig. 3.12.

Handling Infrequent Objects

Line 11 to 18 in Algorithm 9 shows how to handle infrequent objects. We first go through the infrequent objects and, while doing so, partition the infrequent objects into groups, as follows: A new group is initially created, with no objects in it – we refer to it as the current group. The next encountered object is included in the current group: If this causes the number of object instances in the current group to exceed

Algorithm 9 Computing skyline probabilities

Input: a set S of n instances from m uncertain objects

Output: all instances with skyline probabilities

1: $Result = \emptyset \leftarrow$
2: partition objects into two sets: F, $\overline{F}$ // Section 3.3.3
3: **for each** U_i in F **do** // Section 3.3.2
4: obtain $\sigma_i(p)$ for all $p \in S$ by calling WDC(S)
5: **end for**
6: **for each** $p \in S$ (let $p \in S_j$) **do**
7: **for each** U_i in F and $U_i \neq U_j$ **do**
8: $\alpha(p) = \prod_{U_i \in F}(1 - \sigma_i(p))$
9: **end for**
10: **end for**
11: partition $\overline{F}$ into groups: $G_1, \ldots, G_t$ // Section 3.3.3
12: **for each** G_i in $\overline{F}$ **do** // Section 3.3.1
13: obtain $\hat{}_i(p)$ for all $p \in I_i$ by calling Grid(I_i)
14: **for each** $p \notin I_i$ **do**
15: locate p in a grid cell C of G_i
16: $\hat{}_i(p) = \hat{}_i(p')$ // p': bottom-left corner of C
17: **end for**
18: **end for**
19: **for each** $p \in S$ **do**
20: $Pr_{sky}(p) = Pr(p) \cdot \alpha(p) \cdot \prod_{i=1}^{t} \hat{}_i(p)$
21: $Result = Result \cup (p, Pr_{sky}(p))$ // add the pair
22: **end for**
23: **return** *Result*

$(n \log n)^{\frac{1}{3}}$ (i.e., μ), then that group is considered done (i.e., no longer current) and a new (initially empty) current group is started (to which the next object encountered is added, etc).

Comment and notation. Note that the number of instances in a completed group is between μ and 2μ, because an infrequent object does not add more than μ to the current group it joins. The number of completed groups (call it t) is obviously no more than $n/(n \log n)^{\frac{1}{3}}$. We denote these groups as $G_1, \ldots, G_t$. We use m_i to denote $|G_i|$ (= the number of objects in G_i), I_i to denote the set of all instances of the m_i objects in G_i (hence $|I_i| = \sum_{U_j \in G_i} n_j$).

The next step computes, for each group G_i, the quantity

$$\hat{}_i(p) = \prod_{U_j \in G_i, U_j \neq U_k} \left(1 - \overleftarrow{\sum_{p' \in D_{I_i,j}(p)}} Pr(p')\right)$$

for every $p \in S$ (let p belong to object U_k). The challenge is how to do this efficiently — we can no longer afford to use WDC because there are many objects in a G_i. We do the following instead:

For each of the groups $G_1, \ldots, G_t$ in turn, compute for every instance p in S the quantity $\hat{}_i(p)$, $i = 1, \ldots, t$. This is done as follows for G_i:

First, we use the grid method on G_i. This gives $\hat{}_i(p)$ for every grid vertex p, which is the effect of G_i on its own instances (i.e., e_1 in Fig. 3.12) as well as virtual ones. This takes cubic time with regard to the number of original instances in the grid.

Second, we compute the effect of G_i on all instances in S that are not on the grid of G_i, i.e., instances from other groups of infrequent objects and instances from frequent objects, as illustrated by e_2 and e_3 respectively in Fig. 3.12. The grid for G_i partitions the plane into $O(|I_i|^2)$ regions (called cells). We use binary search to first locate each point p of $S - I_i$ in the grid cell in which it lies (two binary searches per point – one to locate the vertical slab in which it lies and the other to locate the cell within the vertical slab). Then for each such point p we set $\hat{}_i(p) = \hat{}_i(p')$ where p' is the bottom-left corner of the grid cell containing p. Note that if such p' cannot

be found for p, then $\hat{}_i(p) = 1$. Since p' can be a virtual instance, it is crucial that in the grid method we compute for all vertices including the virtual instances so that the effect of G_i on p can be obtained instantaneously.

Computing Skyline Probabilities

For every p in S, compute the desired $Pr_{sky}(p)$ as

$$Pr_{sky}(p) = Pr(p) \cdot \alpha(p) \cdot \prod_{i=1}^{t} \hat{}_i(p) \tag{3.19}$$

Equation 3.19 is equivalent to Equation 3.2 because the set of frequent objects and the groups of infrequent objects are partitions of all uncertain objects; also, the uncertain objects are all independent from each other.

Complexity Analysis

In this section, we analyze the time complexity of our algorithm in Algorithm 9 for two-dimensional uncertain objects.

Partitioning objects (line 2 in Algorithm 9) takes $O(m)$ time by going through all m objects. Handling frequent objects consists of calling WDC algorithm to obtain $\sigma_i(p)$'s for all $p \in S$ (line 3 to 5) and then further computing $\alpha(p)$'s (line 6 to 10). The first part takes a total of $O(n^{\frac{5}{3}}(\log n)^{\frac{2}{3}})$ time, as the number of frequent objects is no more than $n/(n \log n)^{\frac{1}{3}}$, and the WDC for each of these takes $O(n \log n)$ time. The second part takes a total of $O(n^{\frac{5}{3}}/(\log n)^{\frac{1}{3}})$ time, as it takes $O(n/(n \log n)^{\frac{1}{3}})$ time to compute $\alpha(p)$ for each p and there are altogether n such p's in S. Therefore, the total time cost to handle frequent objects is $O(n^{\frac{5}{3}}(\log n)^{\frac{2}{3}})$.

Handling infrequent objects is more complicated. Grouping at line 11 takes $O(m)$ time. For each group G_i, calling the grid method at line 13 takes time $O(|I_i|^3)$ where I_i is the set of all instances in G_i. This is $O(n \log n)$ because $|I_i| \leq 2(n \log n)^{\frac{1}{3}}$ is ensured by our grouping method (see Section 3.3.3). For each p in $S - G_i$, we can locate it in

a grid cell of G_i in $O(\log |I_i|) = O(\log n)$ time by two binary searches. Line 16 takes constant time since $\hat{}_i(p')$ is already available from line 13. Thus the above takes a total of $O(tn \log n)$ time, where t is the number of the groups of infrequent objects. Since $|I_i| > (n \log n)^{\frac{1}{3}}$ for any group G_i except the last one, $t = O(n/(n \log n)^{\frac{1}{3}})$, resulting in a time complexity of $O(n^{\frac{5}{3}}(\log n)^{\frac{2}{3}})$ for handling infrequent objects from line 11 to line 18 in Algorithm 9.

Finally, computing the desired skyline probabilities for all p in S takes $O(tn)$ time, which is $O(n^{\frac{5}{3}}/(\log n)^{\frac{1}{3}})$.

The overall time complexity of our algorithm is, as argued in the analysis of each step, $O(n^{\frac{5}{3}}(\log n)^{\frac{2}{3}})$. For higher dimensions, we use the notation "$\tilde{O}(\cdot)$" which is similar to the "$O(\cdot)$" notation except that it ignores polylog factors (whereas the former ignores only constant factors). Similar to the complexity analysis for $d = 2$, for $d > 2$ we can show that a worst-case performance of $\tilde{O}(n^{2-\frac{1}{d+1}})$ time can be achieved to compute skyline probabilities for all instances. The detailed analysis can be found in [17].

3.3.4 Experimental Evaluation

Algorithms with good asymptotic complexity can often be impractical unless n is huge. Our algorithm is not one of those, and is in fact practical even for moderate values of n, as the following brief experimental evaluation demonstrates.

We performed our experiments on synthetic data sets of two dimensions ($d = 2$) and compared it with alternative algorithms for computing all skyline probabilities. We implemented all algorithms in C# and the experiments were performed on an Apple MacBook Pro with Intel T2500 2.0GHz CPU and 2GB main memory. In the rest of the section, we first describe the data sets used in our experiments and then discuss the experimental results.

Data Sets

We generated our synthetic data sets similarly to [16, 43] as follows: We first randomly generated the centers c of each uncertain object. Let $p.x$ and $p.y$ be the values of the first and the second dimension of an instance p, i.e. the x and y coordinates of p if we think of the instance as a point in a 2-d space. Both dimensions have a domain [1, 1000]. The default number of uncertain objects m is 1000. The number of instances for an object is uniformly distributed in the range [1, 40] by default. Therefore, by default the expected number of instances n is around 20,000. The x and y values of an instance are randomly generated in the rectangle centered at c with the edge size uniformly distributed in the range [1, 200].

Efficiency and Scalability

We compared our algorithm (`OURS`) against the priority search tree based algorithm (`PST`) [53], which queries the priority search tree built upon the data set S to find all instances that dominate a given instance p, i.e. the dominance set $D_S(p)$. The skyline probability of p then can be computed from Equation 3.2. The reason that we can leverage `PST` for finding $D_S(p)$ is that this problem can be converted to a two-dimensional range query: Retrieve all instances p' such that $p'.x \leq p.x$ and $p'.y \leq p.y$ (we make sure that if either is "=", then the other must be "¡"). It is well known that this can be done for query point p in time $O(\log |S| + |D_S(p)|)$. Let $n = |S|$, then the tree takes $O(n)$ space and can be built in $O(n \log n)$ time. We also implemented the naïve algorithm (`NAIVE`) with an $O(n^2)$ time complexity for benchmarking. It checks all n instances to determine the dominance set of p before computing the skyline probability of p.

Fig. 3.13 shows the running time of the three algorithms: `OURS`, `PST` and `NAIVE` on data sets with different total number of instances (i.e. n's) ranging from 10k to 60k. We used the default number of instances per object (in [1, 40]). As expected, `NAIVE` has the worst performance overall followed by `PST`, both of which have a worst-

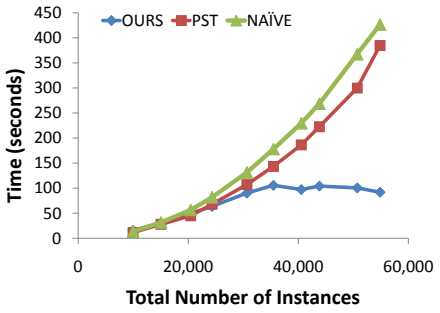

Fig. 3.13.: Running time on different algorithms

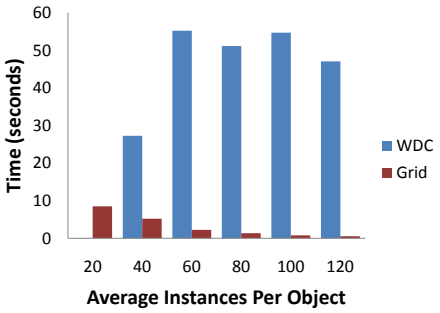

Fig. 3.14.: Effect of the average instance count

case time complexity of $O(n^2)$. Since our algorithm has a worst-case time complexity of $O(n^{\frac{5}{3}}(\log n)^{\frac{2}{3}})$, it performs better with larger n's, as shown in Fig. 3.13. `OURS` outperforms both alternatives even for moderate values of n such as $n = 30k$, while for smaller n's, its running time is slightly longer than the other two. As n increases, the advantage of our algorithm over the other two becomes more salient.

Effect of Instance Count per Object

Since we use the cutoff value $\mu = (n \log n)^{\frac{1}{3}}$ to partition objects into frequent and infrequent objects, different numbers of instances per object can affect the number of frequent objects versus the number of infrequent ones, resulting in different effects on the time cost of the `WDC` and the grid method (`Grid`) within our algorithm. For example, with $n = 20k$, $\mu = 65$. If the instance count per object generated in our synthetic data set is between [1, 80], there can be many more infrequent objects than frequent ones, since the expected instance count is $40 < \mu$. As a result, the time cost of `Grid` in our algorithm will be bigger.

We evaluated the effect of instance count per object on the time cost of `WDC` and `Grid` in Fig. 3.14. The average instance count per object varies from 20 to 120 and n is fixed to be around 20k. We can see that `WDC` generally takes much longer than `Grid`. Given $\mu = 65$, as instance count increases, the number of infrequent objects

decreases, hence the time cost of `Grid` decreases. The time cost of `WDC` first increases as a result of more frequent objects, then maintains at a level.

3.4 Improved Offline Algorithm

In this section, we propose a new algorithm for computing all skyline probabilities (the offline problem) that improves the algorithm presented in Section 3.3.3 by reducing the time complexity from $O(n^{\frac{5}{3}}(\log n)^{\frac{2}{3}})$ to $O(n^{\frac{3}{2}}\log n)$ in the 2D case. The performance gain comes from a different way to compute the effects of infrequent objects, which is more efficient than the way in Section 3.3.3. Such a better infrequent object handling method allows us to increase the cutoff value μ from $(n\log n)^{1/3}$ to $\sqrt{n}$, which reduces the time complexity of computing the effects of frequent objects, because there are less frequent objects when the cutoff value raises. Note that the improved offline algorithm handles the frequent objects in the same way as in Section 3.3.3.

3.4.1 Overview and Preliminaries

As in the preliminary algorithm of Section 3.3.3, we partition all uncertain objects into two categories: frequent objects and infrequent objects. The cutoff value μ is now different: We define objects with at least $\sqrt{n}$ instances as frequent objects and the rest as infrequent objects (i.e., $\mu = \sqrt{n}$).

For an instance $p \in U_j$, we compute the quantity:

$$\gamma(p) \stackrel{\text{def}}{=} \prod_{U_i \in \overline{F}, U_i \neq U_j} (1 - \sigma_i(p)) \tag{3.20}$$

as the effect of infrequent objects on instance p.

We have defined earlier the effect of frequent objects on instance p as $\alpha(p)$ in Equation 3.18. With $\alpha(p)$ and $\gamma(p)$, the skyline probability of an instance p can be easily computed as:

$$\Pr_{\text{sky}}(p) = \Pr(p) \cdot \alpha(p) \cdot \gamma(p).$$

The effect of frequent objects on all instances (i.e., $\alpha(p)$ for all $p \in S$) can be computed in $O(n\sqrt{n}\log n)$ time with $O(n)$ space. This is achieved by the WDC algorithm as we did in Section 3.3.3 in $O(n\sqrt{n}\log n)$ time, because there are only

$O(\sqrt{n})$ frequent objects and each WDC takes $O(n \log n)$ time. The space complexity is $O(n)$.

The effect of infrequent objects on all instances (i.e., $\gamma(p)$ for all $p \in S$) can be computed in $O(n\sqrt{n} \log n)$ time with $O(n)$ space using the *plane sweep* algorithm that will be described in Section 3.4.2. The sweep line algorithm requires a data structure T_y that satisfies the following properties:

- Instances in T_y are sorted according to their y-coordinates.
- Each instance p in T_y is associated with two quantities: w and σ, where w is a weight and σ is a sum of probabilities.
- Retrieving w and retrieving σ of an instance p both take $O(\log n)$ time.
- Inserting a new instance to T_y takes $O(\log n)$ time.
- Group-updating of the w's of instances whose y-coordinates are within a range takes $O(\log n)$ time. Specifically, the operation is called a *range weight update*, denoted as $\text{RWU}(a, b, c)$, with the intended effect that each instance p whose y-coordinate is between a and b ($a \leq b$) has its w multiplied by c.

A linear-space data structure T_y that achieves the above can be constructed as follows: First, sort all the input instances according to their y-coordinates. This is possible because the instances are given offline. Then construct a binary tree on top of the sorted instances. Each leaf node of the tree corresponds to an instance. Each internal node of the tree stores the range of y-coordinates of the instances of all the leaf descendants of that internal node. Each node v will maintain a value $\eta(v)$. The η's are used to compute the w quantities. The way η is used and updated is described below:

- To retrieve the w quantity of an instance p, find the path from the root to the leaf that corresponds to p in $O(\log n)$ time, and then the w quantity is the multiplication result of all the η values associated with the nodes (including the leaf) on that path.

- To insert an instance p with quantity w, first find the path from the root to the leaf that corresponds to p in $O(\log n)$ time, and denote the path by $v_1, v_2, \cdots, v_k$ where k is the length of the path. Then set the η value of the leaf v_k to be

$$w / \prod_{1 \le i \le k} \eta(v_i).$$

- To perform RWU(a, b, c), locate the $O(\log n)$ tree nodes whose subtrees are disjoint and covering exactly all the instances with y-coordinates between a and b, then update the η values of those $O(\log n)$ tree nodes by multiplying c.

3.4.2 Computing the Effects of Infrequent Objects

To simplify the presentation, we assume that instances have distinct y-coordinates, regardless of which object (frequent or infrequent) it is from.

Imagine moving an infinite vertical line (called the *sweep line*) from left to right across the plane, beginning at the leftmost instance of any uncertain object. As the sweep line moves, we maintain the instances from infrequent objects that we have seen so far in a dynamic data structure T_y that was described in 3.4.1. During the plane sweep, whenever the sweep line encounters an instance p, we compute the effect of infrequent objects on p (i.e., $\gamma(p)$) as follows: If p is an instance from a frequent object, we simply query T_y for the relevant w and σ (whose definition will be given below), from which we can compute $\gamma(p)$; otherwise, p belongs to an infrequent object, we need to insert p into T_y and update T_y. When there are more than one instances on the sweep line, we process the instances ordered by their x-coordinates.

