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 Sarvjeet Singh

Entitled Database Support for Uncertain Data

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Sunil Prabhakar Chair	Dr. Rahul Shah
Dr. Susanne Hambrusch	
Dr. Jennifer Neville	
Dr. Ahmen Elmagarmid	

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): Dr. Sunil Prabhakar

Approved by: Dr. Aditya Mathur — 01-26-2009
Head of the Graduate Program — Date

Graduate School Form 20
(Revised 10/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Database Support 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.

Sarvjeet Singh
Signature of Candidate

01-26-2009
Date

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

DATABASE SUPPORT FOR UNCERTAIN DATA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sarvjeet Singh

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2009

Purdue University

West Lafayette, Indiana

Dedicated to my parents who offered me unconditional love and support throughout the course of my Ph.D.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Sunil Prabhakar, without whom this thesis would have been an exercise in futility. His kind words and constant support always inspired me to achieve the very best of me.

I would also like to acknowledge and thank my wife, Niharika, who was always by my side during the long nights before deadlines. She was always there to help me with writing and technical figures. Her support was indispensable for my research and this dissertation.

A special thanks goes to my parents and family who showed their constant support throughout my Ph.D. and encouraged me in times when my resolve began to waiver.

I would also like to thank Dr. Rahul Shah and Dr. Susanne Hambrusch. The hours of discussion with them helped me clear and absorb the technicalities of my field. Finally, I would like to thank my colleagues Chris Mayfield, Sagar Mittal and Reynold Cheng for their significant contribution through discussions, comments and technical papers.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Singh, Sarvjeet Ph.D., Purdue University, May 2009. Database Support for Uncertain Data. Major Professor: Sunil Prabhakar.

In recent years, the field of uncertainty management in databases has received considerable interest due to the presence of numerous applications that handle probabilistic data. In this dissertation, we identify and solve important issues for managing uncertain data natively at the database level. We propose the semantics of join operation in the presence of attribute uncertainty and present various pruning techniques to significantly improve the join performance. Two index structures for indexing categorical uncertain data are also presented. For optimization of probabilistic queries, we discuss novel selectivity estimation techniques. We also introduce a new model for handling arbitrary pdf (both discrete and continuous) attributes natively at the database level. This model is consistent with Possible Worlds Semantics and is closed under the fundamental relation operations of selection, projection and join. We also present and discuss the implementation of Orion – a relational database with native support for uncertain data. Orion is developed as an extension of the open source relational database, PostgreSQL. The experiments performed in Orion show the effectiveness and efficiency of our approach.

1 INTRODUCTION

Traditionally, databases have required data to be modeled in terms of precise values. However there are many applications where uncertainty, or imprecision in values is inherent or desirable [1–3].

Consider, for example, scientific applications that record measurements taken from sensors or other devices. These measurements are usually inexact, with known degrees of errors. Sometimes, errors are also introduced in order to achieve scalability. Consider the case of sensor databases. It is infeasible (due to resource constraints such as batteries and bandwidth) to continuously monitor every single change in value for every sensor. One solution to this problem, while limiting the degree of error, is to allow each sensor to not send updates unless the value has changed significantly, or a specified amount of time has elapsed. In this model, called the dead-reckoning approach [4], the value of the sensor is correctly modeled as a range around the last reported value. As another example, in a Location-based service application, users may wish to provide approximate, imprecise locations in order to preserve their privacy.

Data cleansing applications often result in uncertainty in the "cleaned" value of an attribute. Many cleansing tools provide alternative corrections with associated likelihood. For example, as part of an ongoing project at Purdue University, the movement of nurses is being tracked in order to study their behavior and effectiveness of current practices [5]. Nurses carry RFID tags as they move about a hospital. Numerous readers located around the building report the presence of tags in their vicinity. The collected data is stored centrally in the form "Nurse 10 in Room 5 at 10:05 am." Each nurse carries multiple tags. The variability in the detection range of readers and the presence of interfering objects makes it impossible to position nurses accurately. Often, numerous readers detect the same tag, or a tag is detected repeatedly between two readers (e.g. between room5 and the hallway – is the nurse in room5 during these times, and just that the hallway sensor is detecting her tag, or is she actually going in and out?). Sometimes the consecutive reports are that a nurse moves from one room to another, with no intermediate report even though all possible routes between these rooms must go through the range of other readers. Thus the application may not be able to identify with certainty a single location for the nurse at all times. A similar application is discussed in [6]. Relational database systems, however, do not allow the modeling or storage of these kinds of uncertain data directly.

In the context of automatic data integration, deep web data in the form of dynamic HTML pages can be used to generate relational data [7]. This is a challenging problem and often the mapping from data in the web page to an attribute in the corresponding tuple is unclear. For example, it may be known that a web page contains prices for data items, and another web page contains a set of numeric values. It is challenging for a program to accurately determine which value maps to the price for a given item. Instead, existing algorithms generate multiple candidates for the value of an attribute, each with a likelihood or probability of being the correct value. Again, due to the lack of support for storing such uncertainty, current applications have to build their own complex models for managing the uncertainty, or just choose the most likely value. Similar issues arise in the domain of integrating unstructured text information with structured databases, such as automatic annotation of customer relationship management (CRM) databases [8], and email search databases. A typical example for text annotation is to determine the topic or product that a particular customer complaint is about. Support for uncertain attribute values would enable the system to retain the uncertainty and thereby produce more reliable answers to queries.

Since conventional database systems have limited support for uncertain data, applications that need uncertain data support are either forced to model uncertain data themselves or simply pick one of the alternative values to store in the underlying database. This leads to a no-win situation: The first option unnecessarily shifts the burden of handling uncertainty to the application and significantly complicates the queries, while the second technique results in a substantial loss of information.

1.1 Types of Uncertain data

The uncertain data in most applications falls in two major categories: *Discrete* and *Continuous* uncertainty.

In case of continuous uncertain data, the uncertainty is specified by a continuous range of values. Applications where continuous uncertain data is natural include sensors, location-based services, spatio-temporal databases, flight tracking, health-care monitoring and financial analysis. In many of these applications, either an interval (with the assumption of uniform distribution) or a *probability density function* (pdf) is specified. For example, GPS devices that give location information are known to have a Gaussian distribution of error around the reported value. Similarly, micro-array data in biological experiments are known to have a Lorentzian distribution of error. The continuous uncertain data is represented and stored in terms of the corresponding distribution's parameters. The Attribute uncertainty model described in Chapter 3 discusses a model which can represent continuous uncertain data. Thus, in case of continuous uncertain data, there exist infinite number of possible values for a given attribute.

On the other hand, in case of discrete uncertain data, we have a fixed number of possible values for an attribute. Example of application domains where discrete uncertain data is common are text annotation and information retrieval. In these domains, the uncertain data is expressed as pairs of possible values along with their probabilities.

1.2 Correlations

A major challenge in handling of uncertain data is the correlations between values. The correlations add a great deal of complexity in query processing and storage of uncertain data. In some cases, the correlations are present in the input data. For example, it is observed that the voltage in sensors and the measured temperature are correlated [2]. Similarly, in case of a location-based service, the uncertainty in the longitude and latitude will be dependent or correlated. In these applications, ignoring the input correlations will lead to a substantial loss of information. Even when the input data is independent, after query execution, the resulting tables may have correlations. If the correlations are not properly tracked, the results of subsequent queries over these correlated uncertain data will be incorrect.

There are two major types of correlations that are present in uncertain data. These are *intra-tuple* and *inter-tuple* correlations. As the name suggests, intra-tuple correlations refers to the correlations between two attributes of the same tuple. In the location example, the attribute latitude and longitude will be correlated. On the other hand, inter-tuple correlations are correlations across tuples. As an example, consider a table whose primary key is uncertain. Assume that particular value k_0 occurs in the primary key attribute of multiple tuples. All these tuples will be correlated, as presence of k_0 in any tuple will imply the absence of k_0 in all other tuples.

1.3 Query Semantics

Query semantics for precise data are well defined. The operations of selection, projection and joins are widely used for querying traditional databases. Unfortunately, the semantics of these operations are not clear for uncertain data. Consider, for example, a selection query $\sigma_{x>5}(T)$ over a precise database table T. This query will return all the tuples in which the attribute x is greater than 5. If the attribute x is uncertain, this selection condition can be *partially* true for many tuples. For each tuple, instead of a precise true or false answer, now we will have a real value giving the probability that the tuple passes the selection condition.

Given this problem, alternative semantics for uncertain data have been proposed. Chapter 3 discusses a query semantics where a probabilistic threshold is used to convert a real probability value into a boolean. In Chapter 8, we discuss the *Possible Worlds Semantics* that is used to define probabilistic operations over uncertain data.

1.4 Goals

There are numerous applications that will immediately benefit from advancements in probabilistic data management. Given the need for managing uncertain data, it is important to develop probabilistic models that can express and manage uncertain data at the database level. One of the contributions of this dissertation is the development of uncertainty management model that is expressive enough to handle the different types of uncertain data presented in Section 1.1. A challenge in developing such a model is the issue of correlations discussed in Section 1.2. Implementation of probabilistic models in the database will greatly simplify the existing applications by abstracting away the complexities of uncertain data management. Introducing uncertainty into databases brings about many challenges including issues of query semantics, evaluation, and efficiency. A major focus of this dissertation is efficient execution of queries with the help of database techniques such as indexing. Research and development of a database system that supports uncertain data will advance scientific understanding and enable future work in a variety of fields. But whether emerging applications use databases simply as an information storage technology rather than an effective data management solution depends on to what extent they can reason about and make use of the uncertainty of data directly.

The rest of the thesis is organized as follows. Chapter 2 presents the related work done in this area. The remainder of the thesis is composed of two parts. The first part deals with efficient execution of queries over uncertain data under the attribute uncertainty model discussed in Chapter 3. Chapter 4 summarizes the important contributions of our work. Chapter 5 develops algorithms for join queries. Chapter 6 presents two indexing structures for categorical uncertain data. Query selectivity estimation for uncertain data is discussed in Chapter 7. In the second part of the thesis, we present a new unified model for uncertain data (Chapter 8) and its implementation and related issues (Chapter 9). Future work is presented in Chapter 10 and Chapter 11 concludes this thesis.

2 LITERATURE REVIEW

In this chapter, we review the related work done in the field of uncertainty management in databases. The work done in the field of modeling and querying of uncertain data is summarized. Related work done in the area of traditional (precise) data management systems is also presented.

2.1 Models for Uncertain Data

There has been a great deal of work on the development of models for representing uncertainty in databases [9]. Two main approaches for modeling uncertain data have emerged in this field: tuple uncertainty [1, 10] and attribute uncertainty [11]. In tuple uncertainty model, a probability value is attached to each tuple which represents the probability of that tuple being present in the database. In attribute uncertainty model, uncertainty is associated with individual attributes, as opposed to the complete tuple.

Barbará *et al.* [12] and Dey *et al.* [13] proposed one of the first probabilistic models. Building on their work, many robust models for managing tuple uncertainty have been proposed recently. A significant challenge when modeling uncertain data is tracking arbitrary correlations both within and between tuples. These dependencies are not only present in real-world data, they are more commonly introduced by applying operations to independent base data. Benjelloun *et al.* have proposed a novel technique that combines uncertainty with data lineage to solve this problem [10]. The ProbView system [14] took a similar approach by propagating the formulae necessary for evaluating the resulting probabilities. Sen *et al.* have more recently proposed an alternative approach to represent tuple correlations using probabilistic graphical models [15]. They use factored representations of the relations to represent their dependencies. Antova *et al.* developed a compact representation called world-set decompositions which captures the correlations in the database by representing the finite sets of worlds [16]. Dalvi *et al.* introduced safe plans [1, 17] in an attempt to avoid probabilistic dependencies in queries.

Many of the currently active research efforts in uncertain data management favor tuple uncertainty models. This is partly because their simplicity more closely resembles the standard relational model. Because each attribute is a single value, relations in tuple uncertainty models also abide the first normal form (1NF). This greatly reduces the complexity of defining query operations over such models. Indeed, one problem with modeling uncertain data is burdening the user with its underlying complexity [18, 19].

Most of the work in this dissertation is based on the attribute uncertainty model proposed in [11]. This model is further discussed in Chapter 3. Similar probabilistic models were also proposed in moving-object environments [4, 20] and in sensor networks [2]. Discussion of uncertainty in other data types can be found in [21]. Probabilistic databases have also been recently extended to semi-structured data [22] and XML [23].

An important area of uncertain reasoning and modeling deals with fuzzy sets [24, 25]. Fuzzy databases is a well studied area and a lot of work has been done on query evaluation and indexing [26–29]. In this dissertation, we do not assume a fuzzy model for the data. Instead, the focus is on probabilistic modeling of data uncertainty. Due to the underlying difference in the nature of the uncertainty in the probabilistic model, the existing work on fuzzy databases is not applicable.

2.2 Probabilistic Queries

Probabilistic queries are classified as value-based (return a single-value) and entity-based (return a set of objects) in [11]. Evaluation of probabilistic range queries is discussed in [1, 4, 11, 20]. Nearest-neighbor queries are discussed in [11]. In [1, 11], aggregate value-queries evaluation algorithms are presented. An index called Probabilistic Threshold Index was proposed in [30] that can be used to efficiently execute some classes of probabilistic queries. Ljosa et al [31] discussed k-nearest neighbor queries for uncertain data. Apart from this work, there has been other work by [32, 33] on indexing pdfs.

2.3 Probabilistic Join Queries

Chapter 5 discusses the probabilistic join queries over uncertain data. These queries are not addressed before in the context of attribute uncertainty. Although, [30] did examine the issues of query efficiency, the discussion was limited to range queries.

There is a rich vein of work on interval joins, which are usually used to handle temporal and one-dimensional spatial data. Different efficient algorithms have been proposed, such as nested-loop join [34], sort-merge join [34, 35], partition-based join [36], and index-based join [37]. Recently the idea of implementing interval join on top of a relational database is proposed in [38]. All these algorithms are developed for precise data and thus do not utilize any probabilistic information during the pruning process, and thus potentially retrieve many false candidates. Hence, they are not useful for join processing over uncertain data.

2.4 Indexing Uncertain Categorical Data

Indexing techniques for categorical uncertain data are discussed in Chapter 6. Most of the earlier work [30] is only applicable for continuous numeric domains. Burdick *et al.* consider the problem of OLAP over uncertain data [8]. They model the uncertainty from text annotators as tuple uncertainty and support aggregation queries over this data. Our model differs from theirs in the sense that they limit the classification of the text to one class at a time, whereas we capture multiple classes. Hence we represent the uncertainty as attribute uncertainty whereas they model it using tuple uncertainty. In their work, the value of interest within the domain is predetermined, while we make no assumptions about the value of interest.

Indexing for set valued attributes has been extensively considered in the literature. Faloutsos developed the notion of signature files to index sets [39]. Indexing set-valued attributes in databases has been considered by Mamoulis [40]. Mamoulis *et al.* also applied indexing for computing join queries over set-valued indexes [41]. The indexing problem presented in Chapter 6 is a generalization of the set model where we have probability values in addition to the sets, and is the first work to address the problem of indexing uncertain categorical data.

2.5 Query Selectivity Estimation

Databases rely on automatic optimization of queries. One of the key ingredients for optimization is estimation query result set size (selectivity estimation). There is a rich body of work on selectivity estimation for traditional relational database management systems. Most approaches for selectivity estimation on precise data use histograms. Poosala et al [42] proposed a taxonomy to capture all previously proposed histogram approaches. These approaches are not applicable for uncertain data because both the queries and the underlying data types for uncertain data differ greatly from traditional data and queries. Chapter 7 presents our techniques for query selectivity estimation over uncertain data.

2.6 Uncertainty Management Systems

Several systems that handle uncertainty in data have been recently proposed. Trio [3] is uncertainty management system based on tuple uncertainty model. In Trio, uncertainty of the data and data lineage are first-class citizens. Trio is based on an extended relational model called ULDBs, and it supports a SQL-based query language called *TriQL*.

MayBMS [43] is a probabilistic database management system developed as an extension of the PostgreSQL [44]. Although MayBMS can handle attribute uncertainty and correlations between attributes, it is limited to discrete uncertain data. Other systems for managing discrete uncertain data include MystiQ [17] and [15].