We compute the γ's for all instances by initializing T_y to be empty, then moving the sweep line l from the leftmost instance in S to the rightmost instance in S. For every position x of l, we maintain in T_y the infrequent-object instances that have been encountered by l during its sweep so far, with the weight $w(p.y)$ of such an instance p in T_y being:

$$w(p.y) \stackrel{\text{def}}{=} \prod_{U_i \in \overline{F}} (1 - \sigma_l^i(p.y))$$

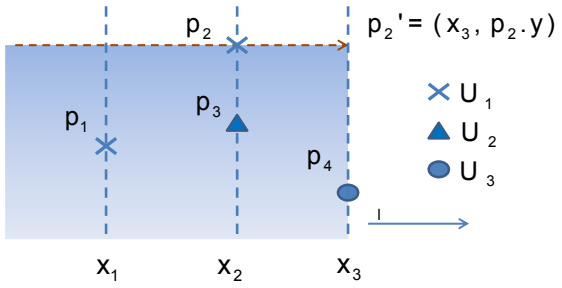

Fig. 3.15.: Example on computing $w(p_2.y)$

where $p.y$ is the y-coordinate of instance p and $\sigma_l^i(p.y)$ is a 1-dimensional version of $\sigma_i(p)$ for the horizontal projections on the current sweep line l of the instances of U_i already encountered by l. In other words, $\sigma_l^i(y)$ is the sum of the probabilities of the instances of U_i that are either at or to the left of the current position of l and also have a smaller y-coordinate than that of instance p. Recall that in the previous section, we mentioned that an instance p in T_y has two quantities: w and σ. We now define w to be $w(p.y)$ and σ to be $\sigma_l^i(p.y)$ (assuming $p \in U_i$). Note that if instance p belongs to a frequent object and is (geometrically) on line l then $w(p.y) = \gamma(p)$, but as l moves past p and sweeps through other instances, $w(p.y)$ changes and ceases to be $\gamma(p)$. If p belongs to an infrequent object, the relationship between $w(p.y)$ and $\gamma(p)$ is not as simple. We will see why this is the case in Section 3.4.2.

Example 3.4.1 *In Fig. 3.15, we have 4 instances from 3 uncertain objects U_1 to U_3. U_2 and U_3 are infrequent objects while U_1 is frequent. Suppose the current sweep line l is at x_2. $\gamma(p_2) = 1 - \Pr(p_3)$, since the only instance from an infrequent object that dominates p_2 is p_3. As the sweep line moves from x_2 to x_3, $w(p_2.y)$ changes as follows:*

- *When l is at x_2: $w(p_2.y) = 1 - \Pr(p_3) = \gamma(p_2)$;*
- *When l is at x_3: $w(p_2.y) = (1 - \Pr(p_3))(1 - \Pr(p_4)) < \gamma(p_2)$.*

The way we compute $w(p.y)$ for any p given the position x of the current sweep line l is that we always horizontally project p to l and get a virtual point $p' = (x, p.y)$. Then we compute the effect of infrequent objects on this virtual point p' without considering the original p. For example, when l is at x_3, p'_2 is dominated by infrequent-object instances p_3 and p_4, hence $w(p_2.y) = (1 - \Pr(p_3))(1 - \Pr(p_4))$. The shaded region in Fig. 3.15 indicates the region that needs to be examined when computing $w(p_2.y)$ for l at position x_3.

We next explain in detail what is done with T_y when the left-to-right sweep encounters an instance p of an object U_j (that could be either frequent or infrequent). Let the current sweep line be l.

Computing $\gamma(p)$

We distinguish two cases when computing $\gamma(p)$, where p is an instance of U_j:

Case 1: $U_j \in F$, i.e., p is an instance from a frequent object.

We first search for p's predecessor in T_y under the order of y-coordinates. This can be done in $O(\log n)$ time by using an auxiliary dictionary data structure to maintain the inserted instances, where y-coordinates are the search keys. Denote the predecessor by p'. Assume that p' is from object U_k. Note that $U_k \in \overline{F}$, because T_y only stores infrequent objects. Retrieve both $w(p'.y)$ and $\sigma_l^k(p'.y)$ from T_y in $O(\log n)$ time, then we can compute

$$\gamma(p) = w(p'.y) * (1 - \sigma_l^k(p'.y) - \Pr(p'))/(1 - \sigma_l^k(p'.y)). \tag{3.21}$$

This is because, after the projection, the instances from infrequent objects that dominate p are exactly those that dominate p', plus p' itself. In Equation 3.21, the division by $1-\sigma_l^k(p'.y)$ represents the cancelation of U_k's effect and the multiplication

by $1 - \sigma_l^k(p'.y) - \Pr(p')$ represents adding U_k's new effect on p after taking p' into account (where $p' \in U_k$). The total time needed to compute $\gamma(p)$ for a frequent-object instance p is just $O(\log n)$.

Case 2: $U_j \in \overline{F}$, i.e., p is an instance from an infrequent object.

We need to be more careful in computing $\gamma(p)$ in this case. First of all, $w(p'.y)$ might contain the effect from instances of U_j itself, which should be removed in computing $\gamma(p)$ (recall that in Equation 3.20, the instances from the same object are not considered), i.e., we should first divide $w(p'.y)$ by $1 - \sigma_l^j(p)$ to cancel the effect of instances from U_j. Then we check if $U_k = U_j$: If so, we are done; otherwise, we go on to incorporate the effect of p'.

As an example, suppose that the sweep line l in Fig. 3.15 now moves past the position x_3 and encounters a new instance p whose y-coordinate is between p_1 and p_4. The predecessor of p in T_y is hence p_4 (note that $p_4 \in U_3$, where U_3 is an infrequent object). If $p \notin U_3$, then $\gamma(p) = 1 - \Pr(p_4)$; otherwise, $\gamma(p) = 1$.

Updating T_y

If U_j is frequent, we do not need to update T_y, as T_y only contains instances from infrequent objects encountered during the sweep. However, if U_j is infrequent, we need to update T_y to reflect the change after seeing $p \in U_j$ at the current sweep line. This update is done as follows:

Let p'' be the nearest predecessor of p from the same U_j as p; this p'' can be obtained from an auxiliary dynamic dictionary structure that maintains the instances of U_j already encountered by the sweep line l, sorted by their y-coordinates. Such a step takes $O(\log \sqrt{n})$ query time, as U_j is an infrequent object with the number of instances no more than $\sqrt{n}$.

We insert instance p into T_y with $w(p.y) = \gamma(p)$ and $\sigma_l^j(p.y) = \sigma_l^j(p''.y) + \Pr(p'')$. Insertion takes $O(\log n)$ time, so does retrieving $\sigma_l^j(p''.y)$ from T_y.

Now we need to update $w(\hat{p}.y)$ for every instance $\hat{p}$ in T_y with greater y coordinate than that of p. The weight of each of such instances $\hat{p}$ has to be updated to reflect the effect of the newly inserted p that $\hat{p}$ dominates. There are two cases that we need to consider:

Case 1: $\hat{p} \in U_j$ (where $p, \hat{p}$ are from the same object):

We do the following updates

$$\begin{aligned} w(\hat{p}.y) \quad & w(\hat{p}.y) * (1 - \sigma_l^j(\hat{p}.y) - \Pr(p))/(1 - \sigma_l^j(\hat{p}.y)), \\ \sigma_l^j(\hat{p}.y) \quad & \sigma_l^j(\hat{p}.y) + \Pr(p). \end{aligned}$$

Note that this step takes $O(n_j \log n)$ time, which is $O(\sqrt{n} \log n)$ because U_j is infrequent.

Case 2: $\hat{p} \notin U_j$ ($p, \hat{p}$ are from different objects):

Although there can be $O(n)$ such non-U_j instances to be updated, this massive update can be done in $O(n_j \log n)$ time (i.e., in $O(\sqrt{n} \log n)$ time) because these non-U_j instances can be partitioned into $O(n_j)$ contiguous groups separated by instances from U_j: We update the weights of each such non-U_j group in logarithmic time using RWU(a, b, c) where a (resp., b) is the y coordinate of the instance p_u (p_l) of U_j that is at the upper (lower) boundary of that contiguous group of non-U_j instances, and

$$c = (1 - \sigma_l^j(p_l.y))/(1 - \sigma_l^j(p_l.y) + \Pr(p)).$$

Note that for non-U_j instances, the old effect of instances from U_j is $1 - \sigma_l^j(p_l.y) + \Pr(p)$ and the new effect is $1 - \sigma_l^j(p_l.y)$. This is because $p_l \in U_j$ and its $\sigma_l^j(p_l.y)$ has already been updated to include the effect of the newly-inserted p in Case 1.

The time for processing an instance encountered by the sweep line l is $O(\sqrt{n} \log n)$, and therefore the total time is $O(n\sqrt{n} \log n)$.

3.4.3 High Dimensional Cases

The extension to d-dimensional case is based on the following idea (assuming d is a fixed constant): Assume that the cutoff value to do the frequent/infrequent partition is μ. The effects of the frequent objects can be done using WDC (like we did in the 2D case), in $O(\frac{n}{\mu} \cdot n \log^{d-1} n)$ time. For the effects of infrequent objects, the algorithm uses a $(d-1)$-dimensional hyperplane to sweep through the instances one by one according to their sorted order. The sorted order is obtained by sorting the instances, where the k-th coordinate serves as the k-th most important key, i.e., the sort order is lexicographic. During the sweeping, the algorithm maintains the w and σ_l^i quantities like our 2D case, using an $O(n \log^{d-2} n)$-space data structure that support insertion, retrieval and range weight update in $O(\log^{d-1} n)$ time. Each time an instance is inserted into the data structure, it needs to do μ^{d-1} range weight updates. The dominating time comes from these range weight updates, so the total time complexity during the sweep is $O(n\mu^{d-1} \log^{d-1} n)$ time.

Based on the above analysis, the total space complexity is $O(n \log^{d-2} n)$, and the total time complexity is

$$O(n\mu^{d-1} \log^{d-1} n + \frac{n}{\mu} \cdot n \log^{d-1} n).$$

Choosing the cutoff value to be $\mu = n^{1/d}$ yields Theorem 3.4.2.

Theorem 3.4.2 *Offline skyline probabilities for d-dimensional uncertain data can be computed in* $O(n^{2-\frac{1}{d}} \log^{d-1} n)$ *time and* $O(n \log^{d-2} n)$ *space.*

3.5 The Online Algorithm

The online problem for probabilistic skylines is to compute the probability that a query point q is not dominated by any instance in a given data set, i.e., the *online skyline probability* of q. In this section, we present a data structure that can be preprocessed in $O(n\sqrt{n}\log n)$ time and space, so that this online skyline probability can be computed in $O(\sqrt{n}\log n)$ time.

Throughout this section, we assume that the x-coordinates of all the instances in S are distinct; the y-coordinates of all the instances in S are distinct; any online query point does not share the same x-coordinate or y-coordinate with any instance in S. This assumption is done without loss of generality, and dropping it would result in a more cluttered exposition but would not cause any change in our asymptotic complexity bounds.

3.5.1 Basic Idea

We partition all uncertain objects into two categories: frequent objects and infrequent objects. The cutoff value μ is set to $\sqrt{n}$ as in the algorithm of Section 3.4.

The effect of frequent objects on a query point q,

$$\hat{\alpha}(q) = \prod_{U_i \in F} (1 - \sigma_i(q)),$$

can be computed in $O(\sum_{U_i \in F} \log n_i) \leq O(\sqrt{n}\log n)$ time if a weighted dominance counting data structure [52] is built to compute $\sigma_i(q)$ for each $U_i \in F$. The total time and space for all such data structures are $O(\sum_{U_i \in F} n_i \log n_i) \leq O(n \log n)$.

It remains to show how to efficiently compute the effect of infrequent objects. For a query point $q \in \mathcal{D}$, define $\hat{\gamma}(q)$ to be the effect of infrequent objects on q, i.e.,

$$\hat{\gamma}(q) = \prod_{U_i \in \overline{F}} (1 - \sigma_i(q)). \tag{3.22}$$

We will show that computing $\hat{\gamma}(q)$ for any point $q \in \mathcal{D}$ takes $O(\log n)$ time with a data structure that can be built in $O(n\sqrt{n}\log n)$ time and space.

The key idea to compute Equation 3.22 efficiently is to transform the computation of $1-\sigma_i(q)$ to a range product problem. More specifically, for an object U_i, we will construct a set of dummy points U_i' and assign a dummy value $v(p)$ for each $p \in U_i'$ (the construction algorithm will be given in Section 3.5.2), such that computing $1-\sigma_i(q)$ for any online query $q \in \mathcal{D}$ is equivalent to a range product query in U_i' over the dummy values, i.e.

$$1-\sigma_i(q) = \prod_{p' \in U_i', p' \prec q} v(p'), \tag{3.23}$$

where $p' \prec q$ means that p' dominates q. We assume that the range product is 1 if there is no point dominating p (i.e., no point in the range).

Under this transformation, Equation 3.22 becomes a range product in the point set $U_{\overline{F}}' = \bigcup_{U_i \in \overline{F}} U_i'$ over the dummy values, i.e.,

$$\hat{\gamma}(q) = \prod_{U_i \in \overline{F}} \prod_{p' \in U_i', p' \prec q} v(p') = \prod_{p' \in U_{\overline{F}}', p' \prec q} v(p').$$

This range product problem can be solved in $O(\log |U_{\overline{F}}'|)$ time with an $O(|U_{\overline{F}}'| \log |U_{\overline{F}}'|)$ time and space preprocessing using a standard range query data structure [52].

We will provide an algorithm in Section 3.5.2 to construct the point set U_i' and their dummy values. Based on our construction, the following lemma holds:

Lemma 3.5.1 *For any $U_i \in \overline{F}$, there exists an algorithm to construct U_i' and the dummy values in $O(n_i^2)$ time and space. The number of dummy points in U_i' is n_i^2. The total number of dummy points in $U_{\overline{F}}'$ is bounded by*

$$\sum_{U_i \in \overline{F}} n_i^2 \leq \sum_{U_i \in \overline{F}} n_i\sqrt{n} = \sqrt{n}\sum_{U_i \in \overline{F}} n_i \leq \sqrt{n} \cdot n.$$

Therefore, the total time and space to build the range product data structure for $U_{\overline{F}}'$ is $O(n\sqrt{n}\log n)$, and a range product query takes $O(\log(n\sqrt{n})) = O(\log n)$ time.

Algorithm 10 and 11 summarize the data structures for online skyline probability. Theorem 3.5.2 summarizes the above analysis for Algorithm 10 and 11. Note that The

Algorithm 10 Preprocessing for Computing Online Skyline Probability

1: **for each** $U_i \in F$ **do**
2: Build a data structure for computing $\sigma_i(q)$.
3: **end for**
4: **for each** $U_i \in \overline{F}$ **do**
5: Compute the dummy point set U_i' and their dummy values.
6: **end for**
7: Put together all the dummy points to form $U'_{\overline{F}}$, i.e., $U'_{\overline{F}} = \bigcup_{U_i \in \overline{F}} U_i'$.
The dummy values are unchanged.
8: Build a range product data structure over the dummy values of $U'_{\overline{F}}$.
Ignore those dummy points whose dummy values are 1.

Algorithm 11 Computing Online Skyline Probability

Input: a query point $q \in \mathcal{D}$

1: Initialize RESULT 1.
2: **for each** $U_i \in F$ **do**
3: RESULT RESULT $\times (1 - \sigma_i(q))$,
where $\sigma_i(q)$ is computed by a weighted dominance counting query in U_i.
4: **end for**
5: RESULT RESULT $\times \hat{\gamma}(q)$,
where $\hat{\gamma}(q)$ is computed by a range product query in $U'_{\overline{F}}$.
6: **return** RESULT.

data structures can also be used to compute all skyline probability in $O(n\sqrt{n}\log n)$ time with slight and straightforward modifications.

Theorem 3.5.2 *Online skyline probability for a 2D query point can be computed in $O(\sqrt{n}\log n)$ time with data structures that can be built in $O(n\sqrt{n}\log n)$ time and space.*

3.5.2 Dummy Points and Dummy Values

For each infrequent object U_i, construct a grid using a method similar to the one described in Section 3.3.1 except that only one object is considered here (i.e., no grouping). There are n_i^2 grid points, and these grid points form the set U_i'. So it remains to assign the dummy values to the grid points so that Equation 3.23 holds. The dummy value assignment scheme is similar to the scheme for extension for the more general ULDB model in [11].

To assign dummy values to the grid points in U_i', we first process the grid to compute $\hat{\sigma}_i(p)$ for each grid point $p \in U_i'$, where

$$\hat{\sigma}_i(p) = \sigma_i(p) + Pr(p).$$

$\Pr(p)$ is the probability of an instance of U_i located in p. Note that $\Pr(p) = 0$ if p does not coincide with any instance of U_i. This processing is straightforward using two "FOR" loops, and it takes $O(n_i^2)$ time.

We then assign the dummy value for a grid point $p \in U_i'$ as follows:

- If p is the bottom-left grid point (the one with the smallest coordinates in all dimensions), the dummy value is set to $1 - \hat{\sigma}_i(p)$.

- If p is on the leftmost vertical line of the grid but not the bottom-left point, we assign value $\frac{1-\hat{\sigma}_i(p)}{1-\hat{\sigma}_i(p_b)}$ to p where p_b is the point immediately below p (see Fig. 3.16).

- If p is on the bottom horizontal line of the grid but not the bottom-left point, we assign value $\frac{1-\hat{\sigma}_i(p)}{1-\hat{\sigma}_i(p_l)}$ to p where p_l is the point immediately to the left of p.

- Otherwise, the dummy value $v(p)$ can be computed as:

$$v(p) = \frac{(1 - \hat{\sigma}_i(p))(1 - \hat{\sigma}_i(p_{lb}))}{(1 - \hat{\sigma}_i(p_l))(1 - \hat{\sigma}_i(p_b))}$$

  where p_{lb} is the point immediately to the left of p_b (or equivalently, the point immediately below p_l).

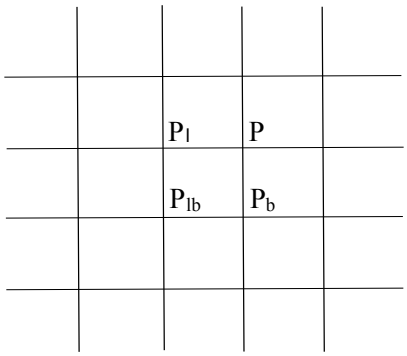

Fig. 3.16.: Illustration of P_b, P_l and P_{lb}

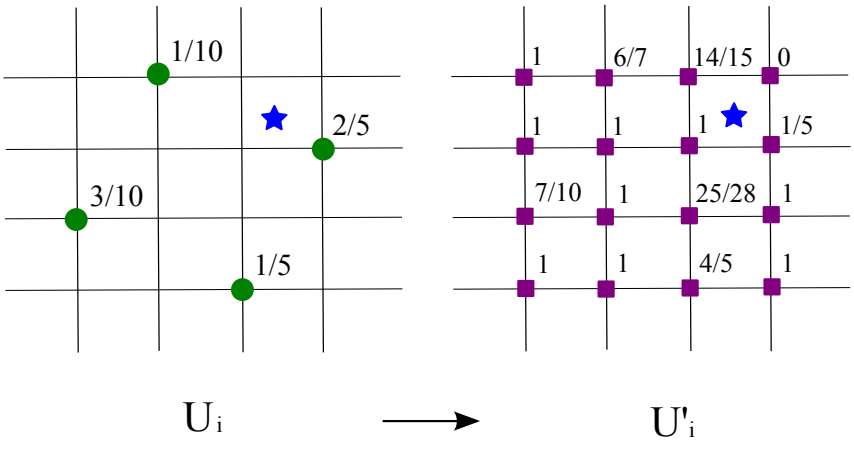

Fig. 3.17.: Illustration of a transformation from U_i to U_i'

The total time to assign the dummy values is clearly $O(n_i^2)$. Equation 3.23 holds, because we can verify that by assigning grid points such dummy values, we make sure that the product of dummy values of all grid points that dominate a query point $q \in \mathcal{D}$ is $1 - \sigma_i(q)$.

An example of the above transformation is illustrated in Fig. 3.17. The circles are the instances of U_i. The squares are the corresponding dummy points in U_i'. If an online query point q is located in the $\star$ position, then $1-\sigma_i(q) = 1-(0.3+0.2) = 0.5$ which is equal to the product of the dummy values $\frac{4}{5}, \frac{7}{10}, \frac{25}{28}$ and six 1's.

3.5.3 High Dimensional Cases

The generalization to d-dimensional cases is straightforward: Algorithm 10 and Algorithm 11 still apply when a d-dimensional data structure is used. We analyze the time and space complexity below.

Assume that the frequent/infrequent cutoff value is μ, and the dimension d is a fixed constant. The d-dimensional range sum data structures in line 2 of Algorithm 10 can be constructed in time $O(n_i \log^{d-1} n_i)$ time and space, such that $\sigma_i(q)$ can be computed in $O(\log^{d-1} n_i)$ time [52]. So the total time and space complexity of line 1 to 3 of Algorithm 10 is bounded by the order of

$$\sum_{U_i \in F} n_i \log^{d-1} n_i \leq \sum_{1 \leq i \leq m} n_i \log^{d-1} n = n \log^{d-1} n.$$

The dummy points at line 5 can be constructed in $O(n_i^d)$ time on a d-dimensional grid. So both the total number of dummy points (i.e., $|U_{\overline{F}}'|$) at line 7 and the total time and space of line 4 to 6 are bounded by the order of

$$\sum_{U_i \in \overline{F}} n_i^d \leq \sum_{U_i \in \overline{F}} n_i \mu^{d-1} = n\mu^{d-1}.$$

The range product data structure in line 8 can be constructed in time and space proportional to

$$|U_{\overline{F}}'| \log^{d-1} |U_{\overline{F}}'| \leq n\mu^{d-1} \log^{d-1}(n\mu^{d-1}) \leq n\mu^{d-1} \log^{d-1}(n^d) = O(n\mu^{d-1} \log^{d-1} n),$$

such that a range product query can be done in $O(\log^{d-1}(n\mu^{d-1})) = O(\log^{d-1} n)$ time [52]. Therefore, the total preprocessing time and space of Algorithm 10 is

$$O\left(n\log^{d-1} n + n\mu^{d-1} + n\mu^{d-1}\log^{d-1} n\right) = O(n\mu^{d-1}\log^{d-1} n).$$

At the query stage, the running time for line 2 to 4 of Algorithm 11 is in the order of

$$\sum_{U_i \in \overline{F}} \log^{d-1} n_i \leq \sum_{U_i \in \overline{F}} \log^{d-1} n = \left|\overline{F}\right| \log^{d-1} n \leq \overleftarrow{\frac{n}{\mu}} \log^{d-1} n.$$

The range product query at line 5 is $O(\log^{d-1} n)$. So the total query time is: $O(\frac{n}{\mu}\log^{d-1} n)$, dominated by line 2 to 4.

Based on the above analysis, different cutoff value μ's can lead to different preprocess/query time/space trade-offs. If we set $\mu = n^{1/d}$, then we get Theorem 3.5.3.

Theorem 3.5.3 *Given a query point q, its online probability for d-dimensional data can be computed in $O(n^{1-\frac{1}{d}}\log^{d-1} n)$ time with data structures that can be built in $O(n^{2-\frac{1}{d}}\log^{d-1} n)$ time and space.*

3.5.4 Experimental Evaluation

We performed our experiments on synthetic data sets of two dimensions ($d = 2$) and compared it with alternative algorithms for computing all skyline probabilities. All the algorithms were implemented in C++ and the experiments were performed on a PC with Intel Q6600 2.4GHz CPU and 3GB memory.

Data Sets

We generated our synthetic data sets similarly to the previous work [17, 43] as follows: We first randomly generated a centers c for each uncertain object. Both dimensions have a domain $[1, 1000]$. The number of uncertain objects m ranges from 100 to 4000. The number of instances for an object is uniformly distributed in the

range [2, 40] by default. The x and y coordinates of an instance are randomly generated in the rectangle centered at c with the edge size uniformly distributed in the range [1, 200].

Efficiency Comparisons

We compared our algorithm (`OURS`) against the previous work (`PRVS`) [17] and a range tree based algorithm (`RNGT`) [52]. The `OURS` is implemented using the online algorithm in Section 3.5, because it also works for the offline problem. The range tree based algorithm queries a range tree built upon the data set S to find all instances that dominate a given instance p, i.e., the dominance set $D_S(p)$. The skyline probability of p then can be computed from Equation 3.2. The time complexity of such a range tree based approach is $O(\sum_{p \in S}(\log |S| + |D_S(p)|))$. We also implemented the naïve algorithm (`NAIVE`) with an $O(n^2)$ time complexity for benchmarking. It checks all n instances to determine the dominance set of p before computing the skyline probability of p.

Fig. 3.18 shows the running time of the four algorithms: `OURS`, `PRVS`, `RNGT` and `NAIVE` on data sets with different total number of instances (i.e., n's) ranging from 2,161 to 84,128. As expected, `NAIVE` has the worst performance overall followed by `RNGT`. The `OURS` performs best, followed by `PRVS`. As n increases, the advantage of our algorithm over the other three becomes more salient.

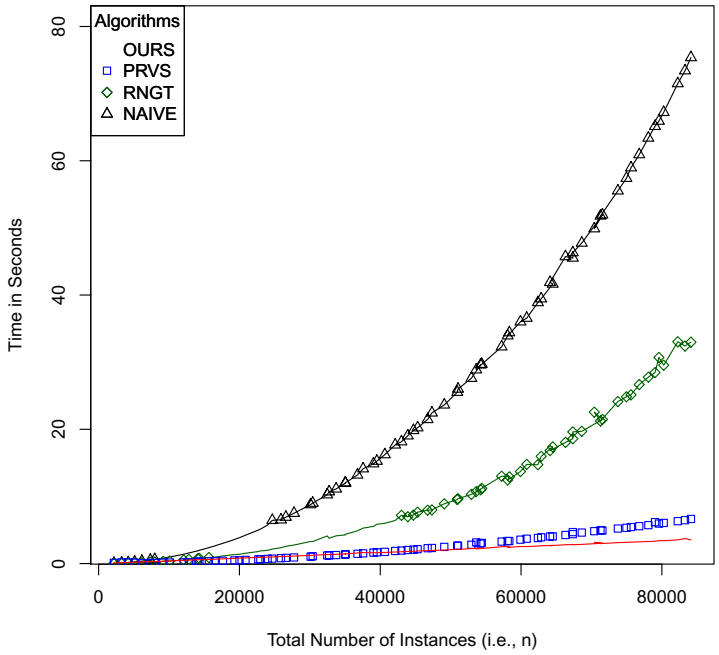

Fig. 3.18.: Efficiency comparisons

4. PROBABILISTIC THRESHOLD SPJ QUERIES

In this chapter, we discuss the optimization opportunities and techniques for probabilistic threshold queries that involve selections, projections and joins (SPJ) under the Orion uncertain data model.

4.1 Problem Definition

We begin with a running example throughout the chapter. With the aim of being as general as possible, we have chosen to use the Orion uncertain data model [1] (see Section 1.1.4) since it encompasses many other recent models (such as Trio [3], MayBMS [6], and MystiQ [4]) while having the advantage of handling continuous data as well as the capability of handling both tuple uncertainty and attribute uncertainty. We then formally define the probabilistic threshold queries and introduce the threshold operator. We finally state the goal for our PTQ optimization.

Note that we do not support duplicate elimination for our threshold SPJ query optimization under the Orion model. Since the Orion model allows continuous data with attribute uncertainty, the semantics of duplicates in this case is not clear. In Chapter 5, we adopt the general tuple uncertainty model (see Section 1.1.2) that only handles discrete uncertainty, and study the threshold SPJ query optimization that allows duplicate elimination.

4.1.1 Running Examples

We present below a query on uncertain tables that serves as a running example throughout this chapter.

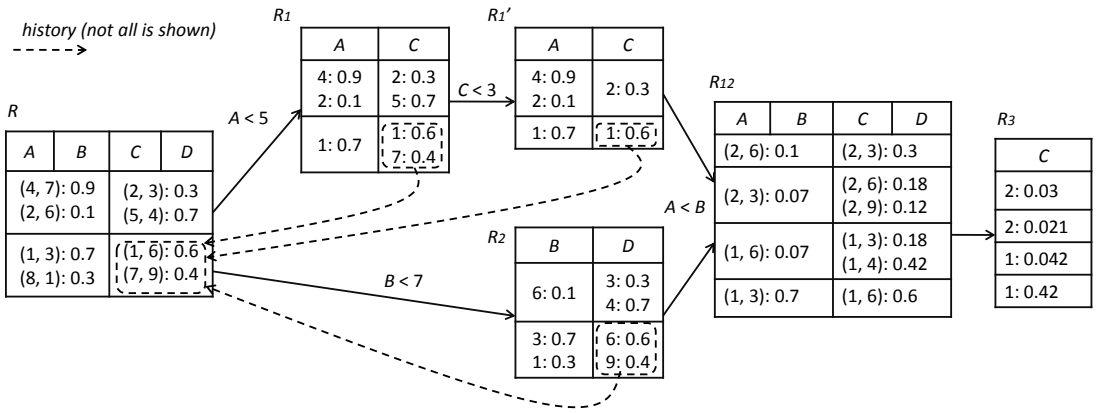

Fig. 4.1.: Running example (the tables and the query)

Example 4.1.1 *Consider a relation R, with four discrete uncertain attributes A, B, C and D, as shown in Fig. 4.1. Attributes A and B are jointly distributed, as are C and D. Each pair may represent, for example, two location coordinates, or values reported by different individuals or sensors. The example shows two tuples in R. The uncertainty in the first tuple is as follows: the values of A and B are* either 4 *and* 7*, respectively, with probability 0.9,* or 2 *and* 6*, respectively with probability 0.1; the values of C and D are* either 2 *and* 3 *with probability 0.3,* or 5 *and* 4 *with probability 0.7. The uncertainty in the second tuple is similar. Table* R_1 *and* R_2 *are derived from R as follows:* $R_1 = \pi_{A,C}(\sigma_{A<5}R)$*,* $R_2 = \pi_{B,D}(\sigma_{B<7}R)$*. The following query is performed over* R_1 *and* R_2*:*

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

as shown in Fig. 4.1 along with all intermediate results during the query evaluation. We explain below how this query is evaluated under the Orion model.

4.1.2 Probabilistic Threshold Query Optimization

We begin with a formal definition of the probabilistic threshold query (PTQ):

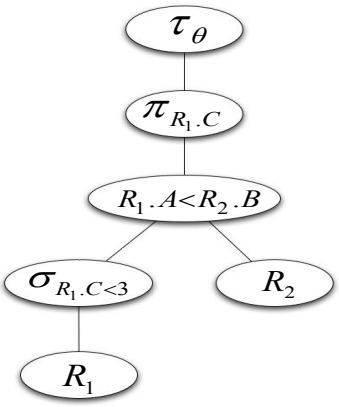

Fig. 4.2.: PTQ plan on the running example

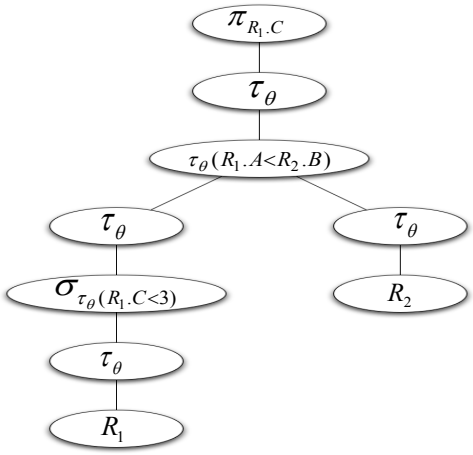

Fig. 4.3.: PTQ plan after optimizations

Definition 4.1.1 (The Probabilistic Threshold Query (PTQ)) *Given a probability threshold θ, and a regular query, a PTQ returns all tuples satisfying the query with tuple probabilities greater than or equal to θ.*

Example 4.1.2 *In Example 1.1.1, suppose the speed limit on Highway 101 is 70 miles per hour (*mph*), the local police want to find all speeding cars with probability at least 0.4. This is a PTQ where $\theta = 0.4$. To answer the query, we first find out all cars on Highway 101: Tuple 1 and Tuple 2, then compute their tuple probabilities after the selection 'Speed $>$ 70' is performed, which are 0.5*0.6=0.3 and 2/3*0.6=0.4, respectively. Note that the tuple probability is computed from the two dependency sets ({Speed}, {Make, Model}) after the selection on Speed "floors" out part of the uniform distribution where Speed ≤ 70. The result of this PTQ is Tuple 2.*

In this example, we take a two-stage approach for the PTQ execution: First we obtain the tuples satisfying the query (Tuple 1 and 2), then among the resulting tuples, we choose those whose probabilities meet the threshold (Tuple 2). We call the first stage "*evaluation stage*" and the second "*pruning stage*". However, for complicated queries, this direct approach can be very inefficient. As with other query

operators (e.g., selection, projection), we could perform significantly better if we could prune out tuples at early stages of the query evaluation based upon the probability threshold operator. This can be especially beneficial for uncertain data for which probability computations can be CPU-intensive.

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

Definition 4.1.2 (The Threshold Operator) *The threshold operator τ_θ when applied on an input relation, only retains those tuples with tuple probabilities greater than or equal to the threshold θ. Formally, we have*

$$\tau_\theta(R) = \{t | t \in R \wedge Pr(t) \quad \theta\}$$

where t is a tuple and R is a relation.

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

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

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

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

The query plan can be viewed as a tree with the root being the last operation to perform. The threshold operator τ_θ sits at the root to filter out results that satisfy the query but do not meet the threshold requirement. This is illustrated in Fig. 4.2, which is the PTQ version of our running example (see Example 4.1.1). We can think of the PTQ optimization process as one that "trickles down" τ_θ along the tree so that unqualified tuples are pruned earlier at lower levels of the query plan tree.

Example 4.1.3 *As seen in Example 4.1.1, the original query in Fig. 4.1 before applying any threshold operators can be defined as*

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

Its PTQ version is to return all tuples with probabilities at least θ after the original query is executed. Notice that if we can place the threshold operator before the join and successfully prune tuples from either R_1 or R_2, the expensive join execution will be much more efficient since there are less tuples to evaluate for the join predicate. We will later prove that such pruning does not prune away any potential result and will come back to the example for a more detailed discussion in Section 4.2.

In summary, the task of the PTQ optimization is to decide where to put τ_θ in the query plan so that the correctness of the query result is preserved while the pruning of unqualified tuples is maximized. Before considering complex queries with multiple operators, we first study the optimization problem for individual operators in Section 4.2.

4.2 Optimization Rules

In this section, we give the optimization rules for PTQ based on selection (σ), projection (π), Cartesian product ($\times$) and join ($\bowtie$). The idea is to perform the threshold pruning at earlier stages during the query execution so that tuples that cannot meet the threshold can be discarded without further evaluation. Note that among the five basic operations for relational algebra, we only discuss three (selection,

projection and Cartesian product), because the set difference and union require a clear definition for equality of two tuples with uncertain attributes, which is left for future work.

4.2.1 General Rules

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

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

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

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

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

Next we introduce a theorem upon which many of our optimization rules are built. It lays a solid foundation for ensuring the correctness of many PTQ optimization rules

that we present later in the chapter. It also ensures the safety to avoid tracking histories or dependencies between attribute sets in many cases, which greatly simplifies the PTQ optimization for uncertain data with arbitrary dependencies.

Theorem 4.2.1 *Given two arbitrary sets of attributes that are disjoint, the probability of the cross product of the two sets is no more than the probability of either set. Formally, let S_1 and S_2 be two arbitrary attribute value sets in tuple t_1 and t_2 respectively (t_1 and t_2 can come from different relations). Let $Pr(S_1, S_2)$ be the probability of the cross product of the two sets and $Pr(S_1)$, $Pr(S_2)$ be the probability of S_1 and S_2 respectively. Then we have:*
$Pr(S_1, S_2) \leq \min(Pr(S_1), Pr(S_2))$.

Proof Our proof consists of a proof for $Pr(S_1, S_2) \leq Pr(S_1)$ and a proof for $Pr(S_1, S_2) \leq Pr(S_2)$, from which we deduce that $Pr(S_1, S_2) \leq \min(Pr(S_1), Pr(S_2))$. For simplicity of notations, we write $Pr(S_{12})$ instead of $Pr(S_1, S_2)$ in our proof. We only show the proof for $Pr(S_{12}) \leq Pr(S_1)$ below, as the proof for $Pr(S_{12}) \leq Pr(S_2)$ is exactly the same.

Without loss of generality, we partition S_i ($i = 1$ or 2) into two parts[1]:

- *Historically dependent attributes:* $C_j, 1 \leq j \leq m$, where $C_j = N_j \cap S_{12}$ and N_j is the common ancestor of S_1 and S_2 (i.e., $N_j \in \Lambda(S_1) \cap \Lambda(S_2)$, the intersection of the histories of S_1 and S_2). Thus C_j is the set of attributes that the ancestor N_j shares with either S_1 or S_2.

- *Historically independent attributes:* $D_i = S_i - \bigcup C_j$ is the set of attributes in S_i that are not shared with any common ancestor of S_1 and S_2.