We have developed Orion [45], which is a state-of-the-art uncertain database management system with built-in support for probabilistic data as first class data types. Orion supports both attribute and tuple uncertainty and can handle both discrete and continuous uncertain data. Chapter 9 further discusses the implementation of Orion.

In the next chapter, we present the uncertainty model that is assumed in the first part of the dissertation.

3 MODELING BACKGROUND

This chapter describes the two major models for managing uncertain data: attribute and tuple uncertainty models. The first part of this thesis is based on the attribute uncertainty model proposed in [11]. In the second part of this thesis, we present our new model (Chapter 8), which is the extension of the basic attribute uncertainty model presented in this chapter to include Possible World Semantics.

3.1 Attribute Uncertainty Model

To model the uncertainty present in a data item, a data scheme known as the *Attribute uncertainty model* was proposed in [11]. This model assumes that individual attributes, as opposed to complete tuples, are uncertain.

The attribute uncertainty model assumes that each data item can be represented by a range of possible values along with the distribution of values over this range. Formally, assume that each tuple of interest consists of an uncertain attribute a. If there are more than one uncertain attributes within the same tuple, they are assumed to be independent of each other. The domain of the uncertain attribute can be continuous (e.g. real-valued) or discrete (e.g. integer). as shown in Figure 3.1.

The *probabilistic uncertainty* of a consists of two components:

Definition 3.1.1 *The* uncertainty interval *of an item a, denoted by U_a, is an interval $[l_a, r_a]$ where $l_a, r_a \in \Re, r_a \geq l_a$ and $a \in U_a$. The range of R_a of a is defined as $R_a = r_a - l_a$.*

Definition 3.1.2 *The* uncertainty pdf *of a, denoted by $f_a(x)$, is a* probability distribution function *(pdf) of a where $f_a(x) = 0$ if $x \notin U_a$. Additionally, we have*

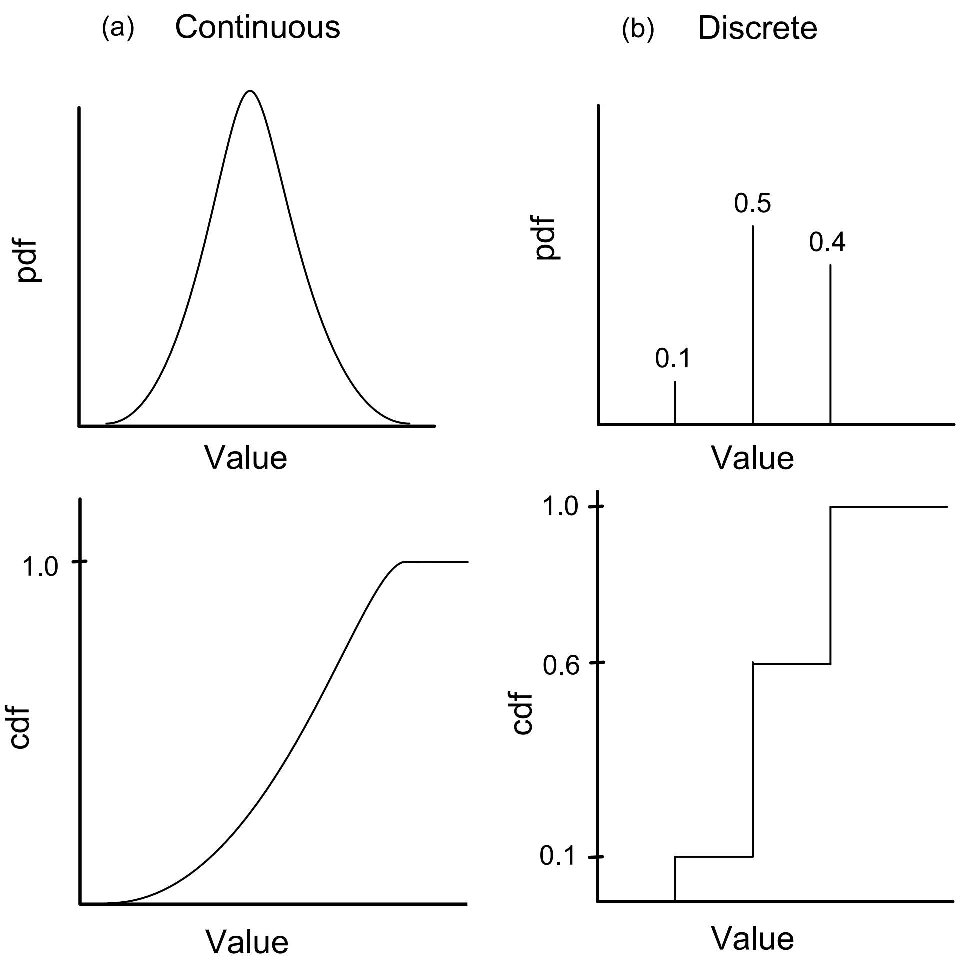

Figure 3.1. Example of a continuous (a) and discrete (b) uncertain attribute. The top graphs show the probability distribution function for the two example distributions. The bottom graphs shows the cummlulative distribution function for the two uncertain values.

$\int_{l_a}^{r_a} f_a(x)dx = 1$ for continuous variable a and $\sum_{U_a} f_a(x) = 1$ in the case when a is discrete.

Definition 3.1.3 *In addition to the pdf $f_a(x)$, we can also define a* cumulative distribution function *(cdf) $F_a(x)$, which is defined as $F_a(x) = \int_{-\infty}^{x} f_a(x)dx$.*

Notice that $F_a(x) = 0$ if $x < l_a$ and $F_a(x) = 1$ if $x > r_a$. Note that, similar to the continuous case, we can also define the pdf and cdf functions in case of a *discrete attribute* by replacing the integral with a sum in the above definitions.

Figure 3.1(b) shows the pdf and cdf functions for an example discrete distribution.

The exact realization of this model is application-dependent. For example, in modeling sensor measurement uncertainty, U_a is an error bound and $f_a(x)$ is a Gaussian distribution. In modeling moving objects, Wolfson et al. [4] suggested a bounded uncertainty model where each moving object only reports its location if its current location deviates from its reported location by more than d, so that at any point of time the uncertainty of the location value stored in the system has uncertainty of not more than d.

The specification of uncertain pdf is also application-specific. For convenience, one may assume that the uncertainty pdf $f(x)$ is a uniform distribution i.e., $f(x) = \frac{1}{r_a - l_a}$ for $a \in [l_a, r_a]$; essentially, this implies a "worst-case" scenario where we have no knowledge of which point in the uncertainty interval possesses a higher probability. In sensor networks, Deshpande et al. [2] assumed the reading of each sensor node is a Gaussian distribution parameterized with a mean and variance value. They also suggested that these Gaussian distributions can be constructed through machine learning algorithms, such as [46]. Another example is a triangular distribution.

Note that although the uncertainty model described here is presented for one-dimensional data, its concept can be extended to multiple dimensions.

3.2 Tuple Uncertainty Model

The tuple uncertainty model [1, 10, 14] assumes that the complete tuple is uncertain. In a sense this model assumes that a joint probability distribution exists for all the attributes in a tuple (and hence all attributes are correlated). A probability value is attached to each tuple which represents the probability of that tuple being present in the database. In addition, multiple tuples can be grouped together to form an *x-tuple* [10]. The tuples present inside a *x-tuple* are called alternatives and they represent mutually exclusive values for the tuple. In addition to x-tuples, a *lineage* is also stored for each tuple.

For discrete domains, the tuple uncertainty model is more powerful than attribute uncertainty model, as it can express correlations between uncertain attributes and tuples. The major disadvantage is that tuple uncertainty model is unable to handle continuous uncertain data. The model presented in Chapter 8 bridges the gap between the two models and does not suffer from their limitations.

3.3 Probabilistic Queries

Once we define the data representation for uncertain data, we need to define the query semantics over this data. The standard database operations, such as selections, projections and joins are not directly applicable over uncertain data.

Consider a simple selection query with condition $x > 1$. If x is precise or certain, this condition evaluates to a boolean and hence each tuple in the input relation is either present in the output relation or not. On the other hand, if x is uncertain, this condition can not be evaluated to a simple true or false. At best, we can evaluate the probability that this condition holds i.e. $Pr(x > 1)$. Because of this, it is not possible to simply extend the usual definitions of selection, projection, join operations for uncertain data.

To overcome this problem, in attribute uncertainty model, we define a probabilistic threshold for each query. This threshold is used to transform the probabilistic conditions to boolean conditions which can then be used for query processing. In the above example, if the query threshold is 0.5, $x > 1$ is transformed into the boolean condition $Pr(x > 1) > 0.5$. This boolean condition is evaluated for each tuple to decide whether or not to include that tuple in the output relation. These queries are also refered to as *Probabilistic Threshold Queries*. The query semantics for probabilistic queries in the tuple uncertainty model are discussed in [10].

3.4 Chapter Summary

In this chapter, we introduced the two major models for uncertain data. The first part of this dissertation is based on the attribute uncertainty model, but in most cases, the techniques for attribute uncertainty model can also be extended to tuple uncertainty model. In the next chapter, we summarize the major contributions of this thesis.

4 CONTRIBUTIONS

As discussed in Chapter 3, there are two major models for managing uncertain data: tuple and attribute uncertainty models. Most of the earlier work done on these two models was disjoint and used different query semantics. The work presented in this thesis can be considered as a bridge between attribute and tuple uncertainty models. The model assumed in first part of this thesis is that of attribute uncertainty, but the results are equally applicable to the tuple uncertainty model as well.

- The join operation in the tuple uncertainty model is straight forward and follows from the basic relational model. However, for attribute uncertainty in general and continuous data in particular, the semantics of join is not obvious. In Chapter 5, we extend the semantics of the join processing for the attribute uncertainty model by presenting the concept of probabilistic threshold join queries. We discuss how performance of probabilistic threshold join queries can be improved considerably by using various pruning techniques.

- Previous work on indexing for uncertain data was focused solely on continuous (numeric) uncertain data. In Chapter 6, we propose two index structures for categorical (discrete) uncertain data. The techniques discussed are applicable to both attribute and tuple uncertainty models.

- We discuss efficient techniques for selectivity estimation (an important ingredient for query optimization) of probabilistic threshold queries over uncertain data in Chapter 7. We also show that our techniques are not only applicable to selectivity estimation, but also to query processing, by showing how they can be used for efficiently processing probabilistic k-nearest neighbor queries. Again, the algorithms presented are applicable to both attribute and tuple uncertainty models.

- A unified model that can handle both attribute and tuple uncertainty is presented in Chapter 8. This is the first model that can represent both continuous and discrete uncertain data with arbitrary correlations natively at the database level. This model is consistent and closed under the possible worlds semantics. Most of the work presented in earlier chapters is applicable to this new model with little or no changes.

- We present Orion, a general purpose uncertainty management system in Chapter 9. Orion is implemented inside PostgreSQL and supports our new model presented in Chapter 8. Orion has efficient access methods, improved query optimization and is integrated with PL/R for graphical visualization of uncertain data.

In the next chapter, we discuss the semantics of join processing for the attribute uncertainty model and various pruning techniques that can be used to significantly improve the performance of threshold join queries.

5 JOIN PROCESSING FOR ATTRIBUTE UNCERTAINTY

In this chapter we address join queries over uncertain data. Earlier work on handling joins is mainly done under the tuple uncertainty model (See Chapter 3). The tuple uncertainty model cannot handle continuous uncertain data which is the focus of this chapter. We propose semantics for the join operation, define probabilistic operators over uncertain data, and propose join algorithms that provide efficient execution of probabilistic joins. The chapter focuses on an important class of joins termed probabilistic threshold joins that avoid some of the semantic complexities of dealing with uncertain data. For this class of joins we develop three sets of optimization techniques: item-level, page-level, and index-level pruning. These techniques facilitate pruning with little space and time overhead, and are easily adapted to most join algorithms. We verify the performance of these techniques experimentally.

5.1 Introduction

Incorporating uncertainty into databases brings about many challenges including issues of query semantics, evaluation, and efficiency. The problem of the semantics of query processing and efficient evaluation of queries for tuple uncertainty have been discussed in earlier work [1]. There has also been some work on simple types of queries (range and nearest-neighbors only) for databases with attribute uncertainty [11, 30]. There is, however, no prior work on more complex queries in the specific area of uncertain attribute data.

There is ongoing research interest in systems that acquire information from the external world. Sensor nets, for example, allow physical entities such as temperature, pressure and voltage to be collected through large numbers of inexpensive sensors [2].

Location devices like cell phones and GPS-equipped devices also allow phone users' and vehicles' locations to be obtained easily. The massive amounts of information collected about the physical world enables the development of novel applications that base their decisions on these physical data.

Unfortunately, joining "natural data" from the sensing instruments is not straightforward, due to the *uncertainty* inherent with the data obtained in the external dynamic environment. In particular, while current technologies only allow data to be acquired in a discrete manner, entities like temperature and location values are continuously evolving. Since the information during the inter-arrival time of data samples is not provided to the system, there is uncertainty between the database value and the actual value. This problem can be aggravated by network issues, where data packets can be delayed or even lost, especially in a wireless network [2, 11]. As a result, the database values may have a large discrepancy compared with the actual values.

As a more concrete example, consider a scientific application where an equality join query is issued over two sets of temperature values (obtained from two sensor networks in separate geographical regions) to discover the pairs of sensors that report the same temperature value. Figure 5.1(a) shows two tables, A and B, storing two attributes $(ID, Temp)$, which represent the temperature values $Temp$ recorded by sensors with names given by ID. Suppose we would like to perform an equality join over the temperature attributes to determine which pairs of entities in A and B record the same temperatures. Joining pairs are shown connected by a line in (a). This result is incorrect if we consider the true values of the sensors given by Figure 5.1(b): since the actual value for A_1 is different from that of B_1, A_1 should not be paired with B_1. Instead, A_1 matches B_2, where both temperature values equal to 11^oF. Thus there is a *false positive* in the result – (A_1, B_1) is wrongly returned to the user. Figure 5.1(b) also shows that A_2 should be matched with B_3. Consequently, (A_1, B_2) and (A_2, B_3) are not returned to the user, resulting in two *false negatives*. As we can see, the join result returned by the database is significantly different from the actual result. If this result is further processed by the application, the error may propagate in the analysis and invalid conclusions/decisions may be made.

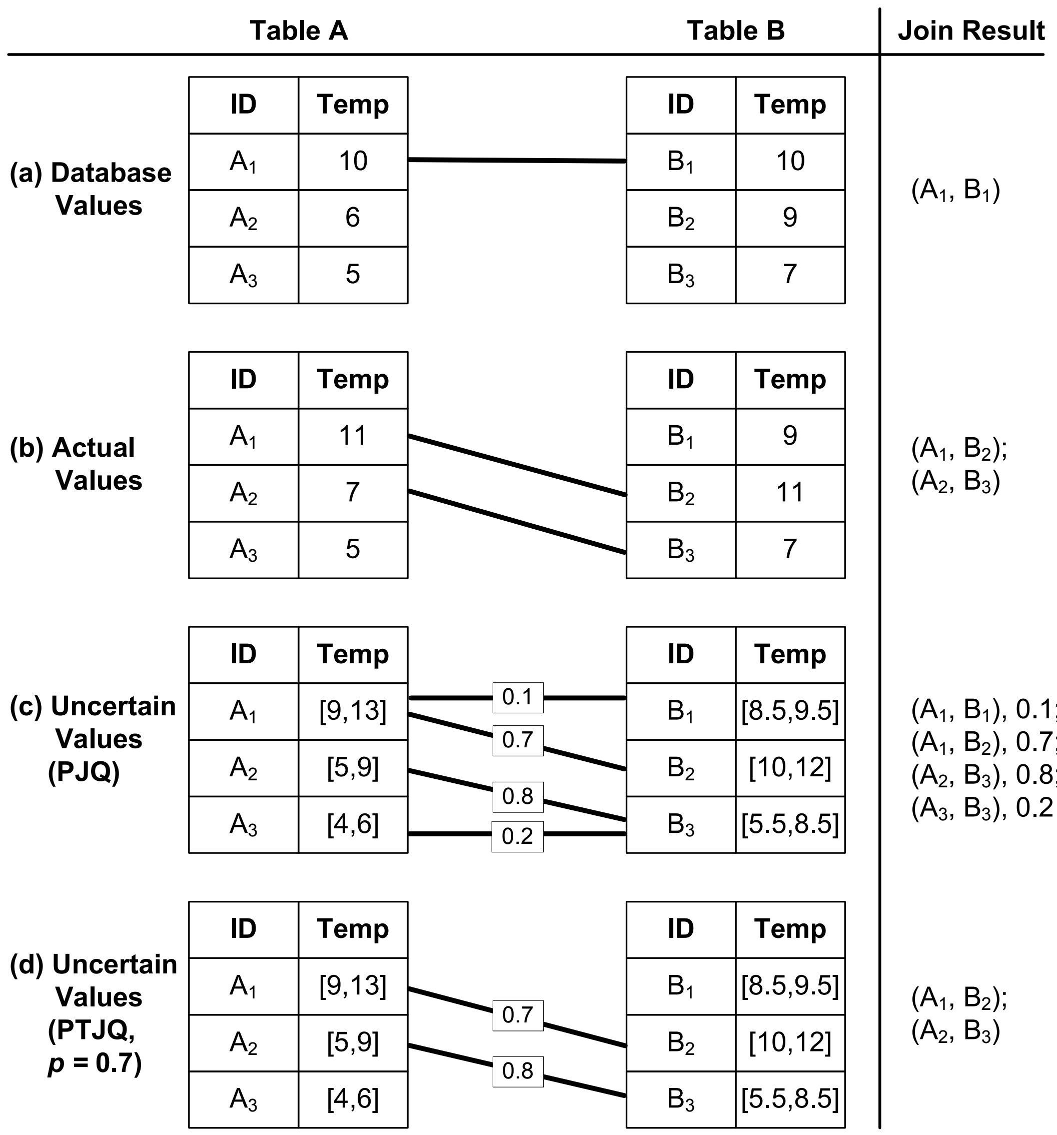

Figure 5.1. Illustrating join over uncertain data.

To avoid incorrectness in query answers, the idea of using an *uncertainty model* rather than a single numerical value to describe an item was proposed in [11]: each item is associated with a range of possible values and a probability density function (pdf) that describes the probability distribution of the value within the range. To address the above uncertainty problem, an uncertainty interval can be a fixed bound d, which is a result of negotiation between the database system and the sensor; if the system does not receive any update from the sensor, it can assume that the sensor's current value must be between $[v - d, v + d]$, where v is the value of the sensor last reported to the server [4]. The pdf of the sensor value within the range may be obtained through machine learning techniques [2]. By incorporating the notion of uncertainty into data values, imprecise, rather than exact, answers are generated. Each join-pair is associated with a probability to indicate the likelihood that the two tuples are matched. We use the term *Probabilistic Join Queries* (PJQ) to describe these types of joins over uncertain data.

Figure 5.1(c) illustrates the idea of PJQ. Each temperature attribute stores a range that encloses the data value, together with a pdf that describes the distribution (not shown here). Each tuple-pair is associated with a probability that indicates the likelihood of the join. Notice that both (A_1, B_2) and (A_2, B_3) are now included in the result. In this example, the false negative problem vanishes. Also, we have a 0.7 and 0.8 confidence for these pairs. On the other hand, the false positive, (A_1, B_1), remains in the result, and a new false positive, (A_3, B_3), is introduced. However, both false positives have a relatively low probability (0.1 and 0.2 respectively), suggesting to the user that these two matches are less likely to occur.

How are these probability values computed? To answer this, we must understand the semantics of join operators for uncertainty. The notions of equality and inequality have to be extended to support uncertain data. We will address the new definitions of comparison operators for the uncertain data model. Furthermore, we demonstrate how it is possible to relax the requirements for comparison operators, in order to allow more flexibility in specifying accuracy requirements of joins over uncertainty.

Another dimension of our study deals with the performance issues of joins over uncertainty. We observe that although the answer probabilities are useful, it is not always necessary to know their exact values. Often the user is only concerned about whether the probability value exceeds a given threshold. If the probability that the join pair meets the join condition exceeds the threshold, it is included in the result, otherwise the pair is not included. This threshold can either be user-specified or a system parameter.

We term this variant of probabilistic join queries, which only returns tuple pairs when their probabilities exceed a certain threshold as *Probabilistic Threshold Join Queries* (PTJQ). An example of PTJQ is shown in Figure 5.1(d), where we assume the user is only interested in tuple pairs whose probabilities exceed threshold $p = 0.7$. As a result, the two pairs with low probability values (0.1 and 0.01) are not included in the answer. Compared with Figure 5.1(c), PTJQ returns fewer false negatives.

We focus on threshold joins and develop various techniques for the efficient (in terms of I/O and CPU cost) algorithms for PTJQ. In particular, we develop three pruning techniques: (1) *item-level pruning*, where two uncertain values are pruned without evaluating the probability; (2) *page-level pruning*, where two pages are pruned without probing into the data stored in each page; and (3) *index-level pruning*, where all the data stored under a subtree is pruned. These techniques incur a small space and time overhead, and can be augmented to existing join algorithms easily.

This can be achieved by employing an indexed nested loop join algorithm which requires an index over uncertain data. Previously, Cheng et al. [30] proposed "uncertainty indexes" designed for answering probabilistic range queries. The indexes built on the uncertain attribute values are non-traditional and were designed for answering probabilistic range queries. We illustrate how uncertainty indexes can be used to support various uncertainty join operators. We consider two scenarios: (1) an uncertainty index is available only on one relation, and (2) uncertainty indexes are available on both relations. We analyze the cost of indexed nested loop join for PTJQ. Extensive experiments are conducted to evaluate the effectiveness of using uncertainty indexes over traditional indexes for supporting joins.

The contributions of this chapter are:

- We extend the semantics of join operators over exact, single-valued data to uncertain data.

- We present the concept of probabilistic join queries (PJQ) and illustrate how they can be evaluated.
- We illustrate how probabilistic threshold join queries (PTJQ), a variant of PJQ that constrains on the answers based on their probability values, can improve the join performance significantly based on various pruning techniques.
- Finally, we present experiments validating the efficiency of our approach.

In Section 5.2, we define the notion of join operators over uncertainty. Section 5.3 presents item-level pruning techniques for each join operator. In Section 5.4, we study how the performance of join can be further improved through page-level and index-level pruning techniques. Experimental results are presented in Section 5.5, and Section 5.6 concludes the chapter.

5.2 Comparing Uncertain Values

We assume the attribute uncertainty model discussed in Chapter 3. This model assumes that each data item can be represented by a range of possible values and their distributions. Formally, assume each tuple of interest consists of a real-valued attribute a. Note that a is treated as a continuous random variable, and it is assumed that each uncertain attribute value is mutually independent. In this chapter, we refer to the uncertainty interval, pdf and cdf of an uncertain attribute a as $a.U$ (i.e $[a.l, a.r]$), $a.f(x)$ and $a.F(x)$ respectively.

We next present the definitions of comparison operators over uncertainty, based on which probabilistic join queries are defined.

5.2.1 Uncertainty Comparison Operators

In order to evaluation join conditions over uncertain attributes, it is first necessary to define operators for this data type. Consider the equality of two uncertain-values a and b, which are modeled with probabilistic uncertainty. Since a and b are not single values, traditional notions of comparison operators (such as equality and inequality) cannot be used. Due to the range of possible values for each data item it is not immediately obvious whether the two are equal in value or not. If there is no overlap in their range, clearly they cannot be equal. However, if there is an overlap, there is the possibility that the two could be equal. We would like to determine the likelihood of them being equal. In this section, we extend the definitions of common comparison operators to support uncertain values. In particular, we express "imprecision" in these operators in terms of probability values.

To understand "equality" for uncertain data, consider Figure 5.2 where the overlap between $a.U$ and $b.U$ is $[a.l, b.r]$. A first thought is that at any point x_0 inside $[a.l, b.r]$, a is equal to b with probability $a.f(x_0) \cdot b.f(x_0)\delta x$ (where δx is infinitesimal), and so the probability a equals to b is simply $\int_{a.l}^{b.r} a.f(x)b.f(x)dx$. However, this is incorrect: both $a.f(x)$ and $b.f(x)$ are continuous functions, thus the probability that a and b are equal to x_0 is zero. Consequently, the probability of equality is always zero, and a and b can never be equal.

Given that the exact values for these data items are not known, the user is more likely to be interested in them being very close in value rather than exactly equal. Naturally, how close they are should be determined by the user. Based upon this observation, we define equality using a parameter, called *resolution* (c), as: a is equal to b if they are within c of each other i.e., $b - c \leq a \leq b + c$ or $a - c \leq b \leq a + c$:

Definition 5.2.1 Equality ($=_c$): *Given a resolution c, a is equal to b with probability*

$$P(a =_c b) = \int_{-\infty}^{\infty} a.f(x) \cdot (b.F(x + c) - b.F(x - c))dx$$

Essentially, a is equal to b when $a = x_0$ if b is in the range $[x_0 - c, x_0 + c]$, with a probability of $b.F(x_0 + c) - b.F(x_0 - c)$, or $\int_{x_0-c}^{x_0+c} b.f(x)dx$. Figure 5.2 illustrates this definition of equality, where we can see a and b only join in $[a.l - c, b.r + c]$. Let $l_{a,b,c}$ be $\max(a.l - c, b.l - c)$ and $u_{a,b,c}$ be $\min(a.u + c, b.u + c)$. For the case that the two intervals are within distance c of each other, Definition 5.2.1 can be rewritten as:

$$P(a =_c b) = \int_{l_{a,b,c}}^{u_{a,b,c}} a.f(x)(b.F(x+c) - b.F(x-c))dx \tag{5.1}$$

where the overlap of $a.U$ and $b.U$ is given by $[l_{a,b,c}, u_{a,b,c}]$. We assert without proof that our definition of equality is symmetric i.e., $P(a =_c b)$ yields the same value as $P(b =_c a)$.

Notice that $P(a =_c b)$ is zero for the case that we are fully confident that a and b cannot be joined. This happens when $b.r + c < a.l$ or $a.r + c < b.l$. This indicates that a and b have no chance of being equal. Based upon the definition of equality, we can define **Inequality** as follows:

Definition 5.2.2 Inequality ($\neq_c$): *Given a resolution c, a is not equal to b with probability*

$$\begin{aligned} P(a \neq_c b) &= 1 - P(a =_c b) \\ &= 1 - \int_{-\infty}^{\infty} a.f(x) \cdot (b.F(x+c) - b.F(x-c))dx \end{aligned}$$

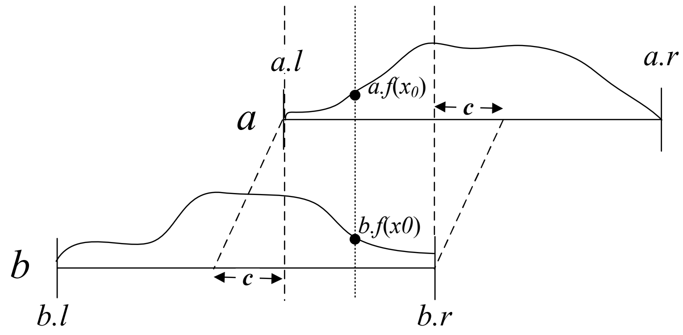

Figure 5.2. Illustrating Comparison Operations for uncertain values a and b.

To address the question "Is a greater than' b?", let us look at Figure 5.2. In $[b.r, a.r]$, b cannot be larger than a, since $b.f(x)$ is 0 when $b > b.r$. Thus if a is within $[b.r, a.r]$, it is larger than b with probability $\int_{b.r}^{a.r} a.f(x)dx$, or $1 - a.F(b.r)$. At any point x_0 inside $[a.l, b.r]$, a is larger than b with a probability $a.f(x_0)b.F(x_0)$, where $b.F(x_0)$ is the probability that b is less than x_0. Therefore, in $[a.l, b.r]$, the probability that a is larger than b is given by $\int_{a.l}^{b.r} a.f(x)b.F(x)dx$. There is no need to consider $[b.l, a.l]$, because b is always less than a when b is in this region. To sum up, the probability that a is larger than b in Figure 5.2 is:

$$\int_{a.l}^{b.r} a.f(x)b.F(x)dx + 1 - a.F(b.r)$$

Upon considering all possible scenarios of overlap between $a.U$ and $b.U$, we obtain the definition of ">":

Definition 5.2.3 Greater than $(>)$*:* $a > b$ *with probability* $P(a > b)$

$$= \begin{cases} \int_{\max(a.l,b.l)}^{b.r} a.f(x)b.F(x)dx + 1 - a.F(b.r) & a.l \leq b.r < a.r \\ \int_{\max(a.l,b.l)}^{a.r} a.f(x)b.F(x)dx & b.l \leq a.r \leq b.r \end{cases}$$

For the case that a lies entirely to the left of b, i.e. $a.r < b.l$, $P(a > b) = 0$. Also, for the case that a lies entirely to the right of b, i.e. $a.l \geq b.r$, $P(a > b) = 1$.

Note that in a continuous-valued domain, $P(a > b)$ is the same as $P(a \geq b)$ because a can never be exactly equal to b. In the sequel we will not discuss $a \geq b$.

In a similar manner, we can redefine $<$ as follows.

Definition 5.2.4 Less than $(<)$*:* $a < b$ *with probability* $P(a < b)$

$$= \begin{cases} \int_{a.l}^{b.r} a.f(x)(1 - b.F(x))dx & b.l < a.l \leq b.r \\ a.F(b.l) + \int_{b.l}^{\min(a.r,b.r)} a.f(x)(1 - b.F(x))dx & a.l \leq b.l \leq a.r \end{cases}$$

Again, for the case that a lies entirely to the left of b, i.e. $a.r < b.l$, $P(a < b) = 1$. Also, for the case that a lies entirely to the right of b, i.e. $a.l \geq b.r$, $P(a < b) = 0$. Also, since $P(a < b)$ is the same as $P(a \leq b)$, and so we will not discuss $a \leq b$.

We can see from that comparison over uncertainty is imprecise. The degree of imprecision, represented by probability values, indicates the confidence of the comparison result. For example, if $P(a > b) = 0.01$, then a is unlikely to be greater than b.

It is worth mentioning that the definitions of comparisons for uncertainty with continuous uncertainty pdfs can be extended to support discrete pdfs.

5.2.2 Comparing Uncertainty with Certainty

Some situations may require the join of uncertain values with "certain" values. For example, a user can join the current locations of people with locations of buildings (where the locations are fixed), in order to find out which persons are in which buildings. In general, operators between an uncertain value a and a certain value $v \in \Re$ can be defined as:

$$\begin{aligned}
P(a =_c v) &= \int_{v-c}^{v+c} a.f(x)dx = a.F(v+c) - a.F(v-c) \\
P(a \neq_c v) &= 1 - P(a =_c v) = 1 - a.F(v+c) + a.F(v-c) \\
P(a > v) &= 1 - a.F(v) \\
P(a < v) &= a.F(v)
\end{aligned}$$

which can be treated as special cases for the definitions of uncertainty operators.

5.2.3 Probabilistic Join Queries

Once the comparison operators for uncertainty are defined, We can now formulate the join problem. Suppose we have two tables R and S containing m and n tuples respectively. Both tables contain an uncertain attribute on which the join will be performed. We name the uncertain attribute of the ith row as R_i for table R, and as S_i for table S. Then the Probabilistic Join Query (PJQ) is defined as follows.

Definition 5.2.5 *Given an uncertainty comparator θ_u (where θ_u is any one of $=_c, \neq_c, >, <$), a* **Probabilistic Join Query (PJQ)** *returns all tuples $(R_i, S_j, P(R_i \theta_u S_j))$ where $i = 1, \ldots, m$, $j = 1, \ldots, n$ and $P(R_i \theta_u S_j) > 0$.*

Essentially, a PJQ returns join pairs with a non-zero probability of meeting the join condition along-with the associated probability. It is more informative than traditional join in which the confidence of the join, expressed in terms of probabilities. Unfortunately, it involves expensive operations – especially in the process of finding the probabilities of the join-pairs using our definitions of uncertainty comparators. In the next section, we will examine how a PJQ can be implemented efficiently. Notice that the probability returned by the join is in effect the probability of the corresponding tuple being part of the join result table. Thus the result of a PJQ over a table with uncertain attribute data is a table with tuple uncertainty. Since the model we have considered for uncertainty does not incorporate tuple uncertainty, this result falls outside the model. This is not desirable since we would like to have a closed model in order to enable query composibility.