Let X_S^t be the random variable for an attribute set S in t. Let x_S^t be an instance of X_S^t. If t is omitted in X_S^t (i.e., X_S), we mean the random variable for the attribute

[1]See Section 1.1 for definitions of historical dependency

set in S. Particularly, if S refers to C_j, we interpret $t.C_j$ as the common attribute set between tuple t and the ancestor C_j. Let $f(x_S^t)$ be the *pdf* of x_S^t, then we have:

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

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

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

$$\begin{aligned} Pr(S_{12}) &= \int f(x_{S_{12}}) dx_{S_{12}} \\ &= \#\int_{f(x_{S_i}^{t_i}) \neq 0} f(x_{D_1}^{t_1}) f(x_{D_2}^{t_2}) \prod_{j=1}^{m} f(x_{C_j}) \\ &= \int f(x_{D_1}^{t_1}) \int f(x_{D_2}^{t_2}) \prod_{j=1}^{m} \int f(x_{C_j}) \\ &= Pr(t_1.D_1) Pr(t_2.D_2) \prod_{j=1}^{m} Pr(C_j) \end{aligned} \tag{4.1}$$

Note that (4.1) $\leq Pr(t_1.D_1) \prod_{j=1}^{m} Pr(C_j)$.

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

$$\begin{aligned} Pr(S_1) &= Pr'(t_1.D_1) \prod_{j=1}^{m} Pr'(t_1.C_j) \\ &= Pr'(t_1.D_1) \prod_{j=1}^{m} Pr'(C_j) \end{aligned} \tag{4.2}$$

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

Comparing (4.1) with (4.2), we notice that $Pr(t_1.D_1) \leq Pr'(t_1.D_1)$ and $Pr(C_j) \leq\leftarrow Pr'(C_j)$ due to more floors when computing $Pr(S_{12})$ than computing $Pr(S_1)$ (when computing $Pr(S_{12})$, we need to consider floors resulting from selection predicates to

obtain both S_1 and S_2 while we only consider floors to get S_1 when computing $Pr(S_1)$, i.e., the former considers either $f(x_{S_1}^{t_1}) = 0$ or $f(x_{S_2}^{t_2}) = 0$ while the latter considers only $f(x_{S_1}^{t_1}) = 0$), we have:

$$(4.1) \leq Pr(t_1.D_1) \prod_{j=1}^{m} Pr(C_j) \leq Pr'(t_1.D_1) \prod_{j=1}^{m} Pr'(C_j)$$

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

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

Theorem 4.2.1 empowers us to avoid tracking histories and dependencies between attribute sets when pruning – we can always prune according to either S_1 or S_2 regardless of whether the two sets are correlated or how they are correlated.

Example 4.2.2 *As a concrete example to illustrate the use of Theorem 4.2.1, let us revisit our running example in Fig. 4.1 and 4.2. We have explained how to prune intuitively in Example 4.1.3 without giving the reason why the pruning is correct, now let us examine the pruning more closely. The reason that we can discard the first tuple in R_1' (call it t_{11}') as well as the first tuple in R_2 (call it t_{21}), hence avoiding the join operation that would have otherwise produced Tuple 1 through Tuple 3 in R_{12}, is that by Theorem 4.2.1, the probability of any tuple t in R_{12} containing either t_{11}' or t_{21} must not exceed the probability of t_{11}' or t_{21} themselves (both below the threshold). Since projections do not change the tuple probabilities (see Section 5.2.1 for details), the tuples in R_3 projected from Tuple 1 to Tuple 3 in R_{12} also have probabilities below the threshold, hence cannot be in the final results. Note that in pruning Tuples 1 to 3 in R_{12}, we do not need to worry about the dependency between attributes A and B or that between C and D. Using Theorem 4.2.1, we simply prune based on the probabilities of t_{11}' and t_{21}.*

From Theorem 4.2.1, we obtain the following corollary:

Corollary 4.2.3 *Given a tuple t and any set of attributes $t.S \subseteq t$, we have $Pr(t) \leq\leftarrow Pr(t.S)$.*

Proof The tuple probability $Pr(t) = Pr(t.S, t.S')$ where $t.S' \cup t.S = t$ and $t.S' \cap\leftarrow t.S = \emptyset$. From Theorem 4.2.1, we know that $Pr(t) \leq\leftarrow \min(Pr(t.S), Pr(t.S')) \leq\leftarrow Pr(t.S)$. ■

The optimization rule below can be deduced immediately from Corollary 4.2.3:

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

In other words, if there exists any dependency set with probability below θ, we can immediately prune the tuple away knowing that there is no way for the whole tuple to have a probability that meets the threshold.

4.2.2 Rules for Selection, Projection and Join

We now present our optimization rules specifically for selection, projection and join.

Selection

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

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

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

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

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

The formula is based on the fact that the only difference between $Pr(t')$ and $Pr(t)$ results from the requirement that c should hold. From Corollary 4.2.3, we have $Pr(t) \leq Pr(t.S_c)$, hence $Pr(t') \leq Pr(c) < \theta$. ■

From Rule 3, we obtain the following equivalence:

$$\tau_\theta(\sigma_c(R)) = \tau_\theta(\sigma_{\tau_\theta(c)}(R))$$

where $\tau_\theta(c)$ means applying the threshold operator to $Pr(c)$ for relation R.

Now we seek further optimizations based on the form of the predicate c:

Simple Predicate

A simple predicate c involves at least one uncertain attribute (e.g. U, U') and has one of the following forms: i) $U\ op\ k$ ii) $U\ op\ A$ iii) $U\ op\ U'$, where A is a certain attribute, k is a constant number and op is a comparison operation. For a simple predicate c, we use Optimization Rule 3 for pruning. To further improve the efficiency of pruning, we can build a Probabilistic Threshold Index (PTI) on the uncertain attribute U [29]. PTI is built based on the concept "x-bound" proposed by Cheng et al. [39], which is a probability bound maintained in the nodes of an R-tree based index to facilitate pruning for probabilistic threshold range queries. Such an index exploits both the range predicate over an attribute and the threshold predicate over the probability of the attribute within the range.

Complex Predicate

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

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

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

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

From Rule 4, we can deduce the equivalence:

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

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

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

Proof We first compute $Pr(t')$ from $Pr(t)$:

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

Since $Pr(t) \leq Pr(t.S_{c_1})$ from Corollary 4.2.3, we have:

$$Pr(t') \leq Pr(t.S_{c_1}) - Pr(c_1) \leq 1 - Pr(c_1) \tag{4.3}$$

If either $Pr(c_1) > 1 - \theta$ or $Pr(c_1) > Pr(t.S_{c_1}) - \theta$ holds, then (4.3) $< \theta$. Hence $Pr(t') < \theta$. ■

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

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

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

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

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

Example 4.2.4 *Suppose a tuple t in relation R with two uncertain attributes a {2: 0.1, 4: 0.2} and b {1: 0.5, 2: 0.1}. Consider PTQ* $\tau_{0.2}(\sigma_{c_1 \vee c_2}(R))$ *where* c_1 *is* $a > 3$ *and* c_2 *is* $b < 2$. *Since* $Pr(a, b) = 0.3 \times 0.6 = 0.18 < 0.2$, *we can immediately discard t without evaluating the predicates.*

The corollary below follows immediately from Rule 6 and Theorem 4.2.1:

Corollary 4.2.5 *Given a PTQ* $\tau_\theta(\sigma_{c_1 \vee c_2}(R))$ *and a tuple t,* $Pr(t.S_{c_1}) < \theta \vee Pr(t.S_{c_2}) < \theta \Rightarrow Pr(t') < \theta$.

Proof From Theorem 4.2.1, we know that

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

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

Projection

For projections $\pi_{\bar{A}}$, where $\bar{A}$ is the set of attributes to be projected, let $Pr(t)$ and $Pr(t')$ be the tuple probability of t before and after projection, we introduce the lemma below, which comes from [1] and is also clear from the possible world semantics:

Lemma 4.2.6 *For a given tuple t, any projection on t does not change the tuple probability.*

From Optimization Rule 1 and Lemma 4.2.6, we can easily deduce the following optimization rule for projections:

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

The above rule can be regarded as a special case of Rule 1, where the outer τ_θ is no longer needed since the projection does not change the tuple probability.

Cartesian Product

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

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

Proof From Theorem 4.2.1, we know that

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

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

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

Join

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

4.2.3 Plan Optimization

Now that we have optimization rules for individual operators, we can apply them to a given query plan with combined operators. Let us first review our running example (let $\theta = 0.4$) in Fig. 4.2. The query is:

$$\tau_{0.4}(\pi_{R_1.C}((\sigma_{R_1.C<3}(R_1)) \bowtie_{R_1.A<R_2.B} R_2))$$

Let predicate c_1 be $R_1.C < 3$ and c_{12} be $R_1.A < R_2.B$. Using Optimization Rule 1, 3, 7 and 8, we "trickle down" $\tau_{0.4}$ along the query plan tree and equivalently, we execute the following query instead:

$$\pi_{R_1.C}(\tau_{0.4}(\sigma_{\tau_{0.4}(c_{12})}(\tau_{0.4}(\sigma_{\tau_{0.4}(c_1)}(\tau_{0.4}(R_1))) \times \tau_{0.4}(R_2)))) \tag{4.4}$$

Note that we execute $\tau_{0.4}$ from inside out ($\tau_{0.4}$s in inner parentheses are executed first): If a tuple can be pruned with some inner $\tau_{0.4}$, we can discard the tuple right away without completing the whole evaluation. The query plan tree corresponding to 4.4 is shown in Fig. 4.3. We show below how various optimization rules work together in pruning for a complicated PTQ with selections, projections and joins all present:

In (4.4), though we cannot prune anything by executing $\tau_{0.4}(R_1)$ (See Fig. 4.1: The two tuples of R_1 have probabilities 1 and 0.7 each), we can use $\tau_{0.4}(F_1)$ to prune the first tuple of R_1' away, as a result of Optimization Rule 3. We can also prune the first tuple of R_2 away with Rule 1 by executing $\tau_{0.4}(R_2)$. The pruning of tuples in R_1' and R_2 before joining them is itself a result from applying Rule 8, which leaves us only one join operation to do: Joining Tuple 2 from R_1' and Tuple 2 from R_2. Tracing back to their history in Fig. 4.1, we correctly compute the joining tuple to be:

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

Its tuple probability is $0.7 * 0.6 = 0.42 > 0.4$. According to Optimization Rule 7, the projection does not change the tuple probability. We hence return the projected tuple as the final answer (Tuple 4 of R_3 in Fig. 4.1).

As we can see from Section 4.2, our contributions lie in providing a new way of optimizing probabilistic threshold queries that is general, and similar to the traditional approach for optimization. The merit of the approach is not in the complexity of the proposed rules, rather in their simplicity and easy applicability while ensuring correct evaluation with respect to possible world semantics.

Generally, given a query plan P of a PTQ in which the threshold is placed at the end of the query, our goal for PTQ optimization is to find an equivalent query plan P' of P such that we can prune as many tuples as possible by leveraging the threshold (i.e., applying the threshold operator) during the query evaluation while keeping the total cost low. Our approach here is to start with a plan P that is guaranteed to be safe (through the use of histories and dependency sets) and then apply optimization rules to P to generate equivalent plans that are also all guaranteed to be safe (the rules ensure correct evaluation). Since there are usually multiple optimization rules that are applicable for P, we can generate multiple equivalent plans by applying different sets of rules or by applying the same set of rules in different order. The query optimizer then chooses which alternative plan to use based on cost estimation, a process similar to that of the current traditional database optimizers.

4.2.4 Experimental Evaluation

The goal of our experiments is to validate the effectiveness of our optimization rules proposed in Section 4.2. We evaluate the effectiveness of the rules on both synthetic and real data sets.

Table 4.1: Sensor data set schema

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

Data Sets

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

The real data set comes from a sensor application that monitors the movement of people within a building using 802.11-based sensors that report approximate locations in real-time. Each tuple consists of a sensor ID (SID) that identifies each sensor, the time stamp (TS) of the measurement, and the measured location $(xpos, ypos)$. Due to the calibrated errors with the sensors, the positions are reported with uncertainty represented as Gaussian distributions around the reported locations. We use it as an example of continuous uncertain attributes. Table 4.1 shows the first 5 tuples in this sensor data set. The notation $prod(Gaus(\mu_1, \sigma_1^2), Gaus(\mu_2, \sigma_2^2))$ stands for the joint *pdf* of the Gaussian distributions of *xpos* and *ypos* (as introduced in [1]), where μ_1 and σ_1^2 are the mean and the variance of *xpos*, and μ_2 and σ_2^2 are the mean and the variance of *ypos*. The cumulative probability of the joint *pdf* is 1, hence the tuple probability is 1 for all tuples.

The synthetic data set that we generate is a simulation of the real sensor data set with *xpos* and *ypos*, having discrete uncertainty. We generate 100,000 tuples in total. Each tuple has a TupleID, along with *xpos* and *ypos* values that are jointly

distributed as one dependency set. The number of instances in the dependency set, k, is uniformly distributed between 1 and 10. The tuple probability of the synthetic data set, which equals the total probability of this dependency set (as TupleID is certain), is randomly generated from 0.001 to 1. The probabilities of the instances are generated randomly and sum up to this total probability. The values of both attributes *xpos* and *ypos* are in the range [1, 1000]. For each uncertain attribute in each tuple, we randomly pick a central point `center` in [1, 1000]. We also generate the `spread` of its instance values in the tuple that obeys Gaussian distribution with a mean 10 and variance 2, which roughly corresponds to 1% of the entire range. With the center and spread fixed, we can randomly generate the values of the k instances such that they are within the range [`center` - `spread` /2, `center` + `spread` /2].

Unless specified otherwise, the default value of the threshold is 0.4 for all experiments, and the default size of the real and synthetic data sets are 10,000 and 100,000 tuples each.

Query Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This is an example of a complicated query similar to our example query in Fig. 4.1. It uses several optimization rules: Optimization Rule 1, 3, 4, 7 and 8. These rules work together to ensure that the threshold operator is pushed down the query plan tree as far as possible so that unqualified tuples from either table T_1 or T_2 can be pruned away before the join and unqualified tuples from the joined table can also be discarded promptly.

Experimental Results

Our experiments compare the optimized PTQ execution with the base case, i.e., the naïve approach that does not use any optimization rules that we proposed earlier. We call them "optim" and "naïve" respectively.

We now show how our optimization rules are actually written in the form of SQL queries. Consider query Q7. If the threshold is p, this query in the naïve form is written as:

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

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

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

">? (x,p)" and "<? (x,p)" are operators defined in Orion which calls PTI index for value x and threshold p. $mass$ is a function that calculates the probability mass of an uncertain variable. The function $floor$ is the floor operation we introduced in Section 1.1, which zeroes out part of the uncertain attribute's pdf that does not satisfy the predicate. We see that TT_1 in the optim query is defined using Rule 3, TT_2 is defined according to Rule 3 and 4. And the join is done according to Rule 3, 4 and 8.

We evaluate all queries from Q1 through Q7 on both real and synthetic data sets in the following aspects:

Effect of Data Set Size

Fig. 4.4 and Fig. 4.5 show the effect of data set size on the run time of selection query Q1 and Q3. The threshold is fixed at 0.4. Due to the small size of the real sensor data set we have, we choose to perform this test on the synthetic data set alone. Let the synthetic data set we generated be T. For the join query Q6, we generate two tables T_1 and T_2 from T. T_1 contains all tuples from T whose $xpos$ is greater than 300 while T_2 contains all tuples from T whose $ypos$ is less than 600. We record the time cost for running optim against naïve. The result in Fig. 4.6 (the data set size here is the size of the table after the join) shows that Optimization Rule 8 significantly contributes to better performance of Q6 in terms of run time.

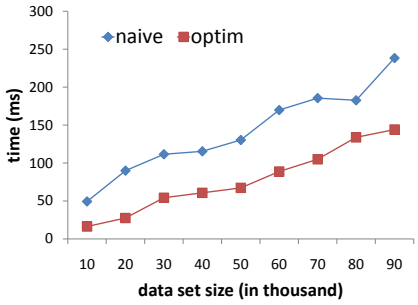

Fig. 4.4.: Effect of data size on Q1

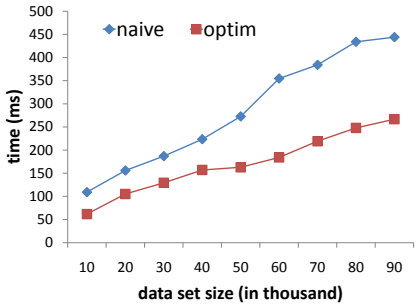

Fig. 4.5.: Effect of data size on Q3

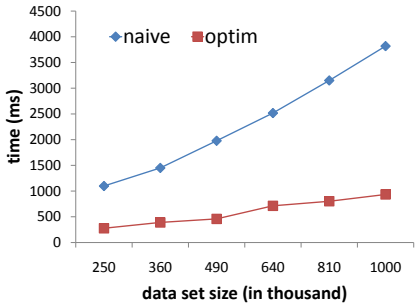

Fig. 4.6.: Effect of data size on Q6

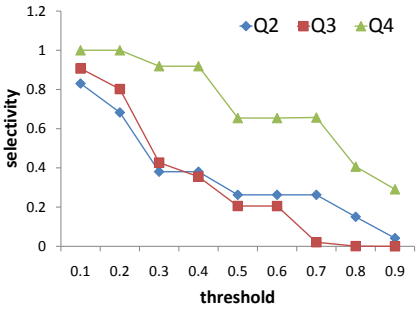

Fig. 4.7.: Query selectivity of Q2, Q3, Q4

Effect of Threshold

Fig. 4.7 gives a comparison between the selectivity of different thresholds for Q2, Q3 and Q4 on the sensor data set. All three queries are selections with predicates. We define selectivity as the ratio of the size of the PTQ result set and the size of the original data set. With an increasing threshold, all queries observe a consistent decrease in the selectivity of query results, as more tuples become unqualified for the threshold. The threshold also affects the run time of the query, as shown in Fig. 4.10 on the sensor data set for the same queries. We compute the ratio of naïve's run time and our optim's run time for thresholds from 0.1 to 0.9 (called *naïve-optim ratio*).

We perform the same experiment for simple selection and projection (i.e. Q1 and Q5) on the sensor data set, and the result is shown in Fig. 4.11. Fig. 4.12 shows the run time of naïve versus that of optim (threshold fixed at 0.6). For join operations,

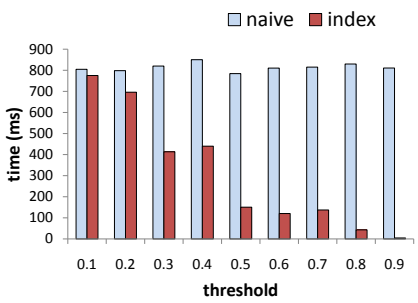

Fig. 4.8.: For Q6 on sensor data

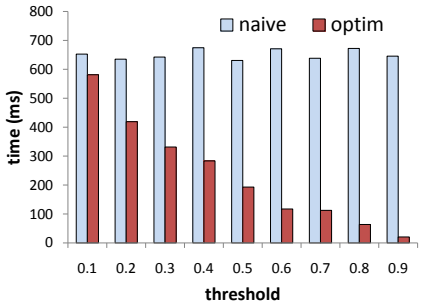

Fig. 4.9.: For Q6 on synthetic data

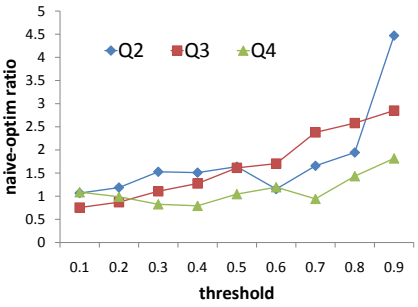

Fig. 4.10.: Naive-optim ratio for Q2-4

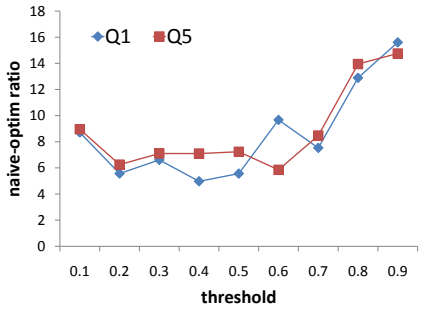

Fig. 4.11.: Naive-optim ratio for Q1, Q5

we plot the run time of optim and naïve on the sensor data set and the synthetic data set in Fig. 4.8 and Fig. 4.9 respectively. An interesting observation is that the threshold does not really affect the run time of naïve while it has dramatic impact on the run time of optim.

Effect of Optimization Rules

We now evaluate a more complicated SQL query that combines all the above operations in one query and leverages multiple optimization rules for faster query execution.

We use Q7 as an example of such queries, which benefits from Optimization Rule 1, 3, 4, 7 and 8. In Fig. 4.13, we show the pruning percentage (number of join tuples pruned over the total number of join tuples) by applying Rule 1, 3, 4 and 8 separately. Note that here we do not measure the pruning percentage for Rule 7,

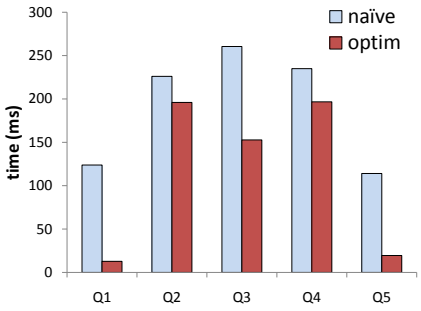

Fig. 4.12.: Run time of naive and optim

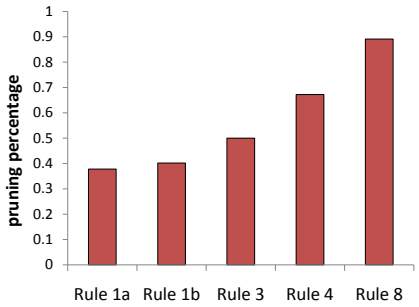

Fig. 4.13.: Effect of pruning by optimization rules

since Rule 7 is equivalent to Rule 1 in terms of pruning (projections do not change the tuple probability). While the first three rules are measured individually, Rule 8 is actually a combination of all these rules. Also, for Rule 1, we measure the pruning percentage for both T_1 and T_2. To distinguish the two, we call the first Rule 1a and the second Rule 1b. As we can see from Fig. 4.13, Rule 8 has the most powerful pruning capability, discarding almost 90% of all tuples due to its high selectivity from both T_1 and T_2.

4.3 Improving Optimization Through Threshold Estimation

So far we have discussed the optimization for threshold SPJ queries under the Orion model, where the optimization is achieved mainly through pushing down the threshold operator along the query plan to prune unqualified tuples away as early as possible. During this process, the original threshold on the query result is "trickled" down the query plan without change: At any stage of query execution, we apply the appropriate optimization rules to prune out tuples of intermediate results whose probabilities are below the same threshold.

However, pruning using the original threshold θ may turn out to be too conservative: Tuples that pass this pruning at an earlier stage of query execution may still fail to meet the minimum threshold requirement of θ at the end. In the rest of the section, we will show that it may be possible and more desirable to push down a threshold greater than θ for more aggressive pruning.

4.3.1 Motivation

We use the example below to illustrate how pruning for SPJ queries can be done more efficiently by varying the threshold during the query evaluation.

T in Table 4.2 has three uncertain attributes A, B, C and two dependency sets $\{A\}, \{B, C\}$. A has continuous uncertainty modeled as uniform distributions while B, C have discrete uncertainty and are jointly distributed. The threshold query is: $\tau_{0.4}(\sigma_{A\geq 5}T)$, where 0.4 is the threshold and $\tau_{0.4}$ is the *threshold operator* [27]. From [27] we know that equivalently, the above threshold query can be evaluated as $\tau_{0.4}(\sigma_{\tau_{0.4}(A\geq 5)}(\tau_{0.4}T))$ where we apply the threshold operator multiple times for early pruning.

First we observe that no tuple can be pruned by applying $\tau_{0.4}$, as all tuple probabilities are at least 0.4. For example, Tuple 1's probability is $1 * (0.4 + 0.1) = 0.5$. Hence $\tau_{0.4}T = T$. We need to further examine $\tau_{0.4}(A \quad 5)$ for possible pruning.

Table 4.2: An uncertain table T

A	B	C
Uniform(0, 10)	(0, 1): 0.4 (0, 2): 0.1	
Uniform(2, 17)	(1, 3): 0.2 (1, 5): 0.3	
Uniform(5, 36)	(1, 3): 0.1 (2, 2): 0.3	
Uniform(2, 7)	(3, 5): 0.5	

We can compute the maximum cumulative probability for dependency set $\{B, C\}$ across tuples as: $\max\{0.4 + 0.1, 0.2 + 0.3, 0.1 + 0.3, 0.5\} = 0.5$. If we had used the same threshold 0.4 on the predicate $A > 5$ (i.e., $\tau_{0.4}(A \quad 5)$), we would have missed out the opportunity to discard tuples with $Pr(A \quad 5) \quad 0.4$ that fail to yield a final tuple probability that meets the 0.4 threshold. In fact, given that the maximum probability for $\{B, C\}$ (denoted as $Pr_{max}(\{B, C\})$) is 0.5, $Pr(A \quad 5)$ should be at least $0.4/0.5 = 0.8$ for the result tuple to pass the threshold requirement. Therefore, we can safely prune all tuples except Tuple 3 with the enhanced 0.8 threshold while no single tuple could have been pruned if we had kept the same threshold 0.4. Therefore, the increased new threshold 0.8 is a more accurate lower bound on $Pr(A \quad 5)$.

Likewise, if we know the maximum probability for $A \quad 5$ (denoted as $Pr_{max}(A \quad 5)$) to hold across all tuples, we can bound the threshold for the probability of dependency set $\{B, C\}$: We use $\theta / Pr_{max}(A \quad 5)$ to prune out tuples with cumulative probabilities of dependency set $\{B, C\}$ under this new threshold. Furthermore, if we also know $Pr_{max}(\{B, C\})$, we can check whether $Pr_{max}(\{B, C\}) < \theta / Pr_{max}(A \quad 5)$ (where θ is the original threshold). If so, we know immediately that no tuple meets the threshold requirement.

4.3.2 Threshold Estimation

We vary the threshold θ as it "trickles" down the query plan tree by estimating the "real" threshold to use when applying optimization rules. This new threshold is the minimum probability threshold needed for the final result tuple to have a probability θ. Note that the new threshold is employed to prune unqualified tuples rather than to guarantee the finding of final results.

To estimate the threshold for pruning, we maintain the maximum cumulative probability over the joint pdf of each dependency set across all tuples, which is denoted as $\Pr_{\max}(\quad)$, where $\quad$ is the dependency set. There are several ways in which this information can be leveraged to improve pruning:

First of all, we know that the probability of a tuple is the product of the cumulative probabilities of all its dependency sets. Therefore, as long as one $\Pr_{\max}(\quad) < \theta$, there is no chance for any tuple in this uncertain table to be a query result that meets the desired threshold of θ, regardless of the SPJ query issued. Hence we completely bypass the query evaluation.

Second, for queries with selections, we can use $\Pr_{\max}(\quad)$ to increase the threshold to be applied to the selection predicate, as illustrated in the example of Table 4.2. Let the attributes in the selection predicate be from $\quad_1, ..., \quad_i$ where $1 \leq i \leq k$ and k is the total number of dependency sets in the table. Then we can compute the new threshold θ' to be applied to the selection as: $\theta' = \theta / \prod_{i+1}^{k} \Pr_{\max}(\quad)$.

In addition, as the example of Table 4.2 suggests, the other way round works too for selection queries: If we know the maximum probability for a selection predicate to hold across all tuples, we can also estimate the new threshold on the rest of dependency sets and see if the product of the maximum cumulative probabilities of these dependency sets meets the new threshold.

5. THRESHOLD SPJ QUERIES WITH DUPLICATE ELIMINATION

While the previous chapter studies threshold select-project-join (SPJ) query optimization under the Orion model, this chapter discusses the threshold SPJ query optimization when duplicate elimination is allowed. Recall that the Orion model does not support duplicate elimination. This is mainly due to the existence of continuous uncertainty in the Orion model, for which the semantics of duplicates is not clear. Therefore, to allow duplicate elimination in our threshold SPJ queries, we limit the uncertainty in data to discrete uncertainty. To make this discrete uncertainty model as general as possible, we adopt the general tuple uncertainty model introduced in Section 1.1.2, which allows arbitrary dependencies between tuples. Many current uncertain models can be converted to this general model, which we will discuss in Section 5.1.1.

Unlike the Orion model where tuples are essentially joint distributions of all dependency sets [1] and uncertainty can come from attributes, under the general tuple uncertainty model, each tuple is in effect a "valuation" of all attributes in the table: Let $A_1, A_2, ..., A_k$ be k attributes in an uncertain table T, then each tuple t in T is a valuation $v = \{A_1 = a_1, A_2 = a_2, ..., A_k = a_k\}$ where $a_i (1 \leq i \leq k)$ is a constant. Such a valuation is associated with a probability that it occurs, i.e., the probability that the tuple t exists, denoted as $\Pr(t)$. We later refer to v as the tuple value of t, denoted as $t = v$. An SPJ query with threshold θ returns tuples that satisfy the SPJ query with probabilities θ. It is important to know what tuples mean for different models in order to understand what we are thresholding on.

We design new threshold SPJ query optimization schemes that can be applied to the general tuple uncertainty model, as well as pruning rules and techniques to specifically handle duplicate elimination, which is not considered in Chapter 4. We

introduce a new query operator *dedup*, denoted as δ (see Definition 5.1.2), as an addition to the set of standard relational algebra operators. The introduction of dedup creates disjunctive relationship between tuples in addition to their initial dependencies. The biggest challenge now is to push the threshold through a dedup operator in the query: We can no longer push the same threshold down the query plan without possibly pruning away potential results – duplicate tuples whose individual probabilities are all below the threshold could still pass a dedup operator if the probability of their disjunction (the dedup result) exceeds the threshold.

While the dedup operator calls for new pruning rules that vary the threshold for more accurate pruning, we show that joins may also take advantage of such a varying threshold to improve pruning in Section 5.2.1. Our pruning rules and algorithms in general propose the threshold to use for each query operator that maximizes pruning while maintaining high precision and recall for the entire query.

Intuitively, there are two aspects we should consider for threshold query optimization:

- The query. By analyzing the query, we know what operators are used at what stages in the query plan. Such information could be leveraged for choosing the right set of pruning rules as well as the right pruning algorithms to estimate the threshold.

- The data. The more we know about the data, the more accurate the threshold estimate could be. This includes knowledge from the dependencies between tuples to distributions of attribute values across tuples.

We discuss in detail how the query and the data can help us design good pruning rules in Section 5.2, based on which pruning techniques are proposed in Section 5.3. We finally present our pruning schemes in Section 5.4.

5.1 Problem Definition

5.1.1 General Tuple Uncertainty Model

We formally define the general tuple uncertainty model mentioned in Section 1.1.2 as follows: Each tuple t in a probabilistic database $\mathcal{DB}^{\mathcal{P}}$ is associated with a probability $\Pr(t)$ that the tuple appears and a boolean expression $\phi(t)$ (referred to as "lineage" in [11]) to capture the dependencies between base tuples in generating t. We assign a unique random variable to each tuple in $\mathcal{DB}^{\mathcal{P}}$. The probability distribution of a random variable r for a tuple t is discrete: Either $r = true$ with probability $\Pr(t)$ or $r = false$ with probability $1 - \Pr(t)$, i.e., random variable r represents the atomic event that tuple t appears. We write r and $\neg r$ as shorthand for the events $r = true$ and $r = false$. For a base tuple t in $\mathcal{DB}^{\mathcal{P}}$ (as opposed to a tuple generated from a query on $\mathcal{DB}^{\mathcal{P}}$), its lineage is simply r. An instance of the above probabilistic database $\mathcal{DB}^{\mathcal{P}}$ is a truth assignment of the random variables for all tuples in $\mathcal{DB}^{\mathcal{P}}$ where the assignment is allowed by the dependencies. For example, if tuples t_1 and t_2 are mutually exclusive, the probability of the assignment $r_1 = true$ and $r_2 = true$ is zero, i.e., it is not a possible instance of $\mathcal{DB}^{\mathcal{P}}$. When the reference is clear from the context, we use t to represent both the tuple and its corresponding random variable.

Note that we choose lineage as the mechanism that the model adopts to capture tuple dependencies only for simple exposition and experimentation. As pointed out earlier in Section 1.1.2, many other mechanisms can be used to capture dependencies that are either inherent in the data or generated during query evaluation. Since we do not assume knowledge about any dependency between tuples at the beginning of a query, we design applicable pruning rules regardless of the dependencies. However, if the exact dependencies are known before the query, we can leverage them for more accurate pruning.

The general tuple uncertainty model is a simple but powerful model, as many existing data models with discrete uncertainty can be converted to this model. The independent tuple model and the x-tuple model are trivially convertible to the general

model. For others that are more complicated, such as MayBMS [6] and BayesStore [5], we can first list out all possible combinations of values across all attributes (i.e., all possible tuples) in each uncertain table based on the original uncertain table representation (usually more succinct than the one used in the general model). At the same time we maintain the dependency information for each tuple obtained above.We replace the original uncertain table representation using the new representation where uncertainty is only at the tuple level. We then inherit the existing mechanisms for tracking dependencies, such as c-tables for MayBMS [6] and Bayesian networks for BayesStore [5].

5.1.2 Threshold SPJ Query With Dedup

We now formally define the threshold SPJ query under the general uncertainty model that allows duplicate elimination:

Definition 5.1.1 *Given a probabilistic database $\mathcal{DB}^{\mathcal{P}}$ under the general tuple uncertainty model, a threshold θ, a SQL query Q with selection, projection, join and duplicate elimination, the threshold SPJ query Q_θ returns all tuples in $\mathcal{DB}^{\mathcal{P}}$ that satisfy Q with probability no less than θ.*

Note that duplicate elimination is allowed, but not enforced. Therefore, the query needs to explicitly use the *dedup* operator for duplicate elimination.

Definition 5.1.2 *Given a relation R in a probabilistic database $\mathcal{DB}^{\mathcal{P}}$, the dedup operator δ on R, denoted as $\delta(R)$, returns all tuples t in R with distinct values such that if t is the only tuple with value v, t is returned with lineage unchanged; otherwise, let $t_1, t_2, ..., t_k$ be k tuples with value v, then a single tuple $t = v$ is returned with new lineage $\bigvee_{i=1}^{k} \phi(t_i)$ and new probability $\Pr(t) = \Pr(\phi(t))$.*

Fig. 5.1 shows an example of a table R with three attributes and four tuples. Each tuple's probability and lineage are shown in separate columns. We assume that

A	B	C	probability	lineage
5	1	3	0.8	t_1
2	4	2	0.4	t_2
2	1	3	0.2	t_3
9	3	6	0.3	t_4

B	C	lineage
1	3	$t_1 \bigvee t_3$

Fig. 5.1.: Query with dedup under general tuple uncertainty model

all tuples are independent from each other. Initially the lineage of each tuple is set to its own event variable. The table on the right is the result after performing the following threshold query: $\tau_{0.5}(\delta(\pi_{B,C}(\sigma_{A<8}R)))$, where $\tau_{0.5}$ is the threshold operator with threshold $\theta = 0.5$. The selection predicate $A < 8$ disqualifies tuple t_4. The dedup operator after the projection leaves two candidates: $\{B = 1, C = 3\}$ from t_1, t_3 and $\{B = 4, C = 2\}$ from t_2. However, only the former remains to be the final result, as its probability computed from the lineage passes the 0.5 threshold: $\Pr(t_1 \bigvee t_3) = \Pr(t_1) + \Pr(t_3) - \Pr(t_1) * \Pr(t_3) = 0.84 > 0.5$.

Computing the probability of a lineage where base tuples are independent is known to be #P-hard [10, 54]. Therefore, the goal of our threshold query optimization is to design pruning algorithms that can quickly prune tuples with probabilities less than the threshold to avoid computing the exact probabilities. The results after pruning should have high precision as well as recall: The pruning algorithms should not return many tuples that are incorrect results, nor should the algorithms prune many correct results away. The definitions for precision and recall are given below:

- precision = #correctly returned tuples / #all returned tuples
- recall = #correctly returned tuples / #all correct tuples

Specifically, given a query and its execution plan, we estimate the appropriate threshold to use for pruning tuples at each stage of the plan such that unqualified tuples can be pruned as early as possible. Unlike the optimization discussed in the

previous chapter, when we "trickle" the threshold down a query plan, we no longer use the same threshold from the beginning to the end: The threshold is allowed to change for different query operators to improve pruning performance. Below we propose our new pruning rules under the general tuple uncertainty model that support duplicate elimination for threshold SPJ queries.