There are two alternatives for addressing this problem. The first is to treat the probability simply as another attribute of the query result. The new attribute is intrinsically defined with the domain of probability values. This requires users to either be aware that a new attribute will be added, or to explicitly add the probability attribute in their SELECT clauses. The second alternative is to not generate tuples with probabilistic values. Instead, each tuple is either part of the result or not. In this case, we have to convert each probabilistic comparison operator into a boolean comparison operator. This is achieved through the specification of a cut-off threshold probability. With this minor change, we define a join to be *Probabilistic Threshold Join Query* (PTJQ). It has an additional constraint that only join pairs whose probabilities exceed a user-defined threshold are returned.

Definition 5.2.6 *Given an uncertainty comparator θ_u (where θ_u is any one of $=_c, \neq_c, >, <$), a* **Probabilistic Threshold Join Query (PTJQ)** *returns all tuples (R_i, S_j) such that $i = 1, \ldots, m$, $j = 1, \ldots, n$, and $P(R_i \theta_u S_j) > p$, where $p \in [0, 1]$ is called the probability threshold.*

A PTJQ only returns join pairs that have probabilities higher than p. Another difference from PJQ is that PTJQ only returns the pairs, (R_i, S_j), but not the actual probability values. In the sequel, we will explain how these two differences are exploited for performance improvement.

5.3 Evaluating PTJQ with Interval Join

To evaluate a PTJQ, common methods like block-nested-loop join and indexed-loop can be used. The advantage of these algorithms is that they have been implemented in typical database systems, hence the system requires little modification to support joins over uncertain data. However, we will demonstrate that these join techniques can be improved by a number of novel techniques.

Figure 5.3 illustrates a possible approach of using traditional join algorithms for processing uncertainty. As shown in Step 2, the main idea is to join the uncertainty intervals with an interval-join algorithm, and store the possible candidates are stored in a set, C. Subsequently, the pdf/cdf information is used to calculate the probability of each candidate pair, and those that have probability greater than p are retained in the result (Step 3). In the rest of this section, we examine these two steps in more details.

The exact method used in Step 2 depends on the type of the comparison operator. For equality over two uncertain intervals $R_i.U$ and $S_j.U$, we can eliminate intervals that do not overlap after considering the resolution c (i.e., pairs that satisfy $R_i.r + c < S_j.l$ or $S_j.r + c < R_i.l$). According to Definition 5.2.1, these tuples have zero chance of being paired up. Thus, any I/O-efficient *overlap join* algorithms over intervals (e.g., [38]) can be used. For $>$, we can immediately eliminate (R_i, S_j) if $R_i.r < S_j.l$, and we can derive similar conditions for $<$. In general, based on the uncertainty operator and uncertainty intervals, we may derive pruning conditions and choose an efficient I/O join algorithm to facilitate pruning.

5.3.1 Item-Level Pruning

The set C of candidate pairs (R_i, S_j), produced in Step 2, is further refined in Step 3. The refinement process can be done by directly computing the join probability, $P(R_i \theta_u S_j)$ for every pair of (R_i, S_j); only those larger than p are retained. The exact way of computing the this probability depends on the type of uncertainty pdf. For

```
**Input**
    $R, S$ /* tables containing common uncertainty attributes */
    $\theta_u$ /* uncertainty join operator */
    $p$ /* probability threshold of PTJQ */

**Output**
    $(R_i, S_j)$ that satisfies $P(R_i \theta_u S_j) > p$

**Begin**
    1. Let $A \leftarrow \phi$ /* $A$ is the answer of PTJQ */
    2. Let $C \leftarrow \{(R_i, S_j)|$ where $(R_i, S_j)$ are results returned by an interval join
       algorithm over $R_i.U$ and $S_j.U$ $\}$ ( For $=_c$ and $\neq_c$, join over
       $[R_i.l{-}c, R_i.r{+}c]$, $[S_j.l{-}c, S_j.r{+}c]$)
    3. $\forall (R_i, S_j)$ in $C$
           i. if $P(R_i \theta_u S_j) > p$ **then** $A \leftarrow A \bigcup (R_i, S_j)$
**End**
```

Figure 5.3. Evaluating a PTJQ with an interval join.

uniform pdf, a closed-form formula can be derived. For Gaussian distribution, the join probability may be implemented by a table lookup. For an arbitrary pdf, $P(R_i \theta_u S_j)$ may not be in closed-form; the join probability can be computed with (relatively expensive) numerical integration methods.

We develop a set of techniques to facilitate the evaluation of Step 3. These methods do not compute $P(R_i \theta_u S_j)$ directly. Instead, they establish *pruning conditions* that can be checked easily to decide whether (R_i, S_j) satisfy the query. They are applicable to any kind of uncertainty pdf, and do not require the knowledge of the specific form of $P(R_i \theta_u S_j)$. They are thus convenient for developing an uncertain database system that supports a wide range of uncertainty pdfs. Moreover, they form the basis of discussions of other pruning techniques in later sections. We term these techniques "item-level-pruning", since pruning is performed based on testing a pair of data items. Let us now discuss the pruning criteria for each operator.

For **Equality** and **Inequality**, we establish the following lemma:

Lemma 1 *Suppose a and b are uncertain-valued variables and $a.U \cap b.U \neq \phi$. Let $l_{a,b,c}$ be $\max(a.l - c, b.l - c)$ and $u_{a,b,c}$ be $\min(a.r + c, b.r + c)$. Then,*

- *$P(a =_c b)$ is at most*

$$\min(a.F(u_{a,b,c}) - a.F(l_{a,b,c}), b.F(u_{a,b,c}) - b.F(l_{a,b,c})) \tag{5.2}$$

- *$P(a \neq_c b)$ is at least*

$$1 - \min(a.F(u_{a,b,c}) - a.F(l_{a,b,c}), b.F(u_{a,b,c}) - b.F(l_{a,b,c})) \tag{5.3}$$

Proof Since a and b overlap at interval $[l_{a,b,c}, u_{a,b,c}]$, from Equation 5.1 we have

$$\begin{aligned} P(a =_c b) &= \int_{l_{a,b,c}}^{u_{a,b,c}} a.f(x)(b.F(x+c) - b.F(x-c))dx \\ &\leq \int_{l_{a,b,c}}^{u_{a,b,c}} a.f(x)dx \\ &= a.F(u_{a,b,c}) - a.F(l_{a,b,c}) \end{aligned}$$

Similarly, we have $P(b =_c a) \leq b.F(u_{a,b,c}) - b.F(l_{a,b,c})$. Since $P(a =_c b)$ is equal to $P(b =_c a)$, $P(a =_c b)$ cannot be larger than the minimum of $a.F(u_{a,b,c}) - a.F(l_{a,b,c})$ and $b.F(u_{a,b,c}) - b.F(l_{a,b,c})$. Thus the lemma holds. The bound on the inequality probability can be similarly established. □

Lemma 1 enables us to quickly decide whether a candidate pair $(R_i, S_j) \in C$ should be included into or excluded from the answer, since uncertainty cdfs are known and Equations 5.2 and 5.3 can be computed easily. For equality, the lemma allows us to prune away (R_i, S_j) when Equation 5.2 is less than p; for inequality, we can immediately claim that (R_i, S_j) is the answer when Equation 5.3 is larger than p.

For **Greater than** and **Less than**, we have the following Lemma 2.

Lemma 2 *Suppose a and b are uncertain-valued variables. Then,*

- *For $a > b$,*

  1. *If $a.l \leq b.r < a.r$, $P(a > b) \geq 1 - a.F(b.r)$.*
  2. *If $a.l \leq b.l \leq a.r$, $P(a > b) \leq 1 - a.F(b.l)$.*

- *For $a < b$,*

  1. *If $a.l \leq b.l \leq a.r$, $P(a < b) \geq a.F(b.l)$.*
  2. *If $b.l < a.l \leq b.r$, $P(a < b) \leq a.F(b.r)$.*

Proof Lemma 2.1 is a direct result from Definition 5.2.3. For Lemma 2.2, when $a.l \leq b.l \leq a.r$, $P(a < b)$ is equal to $a.F(b.l) + \int_{b.l}^{\min(a.r,b.r)} a.f(x)(1 - b.F(x))dx$ (Definition 5.2.4) , which is larger than or equal to $a.F(b.l)$. Since $P(a > b) = 1 - P(a < b)$, $P(a > b)$ must be smaller than or equal to $1 - a.F(b.l)$. □

To understand how this lemma facilitates pruning for $>$, notice that we can immediately include (R_i, S_j) in the answer if $R_i.l \leq S_j.r < R_i.r$ and $1 - R_i.F(S_j.r) \geq p$, since by the first rule of the lemma $P(R_i > S_j)$ has to be larger than p. Observe that (R_i, S_j) can also be included in the answer if $R_i.l > S_j.r$. On the other hand, the second rule of the lemma allows (R_i, S_j) to be excluded from the answer, if the right side expression of $P(a > b)$ has probability value less than p. Notice that (R_i, S_j) can also be excluded from the answer if $R_i.r < S_j.l$. The rules for $<$ in Lemma 2 can be used for pruning in a similar manner.

Given that the pdfs of the uncertain values are known, the above lemmata allow us to perform a constant-time check to decide whether $P(R_i \theta_u S_j)$ has to be evaluated. Thus, for the price of a small overhead, we may be able to avoid the evaluation of actual probabilities in Step 3, which can be expensive. From now on, we assume that checks based on the above lemmata are performed to process the predicate $P(R_i \theta_u S_j)$ in Step 3. In Section 5.5, we experimentally examine the effectiveness of the framework presented in Figure 5.3, where we study two common interval join algorithms: block nested loop join (BNLJ) and indexed nested loop join (INLJ).

Notice that the interval-join operation, performed in Step 2, can generate a lot of candidate pairs that are actually not part of the answer (i.e., their probabilities are less than p) The key problem with Step 2 is that it uses uncertainty intervals as the only pruning criterion. In the next section, we examine algorithms that use *both* uncertainty intervals and uncertainty pdfs for pruning, so that a smaller candidate set is produced. In some of these methods, the I/O performance is improved too.

This interval join not only affects the performance of Step 3, but itself can also generate unnecessary I/Os. In fact, the key problem with the algorithm of Figure 5.3 is that it uses uncertainty intervals as the only pruning criterion. In the next section, we examine join algorithms which use *both* uncertainty intervals and uncertainty pdfs for pruning, improving I/O performance significantly. It also generates a smaller candidate set for Step 3 and thus computation time is saved too.

5.4 Uncertainty-based Joins

The performance of Step 2 in Figure 5.3 is essential to the overall performance since it eliminates some I/O operations. of the join because it performs I/O-level pruning. As explained above,, interval joins may not be the best solution because they do not utilize uncertainty pdfs. We now present join algorithms that are tailored for uncertainty. We discuss how to prune at the page level for different uncertainty operators, and how this page-level pruning can be realized in join algorithms.

The discussion focuses on the **equality** ($=_c$) and **greater than** ($>$) operators. The other operators are similar to these and are thus not discussed in detail.

5.4.1 The Uncertainty Bounds

For database joins like the block-nested-loop join and the indexed-loop-join, the unit of retrieval is a page. Suppose we are given two pages, one from R and the other from S. To perform a join between the uncertain values contained in these two pages, a simple approach is to consider all pairs of values in the two pages. This can be time-consuming, because a page of a modest size can contain many uncertain values[1]. Our goal is "page-level" pruning: with an additional small storage overhead, it can avoid examining the page contents.

The idea of using a small overhead to facilitate the pruning of uncertain values was first proposed in [30] to answer probabilistic threshold range queries – essentially a range query where only uncertain data items that satisfy it with a probability higher than a user-defined threshold are reported. The main idea is to augment some tighter bounds (x-bound) in each node in an interval R-tree. Each x-bound is a pair of bounds that are calculated based on the properties of the uncertainty pdfs associated with the entries stored in that node. Since an x-bound is potentially tighter than the Minimum Bounding Rectangle (MBR), the pruning power can be increased. In this chapter, we borrow the idea of x-bound to facilitate page-level joins. Based on the definition of x-bounds for a tree node in [30], we generalize the definition of x-bound for a page:

Definition 5.4.1 *Given $0 \leq x < 1$, an* **x-bound** *of a page B consists of two values, called left-x-bound ($B.l(x)$), and right-x-bound ($B.r(x)$). For every uncertain value a stored in B, two conditions must hold:*

- *If $a.l < B.l(x)$, then $\int_{a.l}^{B.l(x)} a.f(y)dy \leq x$.*
- *If $a.r > B.r(x)$, then $\int_{B.r(x)}^{a.r} a.f(y)dy \leq x$.*

[1]For example, if an uncertain attribute uses 8 bytes to store its uncertainty interval, 8 bytes to specify the uniform uncertainty pdf and cdf, a $4K$ page can store 256 items. Joining values in two pages then requires examining $256^2 = 65536$ pairs.

Essentially, we require that every uncertain attribute stored in a page must have no more than a probability of x of being outside either the left-x-bound or the right-x-bound. We also assume that x-bounds are "tight", i.e., the left-x-bounds (right-x-bounds) are pushed to the right (left) as much as possible. To illustrate, Figure 5.4 shows a page storing two uncertain attributes, a and b. As we can see, a has a probability less than 0.1 and 0.3 of lying to the left of the left-0.1-bound and left-0.3-bound respectively, i.e., $\int_{a.l}^{B.l(0.1)} a.f(y)dy \leq 0.1$ and $\int_{a.l}^{B.l(0.3)} a.f(y)dy \leq x$. Similarly, a cannot have a probability of over 0.3 of being outside the right-0.3-bound. Similarly, b has a probability of at least 0.7 on the right of the left-0.3-bound. Finally, all the uncertainty intervals must be fully enclosed by the 0-bound, which is akin to the MBR of an index node.

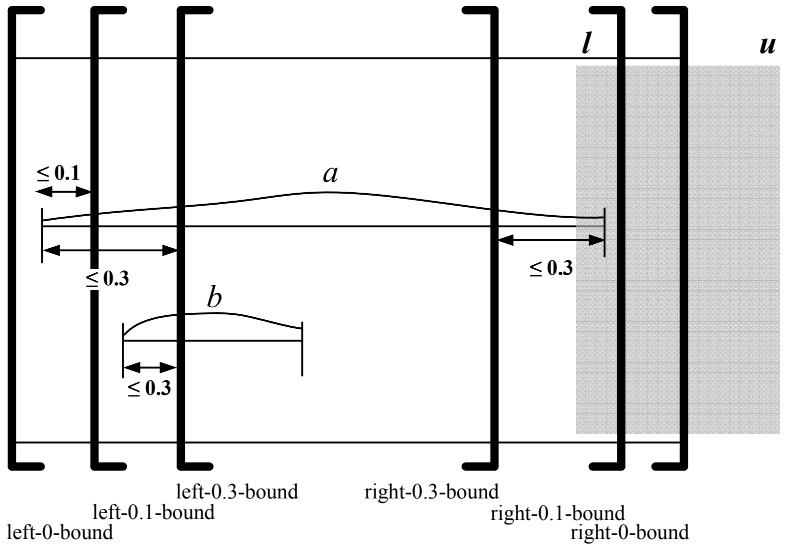

Figure 5.4. 0-bound, 0.1-bound and 0.3-bound. A range query $[l, u]$ with $p = 0.4$ is also shown.

The major purpose of the x-bound is to facilitate pruning for probabilistic threshold range queries. Suppose a range query has a lower bound l, upper bound u and probability threshold p. As shown in Figure 5.4, if p is larger than 0.4, we are immediately guaranteed that none of the uncertain attributes can satisfy the query:

each attribute has a probability of less than 0.3 of being located inside $[l, u]$. We will explain how x-bounds are used to prune in order to process joins effectively.