5.2 Pruning Rules

We design our pruning rules for each query operator, i.e., selection (σ), projection (π), join ($\bowtie$) and duplicate elimination (δ), respectively. Same as in [27], we use the *threshold operator* – τ_θ to represent the step when a probability threshold θ is used to decide if a tuple should be retained or not: Only those with probabilities no less than θ (i.e., pass the threshold operator τ_θ) are retained. As we have mentioned earlier, duplicate elimination is explicit: It is treated as a separate query operator δ rather than being implicitly performed at the end of each query. Therefore, the pruning rules below proposed for selection, projection and join are all rules that apply without duplicate elimination implicitly performed at the end.

5.2.1 Selection, Projection and Join

Selection

The general tuple uncertainty model defined in Section 5.1.1 guarantees that there is a clear yes-or-no answer to the question: Does the tuple satisfy the selection predicate? On the contrary, for the attribute-level uncertainty model, a "tuple" may mean a joint distribution of all attributes in the table [27], hence *part* of the tuple may satisfy the selection predicate while the rest may not. Under the general tuple uncertainty model, however, a tuple is essentially a *valuation* of all attributes and is associated with a probability that such valuation appears, as we have mentioned earlier. Such valuation either satisfies the predicate or not. The following Rule 1

summarizes the above by stating that selection does not change the tuple probability: Thresholding on tuples after performing selection is the same as thresholding on tuples before the selection.

Rule 1. $\tau_\theta(\sigma_c R) = \sigma_c(\tau_\theta R)$ *where* c *is the selection predicate and* R *is the uncertain relation in a probabilistic database* DB^P *with tuple uncertainty.*

Given Rule 1, we can trickle the same threshold θ down a selection operator in a query plan, which enables us to prune at lower levels of the plan (i.e., at an earlier stage of the query evaluation). The lineage of a tuple after selection remains the same.

Projection

Same as selection, projection does not change the tuple probability. After projection (*without* implicitly performing duplicate elimination), the values of the projected attributes (the *new* tuple, denoted as t') are implicitly associated with the original tuple t where the values come from, hence $Pr(t') = Pr(t)$, $\phi(t') = \phi(t)$, where ϕ is the tuple lineage. The pruning rule for projection thus looks similar to that for selection:

Rule 2. $\tau_\theta(\pi_{\overrightarrow{A}} R) = \pi_{\overrightarrow{A}}(\tau_\theta R)$ *where* $\overrightarrow{A}$ *is the list of projected attributes.*

Join

Joining two tables R_1 and R_2 together creates conjunction between tuples from the two tables. Let t_1 and t_2 be two tuples that satisfy the join predicate and are from R_1 and R_2 respectively. Then the lineage of the join tuple $\phi(t_{12})$ should be updated to $\phi(t_1) \wedge \phi(t_2)$, as $t_{12} = t_1 \wedge t_2$. The probability of the new tuple t_{12} depends not only on $\Pr(t_1)$ and $\Pr(t_2)$, but also on the dependency between t_1 and t_2. For example, if the two tuples are mutually exclusive, then $\Pr(t_{12}) = 0$; if $t_1 \Rightarrow t_2$, then $\Pr(t_{12}) = \Pr(t_1)$; if they are independent, then $\Pr(t_{12}) = \Pr(t_1) * \Pr(t_2)$.

We propose the following rule for the join that holds regardless of dependencies between tuples from the two tables:

Rule 3. *$\tau_\theta(R_1 \bowtie_c R_2) = \tau_\theta(\tau_\theta R_1 \bowtie_c \tau_\theta R_2)$ where c is the join predicate.*

Proof Given a join tuple t_{12} from t_1 and t_2, its probability $\Pr(t_{12}) = \Pr(t_1 \wedge t_2) = \Pr(t_1) * \Pr(t_2|t_1) \leq \Pr(t_1)$. Similarly, we have $\Pr(t_{12}) \leq \Pr(t_2)$. Therefore, if either $\Pr(t_1) < \theta$ or $\Pr(t_2) < \theta$, $\Pr(t_{12}) < \theta$, i.e., we can apply the threshold operator τ_θ on R_1 and R_2 first before performing the join. ■

Rule 3 applies universally regardless of dependencies between the two tuples to be joined. However, if the join tables are independent from each other, we can leverage this knowledge for better pruning. The idea is essentially the same as that in Section 4.3 (although for a different setting) where the new threshold that filters through the join operator is boosted.

Rule 4. *If R_1 and R_2 are two uncertain relations that are independent from each other, then $\tau_\theta(R_1 \bowtie_c R_2) = \tau_\theta(\tau_{\theta/\Pr_{\max}(t_2)} R_1 \bowtie_c \tau_{\theta/\Pr_{\max}(t_1)} R_2)$ where t_i is any tuple in R_i and $\Pr_{\max}(t_i)$ is the maximum tuple probability of R_i ($i = \{1, 2\}$).*

Proof Given a join tuple t_{12} from t_1 and t_2, its probability $\Pr(t_{12}) = \Pr(t_1 \wedge t_2) = \Pr(t_1) * \Pr(t_2) \leq \Pr_{\max}(t_1) * \Pr(t_2)$. Therefore, if $\Pr(t_2) < \theta \not/ \Pr_{\max}(t_1)$, $\Pr(t_{12}) < \theta$. Likewise, we have $\Pr(t_2) < \theta \not/ \Pr_{\max}(t_1) \Rightarrow \Pr(t_{12}) < \theta$. Therefore, we can either use $\Pr_{\max}(t_1)$ to boost the threshold to be used on R_2 ($\theta / \Pr_{\max}(t_1)$ θ) or use $\Pr_{\max}(t_2)$ to boost the threshold to be used on R_1. ■

The following Rule 5 can be immediately deduced from Rule 4:

Rule 5. *If R_1 and R_2 are independent from each other and $\Pr_{\max}(t_1) * \Pr_{\max}(t_2) < \theta$, then no tuple satisfies the query: $\tau_\theta(R_1 \bowtie_c R_2)$.*

5.2.2 Duplicate Elimination

Recall that duplicate elimination (dedup) on a table creates disjunctions between tuples. Let $t_1, t_2, ..., t_k$ be k tuples in a table R with the same value v across all attributes in R. After dedup, the lineage of a resulting tuple t (with value v) generated from the k tuples is updated to the following: $\phi(t) = \bigvee_{i=1}^{k} \phi(t_i)$. It is obvious that

we cannot naïvely push down the same threshold through, otherwise we are likely to prune away potential results in situations where for all $1 \leq i \leq k, \Pr(t_i) < \theta$, but it still holds that $\Pr(t) = \Pr(\phi(t))$ θ.

On the other hand, for all tuples with unique values (i.e., $k = 1$), no disjunction is created. We can still use the same threshold for pruning.

To design a pruning rule for the dedup operator δ, it is important to know some information about the data itself such as the number of tuples with the same value (i.e., k) for each possible tuple value v in the table. We call k the *occurrence count* of value v. We can build a histogram on tuple values with bin size 1 to obtain this information. The pruning rule below is designed based on knowing the ks:

Rule 6. *Let v be the value of a tuple t in an uncertain relation R and k be the number of tuples in R with the same value v, then $\tau_\theta(\delta R) = \tau_\theta(\delta(\tau_{\theta'} R))$ where $\theta' = \theta/k$ for each tuple t being evaluated.*

Proof For tuples $t_1, ..., t_k$ in R with the same value v, the probability of value v after dedup (call it tuple t) is computed as: $\Pr(v) = \Pr(t) = \Pr(\bigvee_{i=1}^{k} t_i)$. We need to prove that by thresholding on $t_1, ..., t_k$ using $\theta' = \theta/k$ before dedup is performed, we will not prune away any potential result. Clearly, t will be pruned only if all t_is have $\Pr(t_i) < \theta/k$. In this case, $\Pr(t) = \Pr(\bigvee_{i=1}^{k} t_i) \leq \sum_{i=1}^{k} \Pr(t_i) < \theta$ holds, i.e., t is definitely not a result and should be pruned. Therefore, pruning using θ/k guarantees that no potential result will be dropped. ■

Note that as long as there exists a t_i $(1 \leq i \leq k)$ with $\Pr(t_i)$ θ/k, t will be returned as a result. The values of all t_is are the same (i.e., $\forall i, t_i = v = t$). Hence as long as one t_i passes the new threshold θ/k, t will be retained.

In Fig. 5.1, the result of the threshold query: $\tau_{0.5}(\delta(\pi_{B,C}(\sigma_{A<8} R)))$ is $\{B = 1, C = 3\}$, which comes from both t_1 and t_3. If we apply the above Rule 6 for pruning instead of computing the exact probability of the result tuple, we can see that the tuple count $k = 2$ for $\{B = 1, C = 3\}$ and the new threshold for pruning is $0.5/2 = 0.25$. t_1 passes the new threshold while t_3 fails. As a result, $\{B = 1, C = 3\}$ is still retained even though its lineage is now simply t_1 instead of $t_1 \bigvee t_3$.

If we have a histogram built on tuples in R but with a larger bin size (i.e., there are more than one value associated with the bin), then each bin b is associated with an *average occurrence count* k computed as follows: Let $v_1, v_2, ..., v_l$ be l values in b, we first compute the occurrence count k_i for each v_i as the number of tuples in R with the same value v_i, then we compute k as the average of all k_i's. Given the above histogram and a value v, to find out the occurrence count for v, we first locate the bin that the value v belongs to, and then use the corresponding count k for the bin (which is the average occurrence count of all values in the bin) as the count for v, which may not be its actual occurrence count. Obviously histograms with larger bins are not as accurate in describing the original data as those with bin size 1. Hence using a coarser histogram results in less accurate pruning with Rule 6. When the bin size reaches the total number of distinct tuple values in R, k becomes the average occurrence count for all values. In this case, θ' in Rule 6 becomes fixed as θ/k regardless of tuple values.

The histograms we have discussed so far store only the occurrence counts (ks) corresponding to their value bins. If, on the other hand, we store with each bin of value v the maximum probability of the k tuples $\max \Pr(t_i)$ as well as the sum of all k probabilities $\sum \Pr(t_i)$ in addition to the count k, we can have a better estimation of $\Pr(t)$ where $t = v$ after duplicate elimination: $\max \Pr(t_i) \leq \Pr(t) \leq \min(1, \sum \Pr(t_i))$ [55]. This probability range can be leveraged for pruning as summarized in Rule 7:

Rule 7. *For tuples $t_1, ..., t_k$ in R with the same value v, the probability of value v after dedup (call it tuple t) is in the range:* $[\max \Pr(t_i), \min(1, \sum \Pr(t_i))], 1 \leq i \leq k$: *If* $\max \Pr(t_i)$ θ, *t must be a result; If* $\sum \Pr(t_i) < \theta$, *t must not be a result, hence should be pruned.*

5.3 Pruning Techniques

In this section, we discuss pruning techniques specifically geared to the dedup and the join operators. As we have shown in Section 5.2, neither selection nor projection changes tuple probabilities. Hence pushing the same threshold down either operator

suffice. However, for joins and dedups, we need to be more careful in choosing the right threshold.

5.3.1 Range Partitioning for Joins

Recall that joins can change probabilities of the join results based on dependencies between the two tuples to be joined. Rule 3 of Section 5.2 shows that the probability range of a join tuple t_{12} from tuples t_1 and t_2 is $[0, \min(\Pr(t_1), \Pr(t_2))]$ regardless of dependencies between t_1 and t_2. While Rule 3 enables us to apply the same threshold on t_1 and t_2 first before performing the join, Rule 4 further increases the threshold to be applied to t_1 and t_2 under the independence condition. Below we propose a technique based on Rule 4 to further improve the performance of pruning for join queries, given the independence assumption of the join tables.

Let R_1 and R_2 be two uncertain tables to be joined that are independent from each other. In order to apply Rule 4, we maintain the statistics on the tuple probabilities of R_1 and R_2, namely, the maximum tuple probability $\Pr_{\max}(t_1)$ and $\Pr_{\max}(t_2)$ where t_1 and t_2 are tuples from R_1 and R_2 respectively. However, if both tables are huge, it is very likely that $\Pr_{\max}(t_i)$ is close to 1, hence using $\theta/\Pr_{\max}(t_i)$ as the new threshold for pruning tuples is not much more effective than using the original threshold θ for pruning – there can be many join tuples in the result set with actual probabilities less than θ. Alternatively, we could simply multiply the probabilities of all pairs of tuples that satisfy the join predicate to obtain the exact probabilities of join tuples and decide if any should be pruned. This naïve method, although guarantees to return correct results, requires performing a large amount of joins, many of which could have been avoided had we leveraged a threshold for early pruning.

To fully take advantage of Rule 4 in maximizing pruning, we propose the *range partitioning* technique for joins. We first partition the join tables R_1 and R_2 by the range of their tuple probabilities in linear time. Specifically, let c be the number of partitions for each table (c is a constant, $c > 0$), we scan the original table R_i and

partition it into a set of smaller tables R_{ij} $(0 \leq j < c)$ as follows: If a tuple t has a probability in range $(j/c, (j+1)/c]$, we assign it to R_{ij}. While partitioning, we also maintain the maximum tuple probability $\Pr_{\max}(t_{ij})$ for each R_{ij} $(\Pr_{\max}(t_{ij}) \leq (j+1)/c)$, which is no bigger than $\Pr_{\max}(t_i)$ due to the smaller size of R_{ij} compared with R_i. Such partitioning needs to be done only once for all queries. Given a join query with threshold θ, we first prune out all R_{ij}s whose maximum tuple probabilities are less than θ. Then for each remaining R_{1j}, we use a new threshold $\theta_{1j} = \theta/\Pr_{\max}(t_{1j})$ to prune out all $R_{2j'}$s with $\Pr_{\max}(t_{2j'}) < \theta_{1j}$ $(0 \leq j, j' < c)$. Then we perform the join between tuples in R_{1j} and tuples in the remaining $R_{2j'}$s whose probabilities meet θ_{1j}. Similarly, for each remaining $R_{2j'}$, we can also use a new threshold $\theta_{2j'} = \theta/\Pr_{\max}(t_{2j'})$ to prune out all R_{1j}s with $\Pr_{\max}(t_{1j}) < \theta_{2j'}$ and then perform the join between tuples in $R_{2j'}$ and tuples in the remaining R_{1j}s whose probabilities meet $\theta_{2j'}$.

The advantage of the above range partitioning technique is that it enables finer-grained pruning by maintaining maximum probabilities at sub-table level instead of the table level, thereby boosting the threshold even further using Rule 4. Moreover, thanks to the introduction of sub-tables, which are groups of tuples whose probabilities fall into the same range, it is now possible to eliminate entire sub-tables instead of individual tuples before performing the join. This also optimizes the query evaluation by minimizing the number of tuples to be retrieved for the join.

Example 5.3.1 *As shown in Fig. 5.2, we have already partitioned both tables* R_1 *and* R_2 *into two sub-tables based on their tuple probabilities. The first sub-table has all tuples with probabilities less than or equal to 0.5 while the second has the rest. For example,* R_1 *is partitioned into* R_{11} *and* R_{12}*. To perform an equality join between* R_1 *and* R_2 *on* $A = D$ *with threshold* $\theta = 0.3$*, we first find out all tuples from* R_2 *that should be joined with* R_{11}*. Since the maximum tuple probability of* R_{11} *is 0.5 and the threshold is 0.3, we use* $0.3/0.5 = 0.6$ *as the new threshold for* R_2 *and prune the whole sub-table* R_{21} *away. For the remaining* R_{22}*, we use the 0.6 threshold to prune its tuple* t_{21} *away, leaving a single tuple* t_{24} *to be joined with* R_{11}*. Similarly, for* R_{12} *with maximum tuple probability 0.8, we obtain the new threshold* $0.3/0.8 = 0.375$

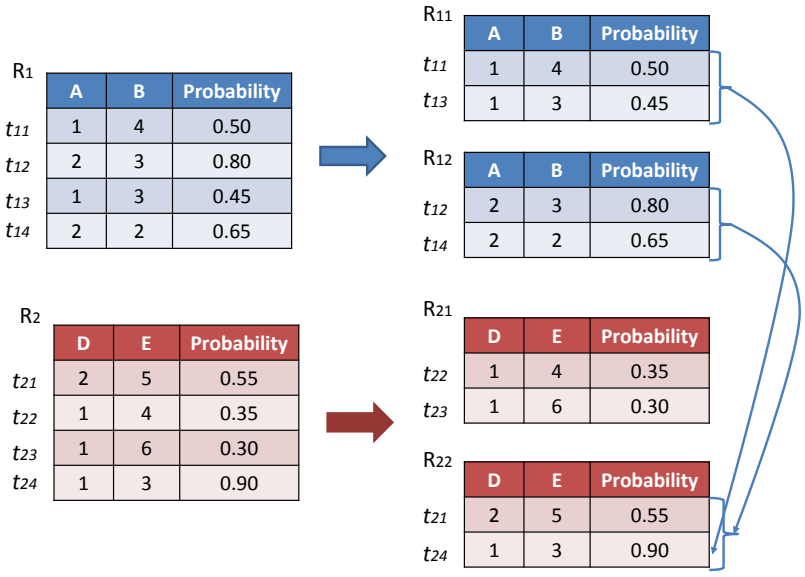

Fig. 5.2.: Range partitioning technique for joining R_1 and R_2 on $A = D$ with $\theta = 0.3$

for pruning. Since R_{21}'s maximum probability is 0.35 < 0.375, we can eliminate the whole sub-table and only perform joins between tuples from R_{12} and R_{22}.

If we had not used the above range partitioning technique, we would have checked all 4*4=16 pairs of tuples for the join, since applying Rule 3 (prune tuples from both tables whose probabilities are below the threshold) would not have achieved any saving (all tuples from R_1 and R_2 have probabilities 0.3). However, with range partitioning, we only need to check $2*1+2*2=6$ pairs of tuples for the join, saving as much as $10/16 = 62.5\%$ of work.

5.3.2 Sampling for Dedup

As mentioned in Rule 6 and 7, to dedup on table R, we can pre-construct histograms on the whole table R for pruning purposes. Later in Section 5.4.4, we will see that histograms on base tables are not useful for all queries, and new ones on

selected attributes must be constructed on the fly for pruning, which are kept for future references.

Maintaining such histograms could be expensive, as potentially there could exist as many as exponential number of histograms for each base table with regard to the number of its attributes. Moreover, in case of complicated SPJ queries that involve multiple tables and have dedup operators, new histograms are likely to be requested at the time of the query, whose construction would significantly slow down the query processing.

In these situations, it is easier to use sampling to estimate the threshold for pruning rather than analyze the complex query and build new histograms from scratch that correspond to the particular query at hand. However, the sampling approach to threshold estimation has no guarantee on the precision or the recall of the final results, as statistics gathered for a sample do not always reflect the statistics about the original data. In practice, it serves as a simple and quick method to estimate the threshold for pruning when there is duplicate elimination. We show in Section 5.5 that it has decent performance in terms of precision and recall in our experiments.

The key to estimating the threshold to be pushed down a dedup operator in Rule 6 is to estimate the occurrence count k. We can use the following algorithm for sampling to obtain this k: Given a sampling percentage x, we first sample all base tables R_is in the query such that the size of the sample table corresponding to R_i is $x * |R_i|$. The samples are generated based on the existing histograms of base tables. Tuple values with larger occurrence counts also have higher probabilities to be fetched to the sample. Note that the histograms here are on whole tuples of the base tables (i.e., all attributes are included), hence we only need k histograms for k base tables. If such histograms are unavailable at the time of the query, we simply obtain our samples by randomly choosing tuples from the base tables.

Once the samples are obtained, we run the entire query (except the final thresholding step) on the samples to obtain the final results (referred as the "sampling results" for short), from which we compute the following statistics of the occurrence

counts: the average, maximum and minimum occurrence counts of all values in the sampling results, denoted as avg, max and min respectively. Similarly, we denote the average, maximum and minimum counts of all values in the actual results obtained from the original tables as AVG, MAX and MIN, clearly we have max $\leq$ MAX and min MIN. However, for average counts, either avg AVG or avg $<$ AVG could hold depending on the tuples in the sample.

Note that we obtain the aggregates of occurrence counts instead of the exact histogram of the sampling results. This is because the samples are usually very small compared with original tables, hence many tuple values present in the original tables are not found in the samples. For such values, we do not know their occurrence count k in order to obtain θ/k as the new threshold to be pushed down a dedup operator for pruning tuples with these values. However, by using aggregates of occurrence counts from the sampling results, we can compute the new threshold as the original threshold divided by the chosen aggregate (avg, max or min), and use this same threshold for all tuple values.

In Section 5.5.2, we present our experimental results on sampling and discuss in detail the effect of sampling percentage and the choice of occurrence count (i.e., avg, max or min).

5.4 General Pruning Schemes

In this section, we present our general pruning schemes for any given SQL query Q that involves selection, projection, join and duplicate elimination. Our schemes leverage the pruning rules proposed in Section 5.2 as well as pruning techniques in Section 5.3, and aim to estimate an accurate threshold at each step during the query evaluation for better pruning.

5.4.1 Naïve Pruning Scheme

Our naïve pruning scheme always uses the same threshold for pruning regardless of the query operator currently being evaluated. This scheme, in effect, applies Rule 1, 2 and 3 for selection, projection and join respectively, all of which trickle down the same threshold through the query operator. In case of dedup, however, instead of applying Rule 6 or 7, the naïve pruning scheme sticks to the same threshold for pruning. In this way, the scheme avoids the cost of building and querying the histograms for obtaining the occurrence count k or the probability bounds (see Section 5.2.2). However, it also introduces pruning errors by possibly decreasing the recall: Tuples pruned individually because their probabilities are below the threshold θ may have a probability over θ after dedup, i.e., the naïve pruning scheme may miss correct results. The precision, on the other hand, remains 1 for all single queries that do not have joins. For such queries with no dedup, this trivially holds as selection and projection do not change tuple probabilities. For those with dedup, precision is 1 because if there exists a duplicate tuple t_i with $\Pr(t_i) \quad \theta$, for tuple t after dedup, $\Pr(t) = \Pr(\bigvee t_i) \quad \max \Pr(t_i) \quad \theta$, i.e., t must be a dedup result. The precision of the naïve scheme may drop below 1 when the query to be evaluated has joins. Applying Rule 3 with the same threshold on both tables before the join does not prune out potential results, but may keep many unqualified tuples whose probabilities after the join fall below the threshold.

5.4.2 Range-based Pruning Scheme

Our range-based pruning scheme examines the query plan in a bottom-up fashion, and at each node in the plan computes the probability range of the node's lineage. Probability range of a node at level j in the query plan tree is computed from probability ranges of its children at level $j + 1$. This process is similar to the one in [55] except that in our case, the lineage may not be in DNF and the random variables in the lineage (the base tuples) may not be independent from each other. Pruning occurs

at the end of the query by comparing the threshold θ with the maximum bound of the lineage associated with the result tuples: If θ is larger than the maximum bound, we prune the tuple away.

The algorithm for computing the bounds is recursive, as follows: For a boolean expression e in the lineage that consists solely of a random variable r, we set both the maximum and minimum bounds of e: $\Pr_{\max}(e)$ and $\Pr_{\max}(e)$ to be $\Pr(r)$ and return the bounds; for a boolean expression e that is a disjunction of k other boolean expressions (denoted as $e = \bigvee_{i=1}^{k} e_i$), we compute $\Pr_{\max}(e)$ as $\sum_{i=1}^{k} \Pr_{\max}(e_i)$ and $\Pr_{\min}(e)$ as $\max_{i=1}^{k} \Pr_{\min}(e_i)$; for a boolean expression e that is a conjunction of k other boolean expressions, i.e., $e = \bigwedge_{i=1}^{k} e_i$, we compute $\Pr_{\max}(e)$ as $\min_{i=1}^{k} \Pr_{\max}(e_i)$ and $\Pr_{\min}(e)$ as 0.

The range-based scheme uses Rule 1, 2, 3 and 7 for pruning.

5.4.3 Sampling-based Pruning Scheme

As discussed in Section 5.3.2, the sampling approach does not depend on the query at hand: All it does is to obtain a small sample for each base table so that when the query comes, it can run the whole query on the corresponding samples and obtain the aggregate occurrence counts to be used for pruning on the original tables.

In general, given a query with dedup, a sampling percentage, and a choice of average, minimum or maximum count to be used for pruning, the sampling-based scheme first obtains samples of size: percentage * table size, then run the whole query and compute the choice of count k from the sampling results immediately after the last dedup operator is applied. Then it pushes down θ/k all the way to the leaf level of the query plan and prunes away tuples from all base tables in the query with tuple probabilities $< \theta/k$. Rule 1, 2, 3 and 6 are applied during this process. Here we are only concerned with the last dedup instead of intermediate dedups because a query plan where dedup is applied multiple times is equivalent to one where dedup is applied at the end.

5.4.4 Histogram-based Pruning Scheme

Rule 6 proposed in Section 5.2 computes the new threshold to be pushed through a dedup operator as the old threshold (θ) divided by the occurrence count (k) of the current tuple value (v). On the other hand, Rule 7 leverages the probability bounds of disjunctions generated by duplicate elimination for pruning. As we have mentioned earlier, histograms can be leveraged in applying Rule 6 and Rule 7. For Rule 6, we can use a simple histogram where tuple values are mapped to their occurrence counts; for Rule 7, we store additional statistics that correspond to the tuple values: the maximum probability and the sum of all probabilities of tuples with the same value.

For a given SPJ query Q with duplicate elimination, our goal is to estimate the threshold to be pushed through each query operator without actually computing the query. We have mentioned earlier that knowledge about the data and the query is helpful in solving the problem. The histogram-based pruning scheme is an ideal example to illustrate how we can leverage both data and query for pruning.

First of all, we analyze the query and find out all attributes that will appear in the query result. We then partition them into k sets such that attributes in the i-th set are from base table i ($1 \leq i \leq k$). For each attribute set i consisting of j attributes $A_{i1}, A_{i2}, ..., Aij$ from table i, we check if there already exists a histogram that is built on these attributes of table i. For example, if the original table R has three attributes A, B, C in which only A is projected in the end, then we need a histogram on R with a single attribute A. If such histograms are not available, we build them on the fly and keep them for later use.

Next we use the information from the histograms for pruning: If the histogram only stores occurrence counts k for values, we estimate the occurrence counts $\hat{k}$ for actual values at the time of dedup as k and use Rule 6 to prune tuples before dedup with a new threshold θ/k. Since the histogram is built upon base table R and not all tuples from base tables become final results that satisfy the whole query, the actual occurrence count $\hat{k} \leq k$. Therefore, using θ/k for pruning will not prune away

potential results, i.e., the histogram-based pruning has a recall of 1. If the histogram also stores the sum of probabilities corresponding to value v, we can apply Rule 7 by checking directly if the sum is $< \theta$: If so we prune all tuples with the same value v; otherwise we keep the tuples. Rule 7 also has a recall of 1 because only tuples that for sure cannot meet the threshold are pruned away.

In both cases, pruning occurs at the leaf level of the query plan, i.e., all pruning is done on base tables. Similar to the sampling approach, histogram-based pruning is concerned only with the last dedup and the final attributes in the query – we do not care whether there are intermediate dedups or what intermediate queries are performed.

After the "top-down" process of pruning base tuples, we go "bottom-up" to evaluate the query and to compute the probability range at each stage of the query for potential pruning, which is essentially the same as the range-pruning scheme introduced in Section 5.4.2. The tuples that pass both top-down and bottom-up pruning are finally returned as results.

None of the above pruning schemes introduced in this section assume the knowledge of tuple dependency or independency. However, if such information is known before the query is evaluated, we can leverage the information for better pruning, such as Rule 4 and 5 for joins as well as the range partitioning technique introduced in Section 5.3.1.

5.5 Experimental Results

We empirically evaluate the performance of our pruning schemes on various synthetic data sets by running simulation of database query processing. All programs were written in C# and were run on a MAC with T2500 2GHz CPU and 2GB main memory.

5.5.1 Data Sets and Experiment Setup

We generated the synthetic data set as follows: For each base table, we first generated the schema: the number of attributes and the minimum and maximum value allowed for each attribute. The actual value of an attribute in a tuple is generated uniformly within this range. Given the total number of tuples in each base table, the number of duplicate tuples in the table, and the maximum number of occurrence count for all tuple values, we first obtained a duplicate tuple value by generating random values for each attribute in the tuple independently. We then generated the actual number of occurrence count k for that value and repeated it k times in the table. The default number of tuples in base tables is 100,000. The default maximum number of occurrence count is 5 and the default number of duplicates is 10% of the table size. By default, base tables are independent from each other. The tuples within the same table are also independent by default. However, we can generate dependencies when needed. For example, to generate mutually exclusive tuples in a table, we can randomly generate sets of tuple IDs in the table, each of which denotes a set of tuples that are mutually exclusive.

All tuples in base tables are assigned distinct random variables, which are used in the lineage of tuples.

In our evaluations, we consider two types of queries: i) queries on single tables, referred to as *single queries* for short, and ii) queries on multiple tables, referred to as *join queries*. Let T_1 and T_2 be two tables with tuple uncertainty. T_1 has three attributes: A_1, A_2, A_3 and T_2 has two attributes: B_1 and B_2. The possible range of each attribute in T_1 is as follows: $A_1 \in [0, 10,000), A_2 \in [0, 5000), A_3 in [0, 3000)$. The value ranges of B_1 and B_2 are the same as those of A_1 and A_2. Similar to T_1, T has three attributes A_1, A_2, A_3 as well. However, the value ranges of these attributes in T are different: $A_1 \in [0, 2000), A_2 \in [0, 3000), A_3 in [0, 5000)$. We test the following two queries in specific (note that we use SELECT DISTINCT for the dedup operator):

Q1:

```
SELECT DISTINCT A1, A2 FROM T WHERE A1 < c
```

Q2:

```
SELECT DISTINCT A1 FROM (
  (SELECT DISTINCT A1, A2 FROM T1 WHERE A1 < c1 AS TT1)
   INNER JOIN
  (SELECT DISTINCT B1 FROM T2 WHERE B1 < c2 AS TT2)
   ON TT1.A1 = TT2.B1)
```

At the end of each query, we apply the threshold operator τ_θ to prune out all result tuples with probabilities less than θ.

5.5.2 Performance of Pruning Schemes

We compare the performance of different pruning schemes in terms of precision, recall and time cost. `Naïve` refers to the naïve pruning scheme, `Range` refers to the range-based pruning scheme, `Sampling` refers to the sampling-based scheme, and finally, `Histogram1` and `Histogram2` refer to the histogram-based schemes with the former applying Rule 6 and the latter applying Rule 7. Precision and recall measure the effectiveness of our schemes while the time cost reflects the efficiency of the algorithms. We compute the precision and recall against the probabilistic version of Karp-Luby Monte Carlo algorithm [55, 56] for threshold query evaluation: Given a query Q, the results are computed first without thresholding. For each result tuple, we use Karp-Luby algorithm to evaluate the probability of its lineage, which takes into account the dependencies between base tuples and returns tuples whose approximate probabilities are over the given threshold.

For each lineage evaluation, we run the simulation 100,000 times to make the approximated probability as close to the real probability as possible. We then compare the results of our pruning algorithms with the results returned by the simulation, and compute the precision and recall of our pruning algorithms.

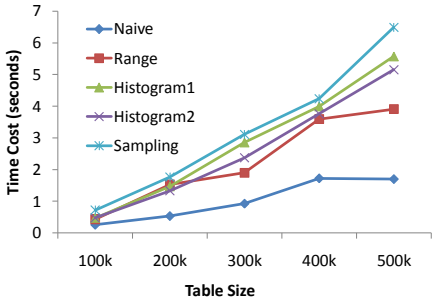

Fig. 5.3.: Effect of table size

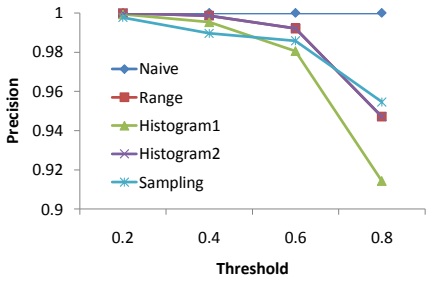

Fig. 5.4.: Effect of threshold

Effect of Data Set Size

We increase the data set size (i.e., table size) to test the scalability of our pruning schemes, as shown in Fig. 5.3. The time cost of all pruning algorithms for Q1 increases as the table size increases from 100k to 500k. `Naïve` is the fastest pruning scheme while `Sampling` is the slowest. This is because `Sampling` needs to generate the samples first, run the whole query on the samples, before getting the estimate of the new threshold for pruning on the original table. Note that in Fig. 5.3, neither `Histogram1` nor `Histogram2` includes the time to build the histogram – since the histogram may be re-used later for future queries, we do not count the time to build the histogram as a cost in the two histogram-based pruning schemes. Later in Fig. 5.14, we show the time cost for constructing histograms separate from pruning with these histograms.

Effect of Threshold

As the threshold increases, fewer tuples satisfy the query with probabilities that meet the threshold. Fig. 5.4 shows the effect of the threshold on the precision of pruning results. For all pruning algorithms except `Naïve`, with an increasing threshold, the precision decreases. The precision of `Naïve`, on the other hand, remains to be 1 regardless of thresholds. This is because Fig. 5.4 was based on Q1 on a single

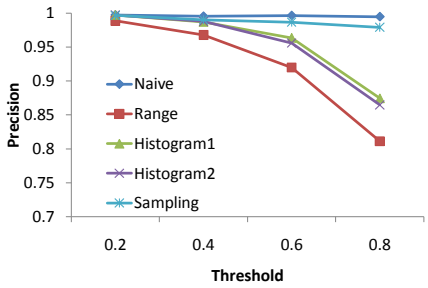

Fig. 5.5.: Effect of threshold on precision for Q2

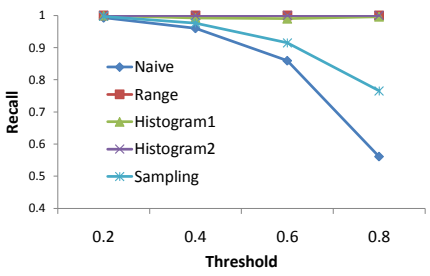

Fig. 5.6.: Effect of threshold on reall for Q2

table. With no joins, the only operator that affects the tuple probability is the dedup operator. Since `Naïve` keeps all tuples whose probabilities meet the threshold, if the tuple t_i is not a duplicate, it must be a result; otherwise, the disjunction of j duplicate tuples $t_1, ..., t_j$ in which t_i θ must also have a probability θ (refer to Rule 7 for the probability range of disjunctions), hence t_i (or more accurately, the value of it) must also be a result. Therefore, the precision of `Naïve` is 1.

For queries with joins, however, `Naïve` no longer keeps precision at 1 – in cases where probabilities of join tuples returned by `Naïve` fail to pass the threshold (even though before the join, both tuples to be joined have probabilities θ), `Naïve` also returns incorrect results, hence the precision is less than 1. Fig. 5.5 and Fig. 5.6 show the effects of the threshold on precision and recall respectively. We can see in Fig. 5.5, the precision of `Naïve` is no longer always 1 – it slightly decreases as the threshold increases, although remains to be the highest precision of all the pruning schemes. However, this is compensated by the fact that `Naïve` has the lowest recall of all, as seen in Fig. 5.6. In general, as the threshold increases, both precision and recall decrease. The recall for `Range` and `Histogram2`, however, is always 1 regardless of thresholds. Both pruning schemes check the probability ranges of tuple lineage and apply Rule 7 for dedup and Rule 3, Rule 4 for joins (the tables in the experiments are independent from each other, so Rule 4 can also be applied). These rules guarantee that no potential result would be dropped, i.e., the recall of tuples after the pruning

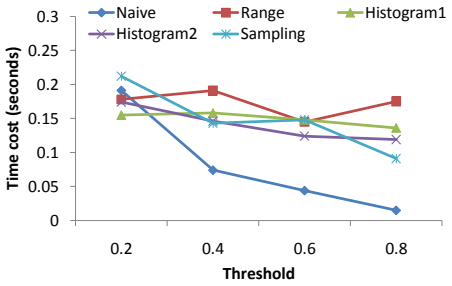

Fig. 5.7.: Effect of threshold on time cost for Q2

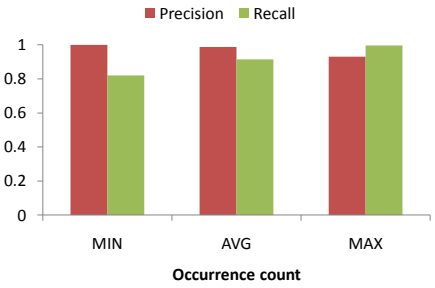

Fig. 5.8.: Effect of occurrence count in sampling

is 1. `Histogram2`, though its recall is close to 1, does not keep its recall at 1 at all times. Recall that `Histogram2` applies Rule 6 to prune out tuples with value v whose probabilities are below θ/k where k is the occurrence count of the current tuple value v. Suppose v remains because there is one tuple t whose probability is θ/k. Suppose also v is to be joined with tuples from another table. If `Histogram2` had not pruned all the other tuples t' with $t' = v$, the probability of result v, which is a disjunction of all tuples with value v (denote as $\Pr(v)$), would have been big enough to yield a join tuple probability θ. However, now that only a single tuple t remains to be joined with the other table and $\Pr(t) \leq \Pr(v)$, the join tuple probability fails to meet the threshold θ. Therefore, the join tuple from v is pruned even though it is actually a result. Hence the recall of `Histogram2` could be less than 1.