The implementation of uncertain items and x-bounds in a page is shown in Figure 5.5. For pdf and cdf, we store the symbol of the type of the distribution, and the parameters relevant to that distribution. For example, if the pdf is Gaussian, then the pdf can be a pair of values (mean, variance), and the cdf may be approximated by a histogram. To implement the x-bounds, we store a table V on the same page, where V_i is a tuple of the form (l, r) for storing the left-W_i-bound and right-W_i-bound. The values of W_i's ($i = 1, \ldots, |W_i|$) are stored in an external table W, sorted in ascending order of W_i's. Our join algorithms require 0-bounds to be stored, with W_1 equal to 0, and $[V_1.l, V_1.r]$ representing the position of the 0-bound. Figure 5.5 shows the implementation of x-bounds for the example in Figure 5.4. The total space cost of V and W is $O(|W|)$, which is usually small since only a few x-bounds are stored.

To insert an item to the page, we first compute the x-bound of the item. This is usually an inexpensive one-time cost. If the uncertainty pdf is a standard distribution (e.g., uniform), the x-bounds are readily obtained. For an arbitrary pdf (e.g., represented by a histogram), its x-bounds can be derived by scanning the histogram once. The x-bound of the page is then expanded to accommodate the new item.

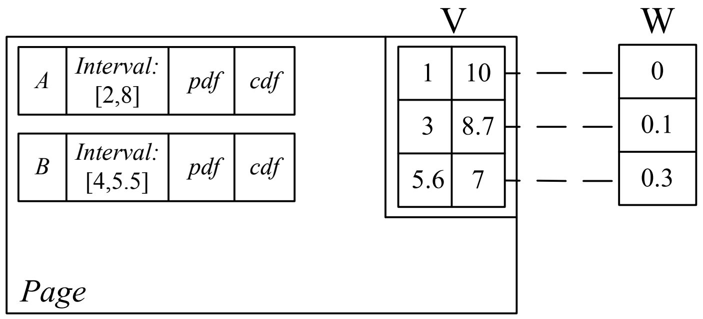

Figure 5.5. Implementing x-bounds in a page.

Given a page B with uncertainty tables, we now present two algorithms (Figure 5.6) to decide if any uncertain attributes have a probability higher than p of satisfying a range query. Algorithm `CheckLeft` checks the range query against left-x-bounds while Algorithm `CheckRight` employs right-x-bounds for checking. They use the idea illustrated in Figure 5.4 for pruning, and we state without proof the following lemma.

Lemma 3 *Given a range query Q with interval $[l, u]$ and probability threshold p, if* `CheckLeft` *or* `CheckRight` *returns* `FALSE`*, no uncertain attribute in B can satisfy Q with probability higher than p.*

These two checking routines form the fundamental building blocks for the page-level join operators. They are usually very efficient since only a few x-bounds need to be stored and W is small.

5.4.2 Page-Level Equality Join

Using `CheckLeft` and `CheckRight`, a page-level equality join can be constructed as follows. Figure 5.7 illustrates **EquiJoin**, which returns `PRUNE` to indicate that two given pages from R and S do not contain any join pairs with probability over p of being equal, in which case the two pages can be pruned without further investigation. **EquiJoin** returns `CHECK` to indicate that there is a possibility that some pairs satisfying the conditions exist which results in a pairwise evaluation of the values in the pages R and S.

EquiJoin applies two sets of criteria. The first test (Step 1) uses `CheckLeft` and `CheckRight` on page B_S (of table S), using the 0-bound of page B_R (extended with resolution c) to form a range query. In other words, the range query with the interval $[B_R.V_1.l - c, B_R.V_1.r + c]$ is checked against B_S using left- and right-x-bounds. If `CheckLeft` or `CheckRight` returns `FALSE`, by Lemma 3 no uncertain attribute in B_S is in $[B_R.V_1.l - c, B_R.V_1.r + c]$ with a probability higher than p. **EquiJoin** then returns `PRUNE` to indicate that these pages cannot be joined.

If Step 1 does not return `PRUNE`, **EquiJoin** uses another set of tests in Step 2, which exchanges the role of B_R and B_S: the range query is now constructed by

```
Input
    $[l, u]$ /* Lower and upper bound of range query $Q$ */
    $p$ /* probability threshold of range query */
    $B$ /* Page with table $B.V$ */
    $W$ /* Global table storing values of $x$ for $x$-bounds */
Output
    FALSE: All intervals in $B$ are guaranteed to fail $Q$,
    TRUE otherwise.
(a) CheckLeft($l, u, p, B, W$) /* prune using left-$x$-bounds */
    1. for $i = 1, \ldots, |W|$ do
        (i) if $u < B.V_i.l$ and $W_i < p$ then
            (a) return FALSE
    2. return TRUE
(b) CheckRight($l, u, p, B, W$) /* prune using right-$x$-bounds */
    1. for $i = 1, \ldots, |W|$ do
        (i) if $l > B.V_i.r$ and $W_i < p$ then
            (a) return FALSE
    2. return TRUE
```

Figure 5.6. Algorithms for deciding whether a page B can be pruned for a range query. (a) `CheckLeft` uses left-x-bounds for pruning. (b) `CheckRight` uses right-x-bounds for pruning.

using the 0-bound of B_S, and tested against the uncertainty bounds in B_R. Again, **EquiJoin** returns `PRUNE` if either `CheckLeft` or `CheckRight` is `FALSE`. If none of these tests work, **EquiJoin** concludes that it cannot prune the pages (Step 3).

```
Input
  $B_R$ /* Page (with uncertainty bounds) from table $R$ */
  $B_S$ /* Page (with uncertainty bounds) from table $S$ */
  $W$ /* Global table storing values of $x$ for $x$-bounds */
  $c$ /* Resolution of equality */
  $p$ /* probability threshold of equality join */
Output
  (i) PRUNE: for every $R_i$ in $B_R$, $S_j$ in $B_S$, ,it is certain that $P(R_i =_c S_j) < p$,
  (ii) CHECK otherwise.

EquiJoin($B_R, B_S, W, c, p$)
  1. if (NOT(CheckLeft($B_R.V_1.l - c, B_R.V_1.r + c, p, B_S, W$))) or
       (NOT(CheckRight($B_R.V_1.l - c, B_R.V_1.r + c, p, B_S, W$)))
       then return PRUNE
  2. if (NOT(CheckLeft($B_S.V_1.l - c, B_S.V_1.r + c, p, B_R, W$))) or
       (NOT(CheckRight($B_S.V_1.l - c, B_S.V_1.r + c, p, B_R, W$)))
       then return PRUNE
  3. return CHECK
```

Figure 5.7. Page Level Join for Equality.

The correctness of **EquiJoin** hinges on the four test conditions. In the rest of this section, we establish the correctness when the first testing procedure in Step 1, namely `CheckLeft`, returns `FALSE` on pages B_R and B_S. The other three conditions use the same principles and their proofs are skipped. We begin with the following lemma.

Lemma 4 *1. If* `CheckLeft` *of Step 1 in* **EquiJoin** *returns* `FALSE`*, then for every uncertain value S_j in B_S, its probability of satisfying the range query formed by any uncertainty interval of R_i stored in B_R extended with c, i.e., $[R_i.l-c, R_i.u+c]$, must be less than p.*

2. If `CheckRight`*$(B_S.V_1.l-c, B_S.V_1.r+c, p, B_R, W)$) returns* `FALSE`*, then for every uncertain value in B_R, its probability of satisfying the range query formed by any uncertainty interval $[S_j.l-c, S_j.u]$ (where S_j is an uncertain value in B_S) must be less than p.*

Proof From Lemma 3, we know that no attributes in B_S satisfies the range query formed by $[B_R.V_1.l-c, B_R.V_1.r+c]$ with probability higher than p. Further, any uncertainty interval $R_i.U$ in B_R must be enclosed by $[B_R.V_1.l, B_R.V_1.r]$, and therefore $R_i.r+c \leq B_R.V_1.r+c$. According to Step 1(i) of `CheckLeft` there must be some q such that $B_R.V_1.r+c < B_S.V_q.l$ and $W_q < p$. Therefore,

$$R_i.r + c < B_S.V_q.l \tag{5.4}$$

As shown in Figure 5.8, none of the uncertainty intervals in B_S crosses the line $B_S.V_q.l$ with a fraction of more than W_q. This implies no values in B_S can satisfy $[R_i.l-c, R_i.r+c]$ with probability higher than p. Symmetric arguments can be made for `CheckRight`.

□

For any R_i and S_j stored in pages B_R and B_S, the intersection between $[R_i.l-c, R_i.r+c]$ and $[S_j.l-c, S_j.r+c]$ is given by $[l_{R_i,S_j,c}, u_{R_i,S_j,c}]$, where $l_{R_i,S_j,c}$ is $\max(R_i.l-c, S_j.l-c)$ and $u_{R_i,S_j,c}$ is $\min(R_i.r+c, S_j.r+c)$. The following lemma can be derived.

Lemma 5 *If* `CheckLeft` *of Step 1 in* **EquiJoin** *returns* `FALSE`*, then*

$$S_j.F(u_{R_i,S_j,c}) - S_j.F(l_{R_i,S_j,c}) < p \tag{5.5}$$

Proof Recall from Lemma 4 that S_j with uncertainty interval $[S_j.l, S_j.r]$ satisfies range query $[R_i.l-c, R_i.r+c]$ with a probability less than p. This implies the

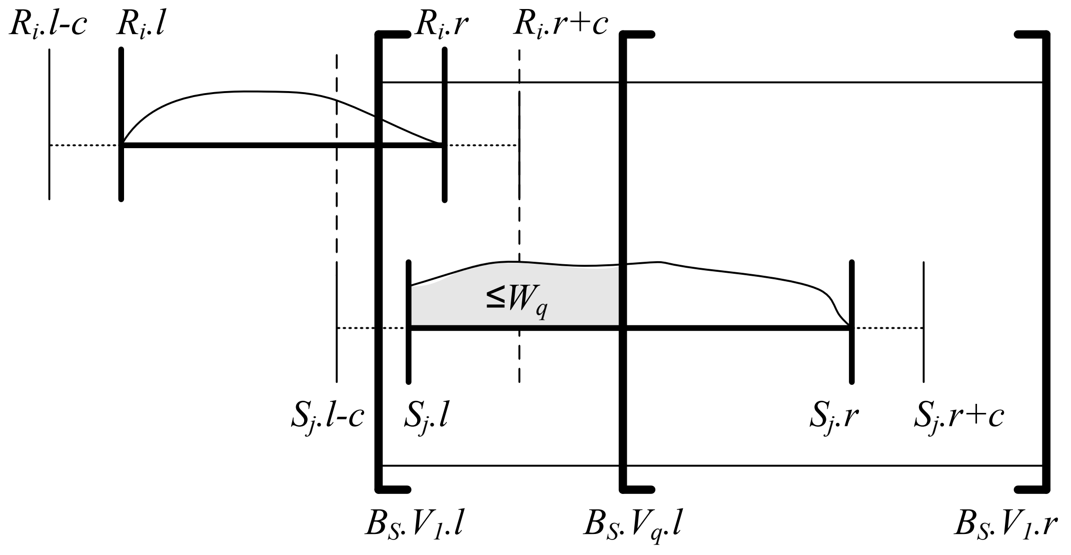

Figure 5.8. Illustrating the correctness of **EquiJoin**.

cumulative probability in the overlap region of $S_j.U$ and $[R_i.l-c, R_i.r+c]$ is less than p, i.e.,

$$S_j.F(\min(R_i.r+c, S_j.r)) - S_j.F(\max(R_i.l-c, S_j.l)) < p \tag{5.6}$$

We now make the following claims.

Claim 1:

$$S_j.F(\max(R_i.l-c, S_j.l)) = S_j.F(l_{R_i,S_j,c}) \tag{5.7}$$

Proof There are two cases:

1. $R_i.l - c \geq S_j.l$. Then $R_i.l - c \geq S_j.l - c$, and hence $\max(R_i.l-c, S_j.l)$ is equal to $\max(R_i.l-c, S_j.l-c)$, and thus Equation 5.7 is correct.

2. $R_i.l - c < S_j.l$. Then $S_j.F(\max(R_i.l-c, S_j.l)) = S_j.F(S_j.l) = 0$. Moreover, $\max(R_i.l-c, S_j.l-c)$ is either $R_i.l-c$ or $S_j.l-c$; the latter is illustrated in Figure 5.8. Since $R_i.l-c$ and $S_j.l-c$ are less than $S_j.l$, by Definition 3.1.2, both $S_j.F(R_i.l-c)$ and $S_j.F(S_j.l-c)$ are equal to 0. Therefore, Equation 5.7 is correct.

□

Claim 2:

$$S_j.F(\min(R_i.r + c, S_j.r)) = S_j.F(u_{R_i,S_j,c}) \tag{5.8}$$

Proof Recall from Equation 5.4 that $R_i.r+c$ must be to the left of the left-W_q-bound, as illustrated in Figure 5.8. Moreover, as $W_q < 1$, $S_j.r$ must be to the right of $B_S.V_q.l$; otherwise the entire interval $S_j.U$ is on the left of the left-W_q-bound, implying that $\int_{S_j.l}^{B_S.V_q.l} S_j.f(y)dy$ is 1, which is larger than W_q and violates Definition 5.4.1. Hence, $R_i.r + c$ is less than $S_j.r$, which in turn cannot be larger than $S_j.r + c$. This means $\min(R_i.r + c, S_j.r)$ is the same as $\min(R_i.r + c, S_j.r + c)$, and thus Equation 5.8 is correct.

□

Based on Equations 5.7 and 5.8, the left hand side of Equation 5.6 is the same as

$$S_j.F(\min(R_i.r + c, S_j.r + c)) - S_j.F(\max(R_i.l - c, S_j.l - c))$$

Thus Lemma 5 holds.

□

We now prove the correctness of **EquiJoin**. Suppose Step 1's `CheckLeft` returns `FALSE`. From Lemma 1, we know that $P(S_j =_c R_i) \leq S_j.F(u_{R_i,S_j,c}) - S_j.F(l_{R_i,S_j,c})$, which is less than p according to Lemma 5. Thus Step 1's `CheckLeft` prunes pages correctly.

For the remaining criteria, the proofs are skipped due to lack of space. By calling four small testing routines, **EquiJoin** can efficiently identify pruning opportunities by using x-bounds of the pages.

5.4.3 Page-Level Join for "Greater than"

We have developed a page-level pruning algorithm for ">" called **GTJoin**. As illustrated in Figure 5.9, **GTJoin** returns three possible answers. The first type of answer, called `PRUNE`, signals to the caller of **GTJoin** that no interval pairs in the pages concerned have a probability of p or more of being joined (Step 1). The second type of answer, called `INCLUDE`, does the opposite: it informs the user that *every* pair of intervals from B_R and B_S join with probability higher than p, and these pairs can be inserted to the answer without hesitation (Step 2). The final kind of answer, `CHECK`, is returned when neither the conditions in Step 1 nor those in Step 2 are satisfied. This implies that all pairs must be checked for possible inclusion in the result.

Intuitively, Step 1 first forms a range query by using the 0-bounds of B_S and query it against the right x-bounds of page B_R, by using `CheckRight`. Figure 5.10(a) illustrates this. If there exists some q such that $B_S.V_1.l \geq B_R.V_q.r$ and $W_q < p$, the page pairs can be pruned. If this test fails to prune, another test based on `CheckLeft` is performed, where the range query is formed by the 0-bounds of B_R, querying against the left x-bounds of B_S. The scenario is shown in Figure 5.10(b).

The function of `CheckRight` and `CheckLeft` of Step 1 is to test whether $P(R_i > S_j) < p$, and if so, "throw away" B_R and B_S. Step 2 performs the opposite: it establishes the conditions in which every pair of items in B_R and B_S can be placed in the answer. Specifically Step 2 verifies the condition $P(S_j > R_i) < 1 - p$, which can be easily achieved by modifying the parameters in Step 1. Since $P(R_i > S_j) = 1 - P(S_j > R_i)$, if any of the two conditions in Step 2 are satisfied, we can conclude that $P(R_i > S_j) \geq p$. **GTJoin** then returns `INCLUDE` to indicate that all combinations of (R_i, S_j) can be inserted to the answer without probing.