Finally, Fig. 5.7 gives the time cost of various pruning algorithms with regard to the threshold. All pruning schemes except `Range` and `Sampling` takes less time to finish as the threshold increases, for pruning capabilities increase as the threshold increases. The exception of `Range` is due to the fact that `Range` does not leverage the threshold for early pruning – it always waits till the end when it has all tuple lineage to estimate the probability range for pruning. Therefore, a larger threshold does not mean as much to `Range` in terms of performance gain as to other pruning schemes. `Sampling` also does not conform to the decreasing trend of the time cost in Fig. 5.7.

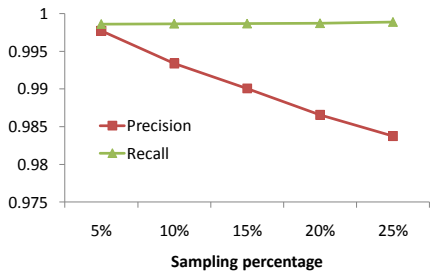

Fig. 5.9.: Effect of sampling percentage for Q1

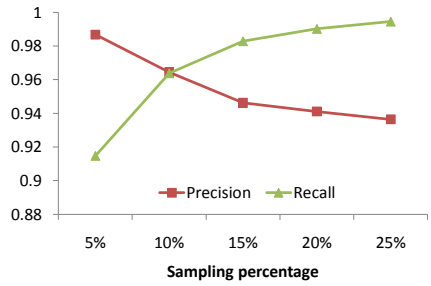

Fig. 5.10.: Effect of sampling percentage for Q2

This may be due to a significant part of the total time spent on the additional step of sampling and running queries on the sample before the actual pruning.

Effect of Occurrence Count and Sample Size

When the choice of occurrence count changes from min to average to max, the precision and recall of `Sampling` also changes, as seen in Fig. 5.8. The sole effect of the choice of occurrence count is on the new threshold to be used later for pruning on the original table. Using Rule 6, the larger the count, the smaller the threshold. Therefore, it is less likely to prune away potential results (hence the increasing recall) but more likely to return tuples whose final probabilities fail the threshold (hence the decreasing precision). Since min $\leq$ avg $\leq$ max, we observe the above trend of an increasing recall and a decreasing precision.

For the sampling-based scheme, the size of the sample affects the performance of the pruning. The larger the sample is, the more accurately the sample captures the characteristics of the original data set. Fig. 5.9 and Fig. 5.10 show the precision and recall of `Sampling` for Q1 and Q2 respectively. In both figures, precision drops and and recall increases as the sampling percentage increases. While for Q1 in Fig. 5.9, recall does not increase much, recall for Q2 in Fig. 5.10 significantly increases as the sampling percentage increases. As the sample size grows, the average of occurrence

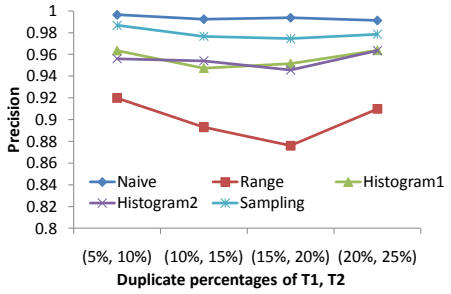

Fig. 5.11.: Effect of duplicate on precision

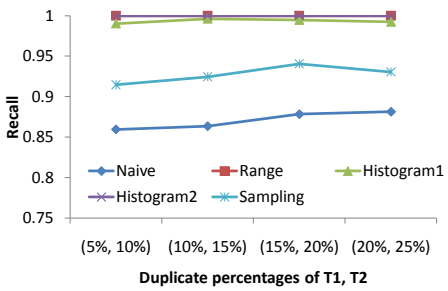

Fig. 5.12.: Effect of duplicate on recall

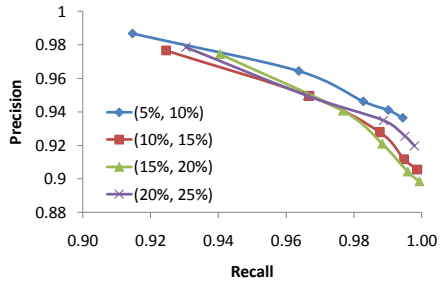

Fig. 5.13.: Effect of duplicate percentage on sampling

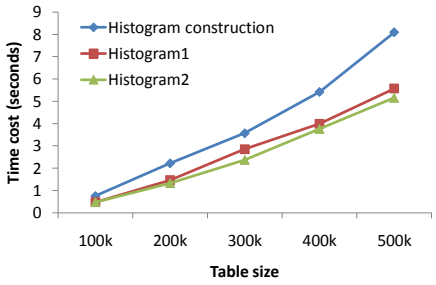

Fig. 5.14.: Time cost for histogram-based pruning

counts (the default choice) also increases, meaning that the new threshold to be used (θ/avg) becomes smaller. Similar to our discussion for Fig. 5.8, we can deduce that the precision should decline while the recall should improve.

Effect of Duplicates

We vary the "degree" of duplicates by changing the total number of duplicate tuples in a table. Fig. 5.11 and Fig. 5.12 show the change of precision and recall with regard to the duplicate percentages of the tables to be joined. Initially, T_1 has 5% duplicates while T_2 has 10%. We gradually increase this to 20% for T_1 and 25% for T_2. It is not easy to see a trend of decreasing precision and increasing recall in the

two figures. With more duplicates in the table, `Sampling`, for example, is likely to get larger occurrence counts and apply smaller thresholds, hence larger recall and smaller precision. This is in accordance with the general trend in Fig. 5.11 and Fig. 5.12 for `Sampling`. Note that we have discussed earlier that for `Histogram2` and `Range`, recall is always 1, which is again proved in Fig. 5.12.

Fig. 5.13 shows precision and recall in the same figure. Each series in the figure is a series of (recall, precision) points for fixed join tables with certain duplicate percentages. Each mark on the series represents a different sampling percentage, from 5% to 25%. We can easily observe that as the sampling percentage increases, precision decreases and recall increases. In order to find a sampling percentage that is "optimal" – with both high precision and high recall, we can find a point on a series that maximizes some criteria such as the sum or the product of the precision and the recall.

Histogram Construction vs. Pruning

Finally, we show the histogram construction time v.s. histogram pruning time in Fig. 5.14. Note that in all figures in this section, the time cost of `Histogram1` and `Histogram2` does not include the time cost for building the histograms that the pruning algorithms use.

We can clearly see in Fig. 5.14 that building histograms on the fly is very expensive, with its time cost much more than the time to prune tuples with existing histograms. Therefore, if the histogram in need is not available at the time of the query, the time to build a new one from scratch will dominate the total time for the histogram-based pruning scheme, i.e., it will significantly slow down the pruning algorithms. In this case, as we pointed out in Section 5.3.2, it may be faster and easier to use the sampling method.

6. CONCLUSIONS

Applications with uncertain data face many challenges that mainly come from two aspects of uncertain data: i) The uncertainty in data calls for effective and efficient data representation, as well as novel techniques for efficiently pruning large search space; ii) The dependencies in data that come from the original data or generated during the query evaluation must be handled properly by mechanisms to capture dependencies as well as by query processing algorithms that take dependencies into account when computing query results.

This dissertation focused on the first aspect of uncertain data in query evaluation. We presented novel algorithms for efficiently processing the following queries on uncertain data: probabilistic nearest neighbor threshold queries (PNNT), probabilistic skyline queries and threshold SPJ queries. We gave an efficient algorithm to process PNNT queries for uncertain data with missing probabilities, a problem that has not been addressed by any previous paper. We designed an augmented R-tree index for efficient pruning with a probability threshold.

For probabilistic skyline queries, we studied two versions of the instance-level probabilistic skylines: one with a threshold, the other without. To the best of our knowledge, we were the first to study the problem of computing all skyline probabilities (probabilistic skylines without thresholds). We designed an efficient algorithm based on space partitioning and weighted dominance counting. We gave strict complexity analysis of our sub-quadratic algorithm for $d = 2$ and showed how to extend it to higher dimensions. Our algorithm provides the user with the greatest flexibility in identifying their own interesting skyline instances by returning skyline probabilities of all instances and making no assumptions on how the user will use the skyline results (i.e. the user utility is not restricted in any way). Such skyline analysis only needs to be done once for all users, and the results are useful for all users regardless of

their different utilities. For cases when the user is interested in instances with skyline probabilities over a certain threshold, we proposed two filtering schemes to avoid the expensive skyline probability computations. In our preliminary filtering scheme, we designed two indexing structures based on the range search tree to facilitate bounding of a query instance's skyline probability. Our more refined filtering scheme further explores the dominance relationship for massive filtering.

While our algorithms for both PNNT and probabilistic skylines are specific to the query in consideration, the optimization rules proposed in Chapter 4 are for general probabilistic threshold queries that involve selections, projections and joins (threshold SPJ queries). We identified query equivalences for SPJ queries and established the correctness of pushing down the threshold operator in the query plan. Our SPJ query optimization works for the complicated uncertain database model proposed in [1] with both attribute and tuple uncertainty as well as dependencies between arbitrary attribute sets. The optimization rules are shown to be effective in reducing query processing time through experiments on both real and synthetic data sets. We further studied the optimization of SPJ queries when duplicate elimination is allowed. We adopted a general tuple uncertainty model for this case and proposed new optimization rules and pruning techniques that aim at effective pruning for queries with duplicate elimination while maintaining high precision and recall.

LIST OF REFERENCES

LIST OF REFERENCES

[1] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, and R. Cheng, "Database support for probabilistic attributes and tuples," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2008.

[2] R. Cheng, D. V. Kalashnikov, and S. Prabhakar, "Querying imprecise data in moving object environments," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2004.

[3] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom, "ULDBs: databases with uncertainty and lineage," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[4] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu, "MYSTIQ: a system for finding more answers by using probabilities," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2005.

[5] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein, "BayesStore: Managing large, uncertain data repositories with probabilistic graphical models," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[6] J. Huang, L. Antova, C. Koch, and D. Olteanu, "MayBMS: A probabilistic database management system (demonstration)," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2009.

[7] P. Sen and A. Deshpande, "Representing and querying correlated tuples in probabilistic databases," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[8] D. Olteanu, J. Huang, and C. Koch, "Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2009.

[9] O. Kennedy and C. Koch, "PIP: A database system for great and small expectations," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2010.

[10] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

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

[12] T. Imielinski and W. Lipski, "Incomplete information in relational databases," *Journal of the ACM*, vol. 31, no. 4, 1984.

[13] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "URank: Formulation and efficient evaluation of top-k queries in uncertain databases," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2007.

[14] M. Hua, J. Pei, W. Zhang, and X. Lin, "Ranking queries on uncertain data: A probabilistic threshold approach," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2008.

[15] T. Ge, S. Zdonik, and S. Madden, "Top-k queries on uncertain data: On score distribution and typical answers," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2009.

[16] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[17] M. Atallah and Y. Qi, "Computing all skyline probabilities for uncertain data," in *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2009.

[18] M. Atallah, Y. Qi, and H. Yuan, "Asymptotically efficient algorithms for skyline probabilities of uncertain data," *ACM Transactions on Database System (TODS)*, vol. 36, no. 3, 2011.

[19] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu, "Probabilistic skyline operator over sliding windows," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2009.

[20] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter, "Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2008.

[21] G. Beskales, M. A. Soliman, and I. F. Ilyas, "Efficient search for the top-k probable nearest neighbors in uncertain databases," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[22] Y. Qi, S. Singh, R. Shah, and S. Prabhakar, "Indexing probabilistic nearest-neighbor threshold queries," in *Proceedings of the Workshop on Management of Uncertain Data (MUD)*, 2008.

[23] M. A. Cheema, X. Lin, W. Wang, W. Zhang, and J. Pei, "Probabilistic reverse nearest neighbor queries on uncertain data," *ACM Transactions on Database System (TODS)*, vol. 22, no. 4, 2010.

[24] X. Lian and L. Chen, "Efficient processing of probabilistic reverse nearest neighbor queries over uncertain data," *VLDB Journal (VLDBJ)*, vol. 18, no. 3, 2009.

[25] Y. Tao, X. Xiao, and R. Cheng, "Range search on multidimensional uncertain data," *ACM Transactions on Database System (TODS)*, vol. 32, no. 3, 2007.

[26] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar, "Indexing multi-dimensional uncertain data with arbitrary probability density functions," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2005.

[27] Y. Qi, R. Jain, S. Singh, and S. Prabhakar, "Threshold query optimization for uncertain data," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2010.

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

[29] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. Vitter, "Efficient indexing methods for probabilistic threshold queries over uncertain data," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[30] P. K. Agarwal, S.-W. Cheng, Y. Tao, and K. Yi, "Indexing uncertain data," in *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2009.

[31] B. Kanagal and A. Deshpande, "Indexing correlated probabilistic databases," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2009.

[32] X. Lian and L. Chen, "Probabilistic ranked queries in uncertain databases," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2008.

[33] F. Li, K. Yi, and J. Jestes, "Ranking distributed probabilistic data," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2009.

[34] G. Cormode, F. Li, and K. Yi, "Semantics of ranking queries for probabilistic data and expected ranks," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2009.

[35] F. Li, K. Yi, and J. Jestes, "Ranking queries on distributed probabilistic data," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2009.

[36] G. Tracevski, O. Wolfson, K. Hinrichs, and S. Chamberlain, "Managing uncertainty in moving objects databases," *ACM Transactions on Database System (TODS)*, vol. 29, no. 3, pp. 463–507, 2004.

[37] R. Cheng, D. V. Kalashnikov, and S. Prabhakar, "Evaluating probabilistic queries over imprecise data," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2003.

[38] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[39] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia, "Efficient join processing over uncertain data," in *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, 2006.

[40] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 1995.

[41] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl, "Fast nearest neighbor search in high-dimensional space," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 1998.

[42] J. M. Kleinberg, "Two algorithms for nearest-neighbor search in high dimensions," in *ACM Symposium on Theory of Computing*, 1997.

[43] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.

[44] P. Wu, D. Agrawal, O. Egecioglu, and A. El Abbadi, "DeltaSky: Optimal maintenance of skyline deletions without exclusive dominance region generation," *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[45] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis, "SKYPEER: Efficient subspace skyline computation over distributed data," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[46] M. Morse, J. M. Patel, and H. V. Jagadish, "Efficient skyline computation over low-cardinality domains," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[47] J. Pei, A. W.-C. Fu, X. Lin, and H. Wang, "Computing compressed multidimensional skyline cubes efficiently," *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[48] X. Lian and L. Chen, "Monochromatic and bichromatic reverse skyline search over uncertain databases," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2008.

[49] Y. Qi and M. Atallah, "Identifying interesting instances for probabilistic skylines," in *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, 2010.

[50] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[51] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*. Springer-Verlag New York, Inc., 1984.

[52] D. E. Willard, "New data structures for orthogonal range queries," *SIAM Journal of Computing*, vol. 14, pp. 232–253, February 1985.

[53] E. M. McCreight, "Priority search trees," *SIAM Journal of Computing*, vol. 14, 1985.

[54] L. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal of Computing*, vol. 8, no. 3, 1979.

[55] D. Olteanu, J. Huang, and C. Koch, "Approximate confidence computation in probabilistic databases," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2010.

[56] R. M. Karp, M. Luby, and N. Madras, "Monte-carlo approximation algorithms for enumeration problems," *Journal of Algorithms*, vol. 10, no. 3, 1989.

VITA

VITA

Yinian Qi was born in the city of Changshu in Jiangsu Province, China. Her hometown has over 5,000 years of history and is famous for its beautiful scenery. She stayed there until after high school, when she left to study computer science at Fudan University in Shanghai. After graduating cum laude from Fudan with a bachelor's degree, she joined Alcatel Shanghai Bell, an Alcatel-Lucent company, as a software engineer to work on GSM networks. She decided to apply for graduate schools in the US a year later, and was happy to get an offer from Purdue University in 2006.

While at Purdue, Yinian worked with both Prof. Sunil Prabhakar and Prof. Mike Atallah in uncertain data research, with a focus on efficient query processing. Her research lead to publications at several top-tier conferences including both SIGMOD and PODS. In May 2008, she received her master's degree in computer science, and went to Silicon Valley for her first internship at a web startup. She subsequently interned at Microsoft for the following two summers.

Apart from academic study and research, Yinian actively participated in activities and events both on and off campus. She served on the Graduate Student Board as well as on the leadership team of Women in Science Program in College of Science. She was a frequent participant of the Grace Hopper Women in Computing Conference, where she was selected several times to compete in the ACM Student Research Competition. In 2010, Yinian was named as a US finalist of the renowned Google Anita Borg Memorial Scholarship, and was recommended by the Computer Science Department as one of the two Ph.D. candidates to receive a Bilsland Dissertation Fellowship.

Yinian defended her dissertation in July 2011, and received her Ph.D. degree in August 2011. After graduation, she joined Oracle in Redwood Shores, California as a member of technical staff to continue advancing database research and development.