Similar to **EquiJoin**, **GTJoin** requires little time as it only calls four small checking subroutines. With this little overhead, the savings can be significant as illustrated in our experiments.

```
Input
    $B_R$ /* Page (with uncertainty bounds) from table $R$ */
    $B_S$ /* Page (with uncertainty bounds) from table $S$ */
    $W$ /* Global table storing values of $x$ for $x$-bounds */
    $p$ /* probability threshold of $>$ join */
Output
    (i) PRUNE: $\forall R_i \in B_R, S_j \in B_S$, it is
            certain that $P(R_i > S_j) < p$;
    (ii) INCLUDE: $\forall R_i \in B_R, S_j \in B_S$, it is
            certain that $P(R_i > S_j) \geq p$;
    (iii) CHECK otherwise.
GTJoin($B_R, B_S, W, p$)
    1. if NOT(CheckRight($B_S.V_1.l, B_S.V_1.r, p, B_R, W$))
       or NOT(CheckLeft($B_R.V_1.l, B_R.V_1.r, p, B_S, W$))
       then return PRUNE
    2. if NOT(CheckRight($B_R.V_1.l, B_R.V_1.r, 1-p, B_S, W$))
       or NOT(CheckLeft($B_S.V_1.l, B_S.V_1.r, 1-p, B_R, W$))
       then return INCLUDE
    3. return CHECK
```

Figure 5.9. Page Level Join for $R_i > S_j$.

5.4.4 Uncertainty-Enhanced Joins

The page-level pruning techniques can be used to improve the performance of interval or spatial join algorithms that retrieve data in units of pages. Whenever two data pages are compared in the join algorithms, uncertainty tables can be read

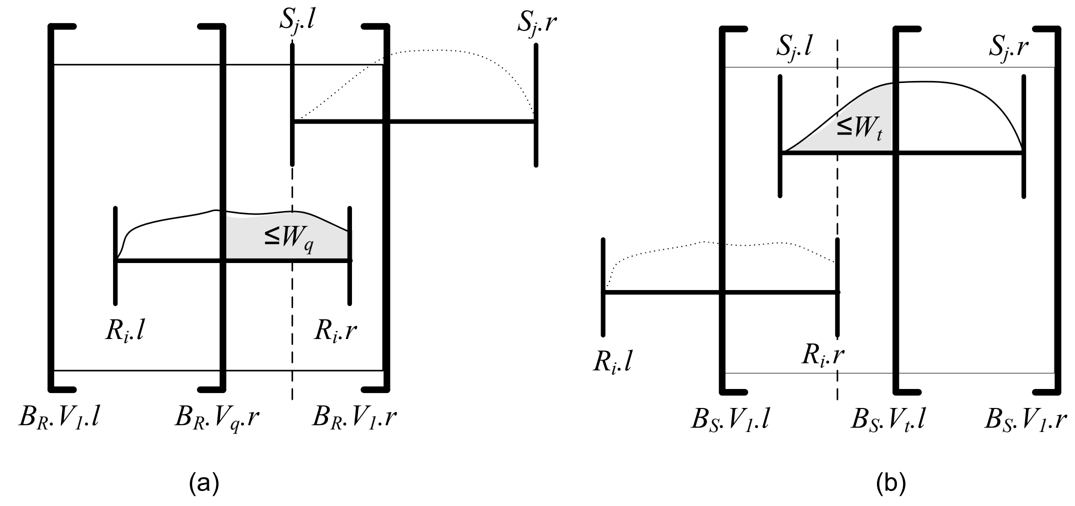

Figure 5.10. Pruning pages for $>$, using (a) right-x-bounds of B_R, and (b) left-x-bounds of B_S.

first, and with our pruning techniques, probing into actual values in the pages can be avoided. Of course, **GTJoin** may not prevent the retrieval of intervals when `INCLUDE` is returned – however, it still improves performance because we can simply add the Cartesian product of the intervals from the two pages to the answer without computing the actual probabilities.

We further illustrate our techniques by studying the example of the Block-Nested-Loop Join (**BNLJ**). In this algorithm, the two relations to be joined are organized as lists of unordered pages. Each page read from the outer relation is matched with each page from the inner relation iteratively, which can be slow because we have to check each pair of intervals from both relations. However, by augmenting each page with an uncertainty table, we can speed up this matching process by using **EquiJoin** or **GTJoin**. We denote the version of **BNLJ** where uncertainty tables are augmented as **Uncertainty-based Block-Nested-Loop Join** (**U-BNLJ** for short). We will compare the performance differences experimentally between these two join algorithms in Section 5.5. Other page-based join algorithms, such as interval hash join and sort-merge-join, can be enhanced in a similar manner and the details are skipped here.

5.4.5 Index-Level Join

Although uncertainty tables can be used to improve the performance of page-based joins, they do not improve I/O performance, simply because the pages still have to be loaded in order to read the uncertainty tables. However, we can extend the idea of page-level pruning to improve I/O performance, by organizing the pages in a tree structure. Conceptually, each tree node still has an uncertainty table, but now each uncertainty interval in a tree node becomes a Minimum Bounding Rectangle (MBR) that encloses all the uncertainty intervals stored in that MBR. Page-level pruning now operates on MBRs instead of uncertainty intervals. The correctness of these algorithms can be shown easily, by using the fact that each MBR tightly encloses the intervals within the subtree, and arguments similar to Lemma 4.

An implementation of uncertainty tables in the index level is the the Probability Threshold Index (PTI) [30], originally designed to answer probability threshold range queries. It is essentially an interval R-Tree, where each intermediate node is augmented with uncertainty tables. Specifically, for each child branch in a node, PTI stores *both* the MBR and the uncertainty table V of each child. We can use PTI to improve join performance in the framework of the Indexed-Nested-Loop-Join (**INLJ**), by constructing a PTI for the inner relation. The 0-bound of each page from the outer relation is then treated as a range query and tested against the PTI in the inner relation. All pages that are retrieved from the PTI are then individually compared with the page from where the range query is constructed, and our page-level pruning techniques can then be used again to reduce computation efforts.

We denote the version of **INLJ** where PTI is used in place of an interval index as **Uncertainty-based Indexed-Loop Join**, or **U-INLJ** for short. We present the performance results of **INLJ** and **U-INLJ** in the next section.

5.5 Experimental results

We have evaluated the performance of our pruning methods for the equality operator. We will present the simulation model followed by the results.

5.5.1 Simulation Model

Two tables of uncertain data are generated, where the uncertainty pdf is uniform for both datasets. For the first table, uncertainty intervals are uniformly distributed in $[0, 10000]$. The length of each interval is normally distributed with a mean μ of 5 and deviation σ of 1. For the other table, intervals are uniformly distributed in $[5000, 15000]$, and the length is normal with $\mu = 10$ and $\sigma = 2$. Each disk page stores up to 50 tuples. We study the performance of joins over these two tables by evaluating the number of tuple-pair candidates output from the join algorithms (N_{pair}) for item-level pruning, and the number of pairs where probability evaluation has to be performed (N_{prob}). Notice that each "probability evaluation" can be expensive because of the costly integration operation involved in finding the probability – which is done when pruning techniques fail. Ideally N_{prob} should be small.

5.5.2 Page-Level Pruning

Figure 5.11 shows that **U-BNLJ** performs substantially better than **BNLJ** in N_{pair}. This is because **U-BNLJ** performs page-level pruning while **BNLJ** does not. However, **U-BNLJ** does not benefit much from large values of p. Since intervals are stored randomly, intervals in each disk page can be widely spread. Consequently all the x-bounds are close to the 0-bound, and the page-level join cannot exploit p effectively.

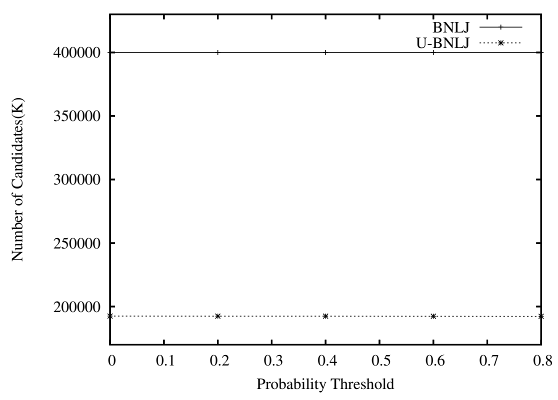

Figure 5.11. **BNLJ** and **U-BNLJ**.

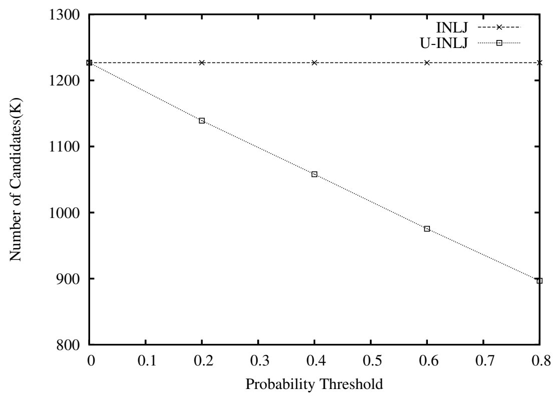

Figure 5.12. **INLJ** and **U-INLJ**.

5.5.3 Index-Level Pruning

The above problem can be alleviated by organizing intervals in a better way, for example, with an index. Figure 5.12 shows that both **INLJ** and **U-INLJ** address a much better performance in N_{pair} than **BNLJ** and **U-BNLJ**. Further, **U-INLJ** exploits p much better than **INLJ** as uncertainty bounds are used effectively.

5.5.4 Item-Level Pruning

Figure 5.13 shows the number of pairs that we have to compute probability (N_{prob}) for the four joins. We see that the four graphs almost coincide. This means regardless of how many tuple-pairs are produced, the final number of intervals that have to be evaluated is almost the same. This implies our item-level pruning techniques can eliminate a large portion of false positives regardless of the join algorithm. The computational effort due to probability evaluation is reduced significantly.

The effect of **Resolution** for the equality operator is illustrated in Figure 5.14. We observe that N_{prob} increases with c. With a larger value of c, the uncertainty interval of each tuple is expanded significantly and thus the chance for pruning is reduced. However, increase in c implies more relaxation of "equality", potentially

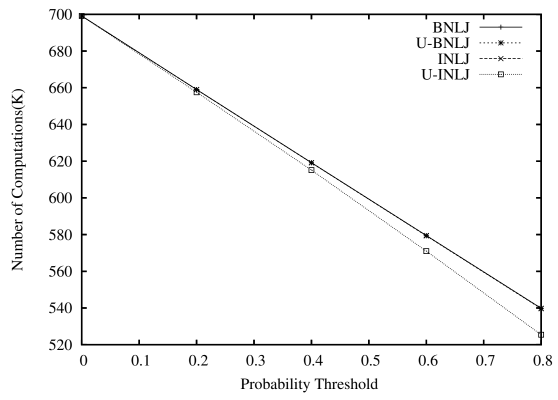

Figure 5.13. N_{prob} vs p.

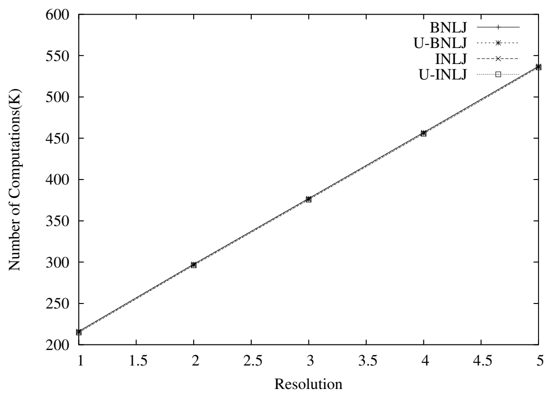

Figure 5.14. N_{prob} vs c.

returns more answers. This is illustrated in Figure 5.15. Interestingly, the growth of number of answers saturates as $c > 3$. This indicates that c does not need to be large in order to obtain all possible matches.

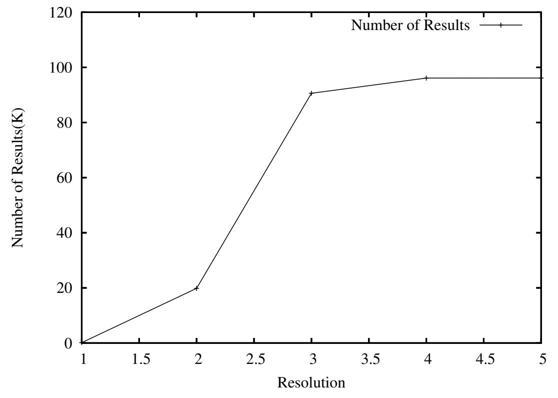

Figure 5.15. No. of results vs c.

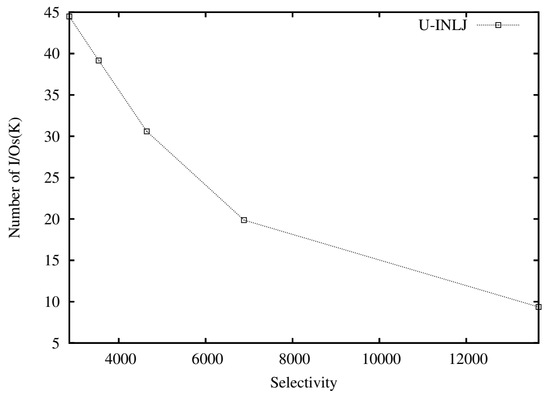

Figure 5.16. Selectivity on **U-INLJ**.

5.5.5 Selectivity

We also test the effect of join selectivity on **U-INLJ**. Figure 5.16 shows that **U-INLJ** benefits from high selectivity. When a join is highly selective, **U-INLJ** requires less traversal over the tree, and thus fewer pages need to be retrieved.

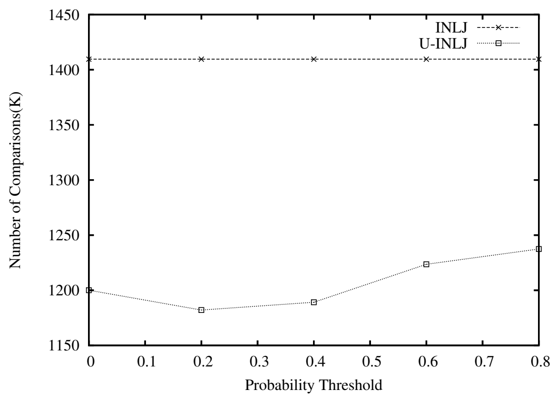

Figure 5.17. **INLJ** and **U-INLJ** (for >).

5.5.6 Greater Than

We present an interesting result for > in Figure 5.17. We observe that **U-INLJ** does not have the same behavior as in Figure 5.12. Here N_{pair} does not show a sharp drop as p increases. Recall that in the page-level join for >, `INCLUDE` may be returned. When p is very low, there is a high chance for objects to be directly included in the answer. Hence N_{pair} is low when p is low.

5.6 Chapter Summary

We identified an important issue in managing data imprecision: the extension of comparison operators for uncertainty and the joining of uncertain-valued attributes. Joining uncertainty can be costly, and we discussed numerous techniques to reduce the cost. We illustrate how pruning can be achieved at different granularity: item

Table 5.1
Pruning Methods for Uncertainty Joins.

Level	**Savings**	**Applicability**	**Algorithms**
Item	Computation	$=_c, \neq_c, >, <$	BNLJ, INLJ
Page	Computation	$=_c, >, <$	U-BNLJ
Index	I/O & computation	$=_c, >, <$	U-INLJ

level, page level, and index level. Their properties are summarized in Table 5.1. With only a small overhead, these techniques can improve join performance significantly.

The results of this chapter are presented in [47]. In the next chapter, we shift our focus to indexing techniques for uncertain categorical data.

6 INDEXING UNCERTAIN CATEGORICAL DATA

Uncertainty in categorical data is commonplace in many applications, including data cleaning, database integration, and biological annotation. In such domains, the correct value of an attribute is often unknown, but may be selected from a reasonable number of alternatives. In this chapter we extend traditional systems to explicitly handle categorical uncertainty in data values. We propose two index structures for efficiently searching uncertain categorical data, one based on the R-tree and another based on an inverted index structure. Using these structures, we provide a detailed description of the probabilistic equality queries they support. Experimental results using real and synthetic datasets demonstrate how these index structures can effectively improve the performance of queries through the use of internal probabilistic information.

6.1 Introduction

As discussed in previous chapters, there are many applications for which the data exhibits uncertainty in attribute values. As with traditional data, these is a need for efficient execution of queries over uncertain data. Existing database index structures that are developed for precise data are not directly applicable for uncertain data. For uncertain data, indexing support has only been developed for real-valued attributes [30]. These index structures are inapplicable for categorical uncertain data.

This chapter addresses the problem of indexing uncertain categorical data represented as a set of values with associated probabilities. We propose two different index structures. We show that these structures support a broad range of probabilistic queries over uncertain data, including the typical equality, probability threshold, and top-K queries. Our index structures can also be used for queries that are only meaningful for uncertain data such as distribution similarity queries. The new indexes are shown to provide efficient execution of these queries with good scalability through experimental validation using real and synthetic data. To sum up, the contributions of this chapter are:

- The development of two index structures for uncertain categorical data; and
- The experimental evaluation of these structures with real and synthetic data.

The rest of this chapter is organized as follows. Section 6.2 describes the model for uncertain data and definitions for queries over this data. Section 6.3 presents the new index structures and experimental results are discussed in Section 6.4. Section 6.5 concludes the chapter.

6.2 Data Model and Problem Definitions

The data model used in this chapter is an extension of the attribute uncertainty model (presented in Chapter 3) for categorical uncertain data. Under the categorical uncertainty model [12], a relation can have attributes that are allowed to take on uncertain values. For the sake of simplicity, we limit the discussion to relations with a single uncertain attribute, although the model makes no such restriction. The focus of this chapter is on uncertain attributes that are drawn from categorical domains. We shall call such an attribute an *uncertain discrete attribute* (UDA)[1]. Let $R.a$ be a particular attribute in relation R which is uncertain. $R.a$ takes values from the categorical domain D with cardinality $|D| = N$. For a regular (certain) relation, the value of an attribute a for each tuple, $t.a$, would be a single value in D, i.e., $t.a \in D$. In the case of an uncertain relation, $t.a$ is a probability distribution over D instead of a single value. Let $D = \{d_1, d_2, ..., d_N\}$, then $t.a$ is given by the probability

[1]In this chapter, we use the term *discrete* to mean discrete categorical data. The alternative to this is discrete numeric data, on which some more operations can be defined, is not the focus of the chapter.

distribution $Pr(t.a = d_i)$ for all values of $i \in \{1, ..., N\}$. Thus, $t.a$ can be represented by a probability vector $t.a = \langle p_1, p_2, ..., p_N \rangle$ such that[2] $\sum_{i=1}^{N} p_i = 1$. In many cases, the probability vector is sparse and most p_is are zeros. In such cases, we may represent $t.a$ by a set of pairs $\{(d, p) | (Pr(t.a = d) = p) \wedge (p \neq 0)\}$. Hereafter we denote a UDA by u instead of $t.a$ unless noted otherwise. Also, we denote $Pr(u = d_i)$ by $u.p_i$.

Table 6.1 shows an example of a CRM application with UDA attribute `Problem`. The `Problem` field is derived from the `Text` field in the given tuple using a text classifier. A typical query on this data would be to report all the tuples which are highly likely to have a brake problem (i.e., `Problem = Brake`). Table 6.2 shows a table from a personnel planning database where `Department` is uncertain field. Again, one might be interested in finding employees which are highly likely to be placed in the `Shoes` or `Clothes` department. Formally we define UDA as follows.

Table 6.1
Example Uncertain Relation: CRM Application.

Make	**Location**	**Date**	**Text**	**Problem**
Explorer	WA	2/3/06	$\cdots$	{(Brake, 0.5), (Tires, 0.5)}
Camry	CA	3/5/05	$\cdots$	{(Trans, 0.2, (Suspension, 0.8)}
Civic	TX	10/2/06	$\cdots$	{(Exhaust, 0.4), (Brake, 0.6)}
Caravan	IN	7/2/06	$\cdots$	{(Trans, 1.0)}

Definition 6.2.1 *Given a discrete categorical domain $D = \{d_1, .., d_N\}$, an uncertain discrete attribute (UDA) u is a probability distribution over D. It can be represented by the probability vector $u.P = \langle p_1, ..., p_N \rangle$ such that $Pr(u = d_i) = u.p_i$.*

[2]We wish to note that the sum can be < 1 in the case of missing values, and our model can also handle this case without any changes. In this chapter, we do not concern ourselves with this issue and assume that the sum is 1.

Table 6.2
Example Uncertain Relation: Personnel Planning Database.

Employee	**Department**
Jim	{(Shoes, 0.5),(Sales, 0.5)}
Tom	{(Sales, 0.4), (Clothes, 0.6)}
Lin	{(Hardware, 0.6), (Sales, 0.4) }
Nancy	{(HR, 1.0)}

Semantically, we assume that the uncertainty is due to lack of knowledge of the exact value. However, the actual value of attribute is just one of the given possibilities. With this interpretation, we define the semantics of operators on UDAs. Given an element $d_i \in D$, the equality of $u = d_i$ is a probabilistic event. The probability of this equality is given by $Pr(u = d_i) = p_i$. The definition can be extended to equality between two UDAs u and v under the independence assumption as follows:

Definition 6.2.2 *Given two UDAs u and v, the probability that they are equal is given by $Pr(u = v) = \sum_{i=1}^{N} u.p_i \times v.p_i$.*

This definition of equality is a natural extension of the usual equality operator for certain data. As with the regular equality operator, this uncertain version can be used to define operations such as joins over uncertain attributes. The notion of equality is a very important concept in modeling uncertainty of attributes. This enables further operations like joins in uncertain databases. It is semantically consistent and is also important for propagating join results in a hierarchy. Example uses of this operator are to compute the probability of pairs of cars having the same problem, or of two employees working for the same department. Analogous to the notion of equality of value is that of *distributional similarity.*

Distribution similarity is the inverse of distributional divergence, which can be seen as a distance between two probability distributions. We consider the following distance functions between two distributions:

L_1: $L_1(u, v) = \sum_{i=1}^{N} |u.p_i - v.p_i|$. This is the Manhattan distance between two distributions.

L_2: $L_2(u, v) = \sqrt{\sum_{i=1}^{N} (u.p_i - v.p_i)^2}$. This is the Euclidean distance between two distributions.

$KL(u, v)$: $KL(u, v) = \sum_{i=1}^{N} u.p_i \log(u.p_i/v.p_i)$. This is Kullback-Leibler (KL) divergence based on cross entropy measure. This measure comes from information theory. Unlike the above two, this is not a metric. Hence it is not directly usable for pruning search paths but can be used for clustering in an index [48].

Divergence functions such as KL which tend to compare the probability values by their ratios are also important in equality based indexing. Since each probability value in the computation of equality probability is multiplied by a scaling factor, it is meaningful to consider ratios. If UDA u has a high equality probability with UDA q, and $KL(u, v)$ is small, then v is also likely to have a high equality probability with q. This principle is used to cluster UDAs for efficiently answering queries.

There is one major distinction between the notions of distributional similarity and equality between two UDAs. Two distributions may be exactly similar but can have less probability of being equal than two unequal distributions. For example, consider the case where two UDAs u and v have the same vector: $\langle 0.2, 0.2, 0.2, 0.2, 0.2 \rangle$. In this case, $Pr(u = v) = 0.2$. However, if $u = \langle 0.6, 0.4, 0, 0, 0 \rangle$ and $v = \langle 0.4, 0.6, 0, 0, 0 \rangle$, the probability of equality, $Pr(u = v) = 0.48$, is higher even though they are very different in terms of distributional distance.

Having defined the model and primitives, we next define the basic query and join operators. We define equality queries, queries with probabilistic thresholds and queries which give top-k most probable answers. For each of these queries we can define a corresponding join operator.

Definition 6.2.3 *Probabilistic equality query (PEQ): Given a UDA q, and a relation R with a UDA a, the query returns all tuples t from R, along with probability values, such that the probability value $Pr(q = t.a) \geq 0$.*

Often with PEQ there are many tuples qualifying with very low probabilities. In practice, only those tuples which qualify with sufficiently high probability are likely to be of interest. Hence the following queries are more meaningful: (1) equality queries which use probabilistic thresholds [12], and (2) equality queries which select k tuples with the highest probability values.

Definition 6.2.4 *Probabilistic equality threshold query (PETQ): Given a UDA q, a relation R with UDA a, and a threshold τ, $\tau \geq 0$. The answer to the query is all tuples t from R such that $Pr(q = t.a) \geq \tau$.*

An example PETQ for the data in Table 6.2 determines which pairs of employees have a given minimum probability of potentially working for the same department. In a medical database with an uncertain attribute for possible diagnoses, a PETQ query can be used to identify patients that have similar problems. Analogous to PETQ, we define the top-k query PEQ-top-k, which returns the k tuples with the highest equality probability to the query UDA. Such a query can determine the k patients that are most similar to a given patient in terms of their likely diseases. In our indexing framework, the top-k queries are executed essentially using threshold queries. This is achieved by dynamically adjusting the threshold τ to the kth highest probability in the current result set, as the index processes candidates.

Similar to probabilistic equality-based queries, we can define all of the above queries with distributional similarity. Instead of equality probability, the measure here is distributional similarity. Given a divergence threshold, τ_d, the tuples which qualify for query with UDA q are those whose distributional distance with q is at most τ_d. These are called *distributional similarity threshold queries* (DSTQ).

Definition 6.2.5 *DSTQ: Given a UDA q, a relation R with UDA a, a threshold τ_d, and a divergence function F, DSTQ returns all tuples t from R such that $F(q, t.a) \leq \tau_d$.*

There is again a similar notion for DSQ-top-k. The distributional distance can be any of the divergence functions (L_1, L_2, KL) defined above. An example application of a DSTQ is to find similar documents (e.g. web pages) in collections of documents. Although the focus of this chapter is on probabilistic equality queries, it is straightforward to adapt our framework of indexing to distributional similarity queries. In addition, distributional distance is a key concept used for clustering in one of our indexes.

We can extend the select query operators above to join operators. Given two UDAs u and v, and a probability threshold τ, u joins with v if and only if $Pr(u, v) \geq \tau$. Thus, given two relations R and S both having UDA a, we can define threshold equality join:

Definition 6.2.6 *Given two uncertain relations R, S both with UDAs a, b, respectively, relation $R \bowtie_{R_a = S_b, \tau} S$ consists of all pairs of tuples r, s from R, S respectively such that $Pr(r.a = s.b) \geq \tau$. This is called probabilistic equality threshold join (PETJ).*

This definition may also be extended to define PEJ-top-k, DSTJ, and DSJ-top-k joins. We wish to note here that joining does introduce new correlations between the resultant tuples and they are no longer independent of each other. Our model only includes the selection based on thresholds. Tracking dependencies requires keeping track of lineage and is not considered in this chapter.

Although this chapter addresses the general case of categorical uncertainty, it should be noted that for the special case of totally ordered categorical domains, e.g., $D = \{1, .., N\}$, additional inequality probabilistic relations and operators can be defined between two UDAs. For example, we can define $Pr(u \geq v)$, and $Pr(|u - v| < c)$. The notion of probabilistic equality can be slightly relaxed to allow a window within which the values are considered equal. The techniques require to index these queries are discretized versions of those in [30].

6.3 Index Structures

In this section, we describe our index structures to efficiently evaluate queries and joins defined in the previous section. We develop two types of index structures: (1) Inverted index based structures, and (2) R-tree based structures. Although both structures have been explored for indexing set attributes [40, 41], the extension to the case of uncertain data with probabilities attached to members is not straightforward. Experimental results show there is no clear winner between these two index structures. Section 6.4 discusses the advantages and disadvantages of each structure with respect to performance, depending on the nature of data and queries.

6.3.1 Probabilistic Inverted Index

Inverted indexes are popular structures in information retrieval [49]. The basic technique is to maintain a list of lists, where each element in the outer list corresponds to a domain element (i.e. the words). Each inner list stores the ids of documents in which the given word occurs, and for each document, the frequencies at which the word occurs. Traditional applications assume these inner lists are sorted by document ID. We introduce a probabilistic version of this structure, in which we store for each value in a categorical domain D a list of tuple-ids potentially belonging to D. Along with each tuple-id, we store the probability value that the tuple may belong to the given category. In contrast to the traditional structure, these inner lists are sorted by descending probabilities. Depending on the type of data, the inner lists can be long. In practice, these lists (both inner or outer) are organized as dynamic structures such as B-trees, allowing efficient searches, insertions, and deletions.

Figure 6.1 shows an example of a probabilistic inverted index. At the base of the structure is a list of categories storing pointers to lists, corresponding to each item

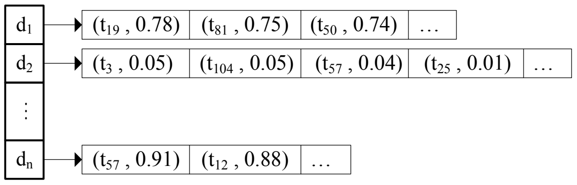

Figure 6.1. Probabilistic Inverted Index.

in D that occurs in the dataset. This is an inverted array storing, for each value in D, a pointer to a list of pairs. In the list $d_i.list$ corresponding to $d_i \in D$, the pairs (tid, p) store tuple-ids along with probabilities, indicating that tuple tid contains item d_i with probability p. That is, $d_i.list = \{(tid, p)|Pr(tid = d_i) = p > 0\}$. Again, we sort these lists in order of descending probabilities.

We first describe the insert and delete operations which are relatively more straight forward than search. To insert (delete) a tuple (UDA) tid in the index, we add (remove) the tuple's information in tuple-list. To insert it in the inverted list, we dissect the tuple into the list of pairs. For each pair (d, p), we access the list of d and insert pair (tid, p) in the B-tree of this list. To delete, we search for tid in the list of d and remove tid from the list.

Next we describe search algorithms to answer the PETQ query given a UDA q and threshold τ. Let $q = \langle(d_{i_1}, p_{i_1}), (d_{i_2}, p_{i_2}), ..., (d_{i_l}, p_{i_l})\rangle$ such that $p_{i_1} \geq p_{i_2} \geq ... \geq p_{i_l}$. We first describe the brute force inverted index search which does not use probabilistic information to prune the search. Next we shall describe three heuristics by which the search can be concluded early. These methods search the tuples in decreasing probability order, stopping when no more tuples are likely to satisfy the threshold τ. These optimizations are especially useful when the data or query is likely to contain many insignificantly low probability values. The three methods differ mainly in their stopping criteria and searching directions. Depending on the nature of queries and data, one may be preferable over others.

Inv-index-search: This follows the brute-force inverted index based lookup. For all pairs (d_{i_j}, p_{i_j}) in q, we retrieve all the tuples in the list corresponding to each d.

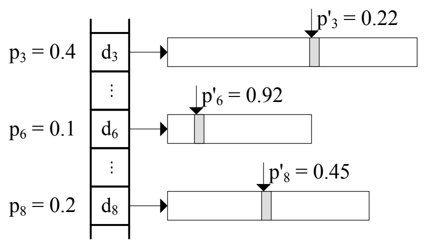

Figure 6.2. Highest-prob-first Search for $q = \langle (d_3, 0.4), (d_8, 0.2), (d_6, 0.1) \rangle$.

Now, from these candidate tuples we match with q to find out which of these qualify more than the threshold. This is a very simple method, and in many cases when these lists are not too big and the query involves fewer d_{i_j}, this could be as good as any other method. However, the drawback of this method is that it reads the entire list for every query.

Highest-prob-first: Here, we simultaneously search the lists for each d_{i_j}, maintaining in each $d_{i_j}.list$ a current pointer of the next item to process (see Figure 6.2). Let p'_{i_j} be the probability value of the pair pointed by the current pointer in this list. At each step, we consider the most promising tuple-id. That is, among all the tuples pointed by current pointers, move forward in that list of d_j where the next pair (tid, p'_{i_j}) maximizes the value $p'_{i_j} p_{i_j}$. The process stops when there are no more promising tuples. This happens when the sum of all current pointer probabilities scaled by their probability in query q falls below the threshold, i.e. when $\sum_{j=1}^{l} p'_{i_j} p_{i_j} < \tau$. This works very well for top-k queries when k is small.

Row Pruning: In this approach, we employ the naive inverted index search but only consider lists of those items in D whose probability in query q is higher than threshold τ. It is easy to check that a tuple, all of whose items have probability less than τ in q, can never meet the threshold criteria. For processing top-k using this approach, we can start examining candidate tuples as we get them and update the threshold dynamically.

Column Pruning: This approach is orthogonal to the row pruning. We retrieve all the lists which occur in the query. Each of these lists is pruned by probability τ. Thus, we ignore the part of the lists which have probability less than the threshold τ. This approach is more conducive to top-k queries.

Note that the above methods require a random access for each candidate tuple. If the candidate set is significantly larger than the actual query answer, then this may result in too many I/Os. We also use no-random-access versions of these algorithms. Nevertheless, we first argue the correctness of our stopping criteria. This applies to all three of the above cases.

Lemma 6 *Let the query* $q = \{(d_{i_j}, p_{i_j}) | 1 \leq j \leq l\}$ *and threshold* τ. *Let* p'_{i_j} *be probability values such that* $\sum_{j=1}^{l} p_{i_j} p'_{i_j} < \tau$. *Then, any tuple tid which does not occur in any of the* d_{i_j}*.list with probability at least* p'_{i_j}*, cannot satisfy the threshold query* (q, τ).

Proof For any such tuple tid, $tid.p_{i_j} \leq p'_{i_j}$. Hence, $\sum_{j=1}^{l} p_{i_j} tid.p_{i_j} < \tau$. Since q only has positive probability values for indices i_j's, $Pr(q = tid) < \tau$. □

In many cases, the random access to check whether the tuple qualifies performs poorly as against simply joining the relevant parts of inverted lists. Here, we use rank-join algorithms with early-out stopping [50, 51]. For each tuple so far encountered in our search, we maintain its lack parameter – the amount of probability value required for the tuple, and which lists it could come from. As soon as the probability values of required lists drop below a certain boundary such that a tuple can never qualify, we discard the tuple. If at any point the tuple's current probability value exceeds the threshold, we include it in the result set. The other tuples remain in the candidate set. A list can be discarded when no tuples in the candidate set reference it. Finally, once the size of this candidate set falls below some number (predetermined or determined by ratio to already selected result) we perform random accesses for these tuples.

6.3.2 Probabilistic Distribution R-Tree (PDR-Tree)

In this subsection, we describe an alternative indexing method based on the R-tree [52]. In this index, each UDA u is stored in a page with other similar UDAs which are organized as a tree. The tree-based approach is orthogonal to the inverted index approach where each UDA is shredded and indexed by its components. Here, the entire UDA is stored together in one of the leaf pages of the tree.

Conceptually, we can consider each UDA u as a point in high-dimensional space R^N. These points are clustered to form an index. A major distinction with the regular R-tree is that the queries for uncertain data have very different semantics. They are equivalent to hyperplane queries on the N-dimensional cube. Thus a straight-forward extension of the R-tree or related structures is inefficient due to the nature of queries and the curse of dimensionality (as the number of dimensions – the domain size – can be very large).

We now describe our structure and operations by analogy to the R-tree. We design new definitions and methods for *Minimum Bounding Rectangles* (MBR), the area of an MBR, the MBR boundary, splitting criteria and insertion criteria. The concept of distributional clustering is central to this index. At the leaf level, each page contains several UDAs (as many as fit in one block) using the aforementioned *pairs* representation. Each list of pairs also stores the number of pairs in the list. The page stores the number of UDAs contained in it. Figure 6.3 shows an example of a PDR-tree index.

Each page can be described by its MBR boundaries. The MBR boundary for a page is a vector $v = \langle v_1, v_2, ..., v_N \rangle$ in R^N such that v_i is the maximum probability of item d_i in any of the UDA indexed in the subtree of the current page. We maintain the essential pruning property of R-trees; if the MBR boundary does not qualify for the query, then we can be sure that none of the UDAs in the subtree of that page will qualify for the query. In this case, for good performance it is essential that we only insert a UDA in a given MBR if it is sufficiently tight with respect to its boundaries.

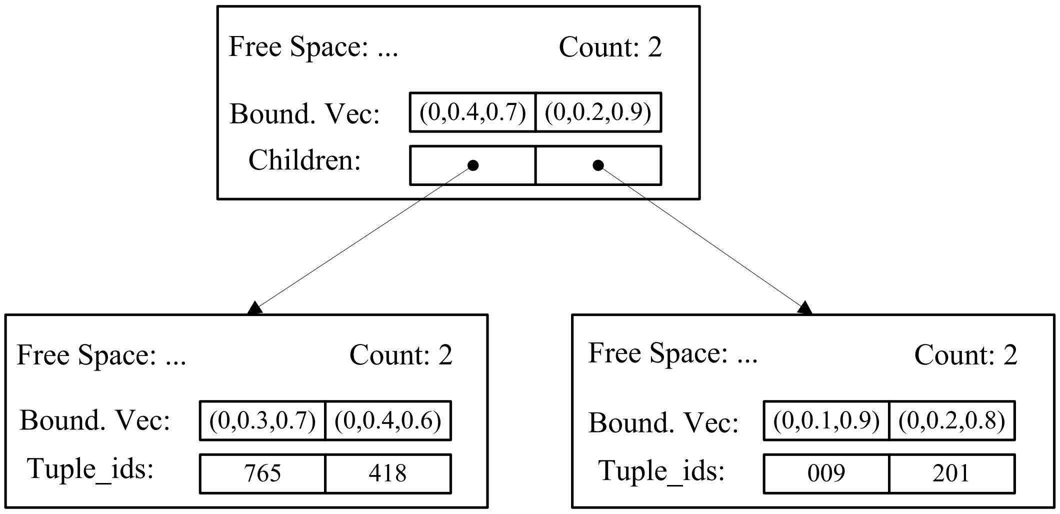

Figure 6.3. Probabilistic Distribution R-tree.

This will be further explained when we discuss insertion. There are several measures for the "area" of an MBR, the simplest one being the L_1 measure of the boundaries, which is $\sum_{i=1}^{N} v_i$. Our methods are designed to minimize the area of any MBR. Next, we describe how insert, split and PETQ are performed.

Insert(u): To insert a UDA into a page, we first update its MBR information according to u. Next, from the children of the current page we pick the best page to accommodate this new UDA. The following criteria (or combination of these) are used to pick the best page: (1) Minimum area increase: we pick a page whose area increase is minimized after insertion of this new UDA; (2) Most similar MBR: we use distributional similarity measure of u with MBR boundary. This makes sure that even if a probability distribution fits in an MBR without causing an area increase, we may not end up having too many UDAs which are much smaller in probability values. Minimizing this will ensure that we do not hit too many non qualifying UDAs when a query accepts (does not prune) an MBR. Even though an MBR boundary is not a probability distribution in the strict sense, we can still apply most divergence measures described in Section 6.2.

Split( ): There are two alternative strategies to split an overfull page: *top-down* and *bottom-up*. In the top-down strategy, we pick two children MBRs whose boundaries are distributionally farthest from each other according to the divergence measures. With these two serving as the seeds for two clusters, all other UDAs are inserted into the closer cluster. An additional consideration is to create a balanced split, so that two new nodes have a comparable number of objects. No cluster is allowed to contain more that $\frac{3}{4}$ of the total elements. In the bottom-up strategy, we begin with each element forming an independent cluster. In each step the closest pair of clusters (in terms of their distributional distance) are merged. This process stops when only two clusters remain. As with the top-down approach, no cluster is allowed to contain more than $\frac{3}{4}$ of the total elements.

PETQ(q, τ): Given the structure, the query algorithm is straightforward. We do a depth-first search in the tree, pruning by MBRs. Let $\langle\langle \cdot, \cdot \rangle\rangle$ denote the dot-product of two vectors. For a node c, let $c.v$ denote its MBR boundary vector. If an MBR qualifies for the query, i.e., if $\langle\langle c.v, q \rangle\rangle \geq \tau$, our search enters the MBR, else that branch is pruned. At the leaf level, we evaluate each UDA in the page against the query and output the qualifying ones. For top-k queries, we need to upgrade the threshold probability dynamically during the search. An efficiency improvement over the raw depth-first search is to greedily select that child node c first for which $\langle\langle c.v, q \rangle\rangle$ is the maximum. This way we can upgrade our threshold quickly by finding better candidates at the beginning of the search which in turn results in better pruning.

The following lemma proves the correctness of the pruning criteria.

Lemma 7 *Consider a node c in the tree. If $\langle\langle c.v, q \rangle\rangle < \tau$ then no UDA stored under the subtree of c qualifies for the threshold query (q, τ).*

Proof Consider any UDA u stored in the subtree of c. Since an MBR boundary is formed by taking the point-wise maximum of its children MBR boundaries, we can show by induction that $u.p_i \geq c.v.p_i$ and $q_i \geq 0$ for any i, $\langle\langle u, q\rangle\rangle < \langle\langle c.v, q\rangle\rangle < \tau$. Thus, u cannot qualify. □

6.3.3 Compression Techniques

An issue that was overlooked earlier is the description of MBR boundaries. Note that an MBR boundary may be described in terms of $|D|$ floating-point values. This may be space inefficient if the data domain is large. Consider the case when $|D| = 1000$ and page size is $8K$. The description of an MBR boundary may not just fit in a page. This results in a small constant fan-out for the index structure. The MBR description does not need to be precise and can be stored in approximate form. Thus, we can apply some lossy compression techniques. With this, the length of the representation of an MBR becomes variable. These variable length objects are packed appropriately. The compression technique needs to make sure that pruning correctness is not compromised. Hence the lossy representation of an MBR boundary vector must be an over-estimation of the actual values. There are two orthogonal approaches to this compression:

Set-Signature based approach: In this case, we define a function $f : D \rightarrow C$ where $|C| < |D|$. Thus C is the compressed domain. In a given compressed distribution $Pr(c_i) = \max\{Pr(d_j) | f(d_j) = c_i\}$. This approach is akin to that taken by signature trees for set-values attributes [41]. Good correlation detection and clustering methods ensure meaningful f and $|C|$.

Discretized-overestimation: This reduces the number of bits required to represent each p_i in a UDA. Say we allow 2 bits (instead of 4 bytes) to represent each p_i. Then, we essentially approximate p_i by multiple of 0.25 which is greater than p_i. For example, a value of 0.62 will be mapped to 0.75 and can be represented in 2 bits by representing the multiplier 3. When considering more slabs, we may be able to code each multiplier using an optimal number of bits as per its frequency and achieve entropy coding. This also substantially reduces the size of the MBR boundary description.

6.4 Experimental Evaluation

In this section we present the experimental evaluation of the proposed index structures using real and synthetic datasets. The real dataset is generated by text clustering/categorization of customer service constraints for a major cell phone service provider in the context of CRM databases. The base data consists of 100,000 text documents consisting of complaints, responses, and ensuing communications between customers and service representatives. The dataset CRM1 consists of probability values generated by automatic categorization of the text into 50 categories. Dataset CRM2 is generated by unsupervised fuzzy clustering of the text [53, 54]. Each tuple has a fuzzy membership among 50 clusters.

The synthetic datasets are generated to simulate varying degrees of correlation and sparsity. The Uniform dataset has 5 items and the probability of each item is chosen randomly for all tuples. The Pairwise dataset also has 5 elements but the individual tuples have only 2 non-zero items with roughly equal probabilities. In addition, the total number of item combinations is restricted to 5. Both these datasets have $10k$ tuples. These two datasets represent the two extreme possible scenarios that our algorithms can face.

The dataset Gen3 used for studying scalability with domain size is also generated synthetically. Initially, a number of item groups are picked at random from the domain. The size of the item groups, which determines the fill factor (expected number of non-zero items in a tuple), is distributed geometrically. The expected group size was varied from 3 (in domain size 10) to 10 (in domain size 500). The item probabilities inside a group are chosen randomly.

All experiments are conducted with page size of 8 KB. We measure the number of I/O operations performed for processing queries. We test both equality threshold

(PETQ) and PETQ-top-k queries. Multiple thresholds and values for k are considered in order to produce queries with varying selectivities. All graphs shown below report the number of I/O operations for executing queries. In order to simulate the effect of buffering, all experiments are conducted with a buffer manager that allocates 100 blocks to each query. A clock replacement algorithm is used to manage the buffer pool.

6.4.1 Results

Most of the graphs below show how performance (measured in disk I/Os on the y-axis) is affected by the selectivity of the queries (shown as a percentage on the x-axis).

Divergence Measures

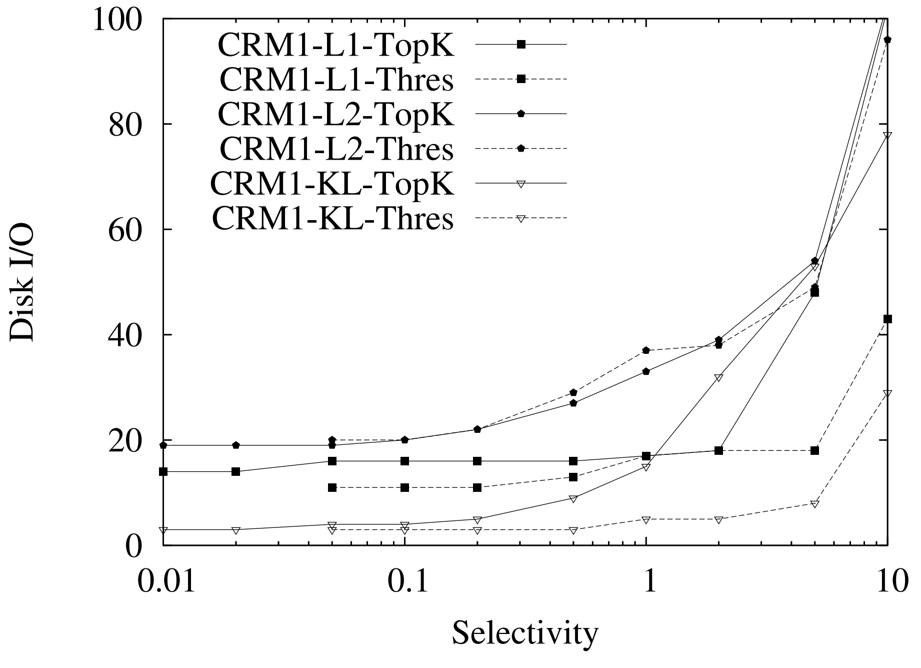

Figure 6.4. L1 vs L2 vs KL (PDR-tree).

The first experiment studies the relative performance of the three distribution similarity measures, L_1, L_2, and KL. The results for the CRM1 dataset are shown in Figure 6.4. The x-axis shows the query selectivity and the y-axis shows the number of disk I/O per query. For low selectivities, the KL measure clearly outperforms L_1 which in turn outperforms L_2. For high selectivities, all three perform similarly for top-K queries while the trend for threshold queries remains the same. The superior performance of KL was observed consistently in all our experiments. Consequently, we do not present the performance of L_1 and L_2 in the remainder of this section. We can also observe that for a given selectivity, the performance of top-k queries is poorer than that of threshold queries by roughly a constant factor. This is because a top-k query needs to explore more tuples in order to guarantee that the selected top-k tuples do indeed give the largest probabilities. This relative behavior of top-k queries versus threshold queries was observed in all our experiments.

Synthetic Data

In this experiment we compare the performance of the two index structures for synthetic datasets: Uniform and Pairwise. The results are shown in Figure 6.5. The x-axis shows the query selectivity (as a percentage), and the y-axis shows the number of disk I/O per query. For the Uniform dataset, the performance of the inverted index is clearly inferior to that of the PDR-tree. Because each data item included nonzero probabilities in many categories, evaluating the query results in accessing large numbers of lists in the inverted index structure. For the Pairwise dataset, the inverted index yields a much better performance than for the Uniform data. However, the PDR-tree continues to outperform the inverted index even in this case.

Real Datasets

This experiment compares the performance of the two index structures for the two real datasets, CRM1 and CRM2. The results for CRM1 are shown in Figure 6.6

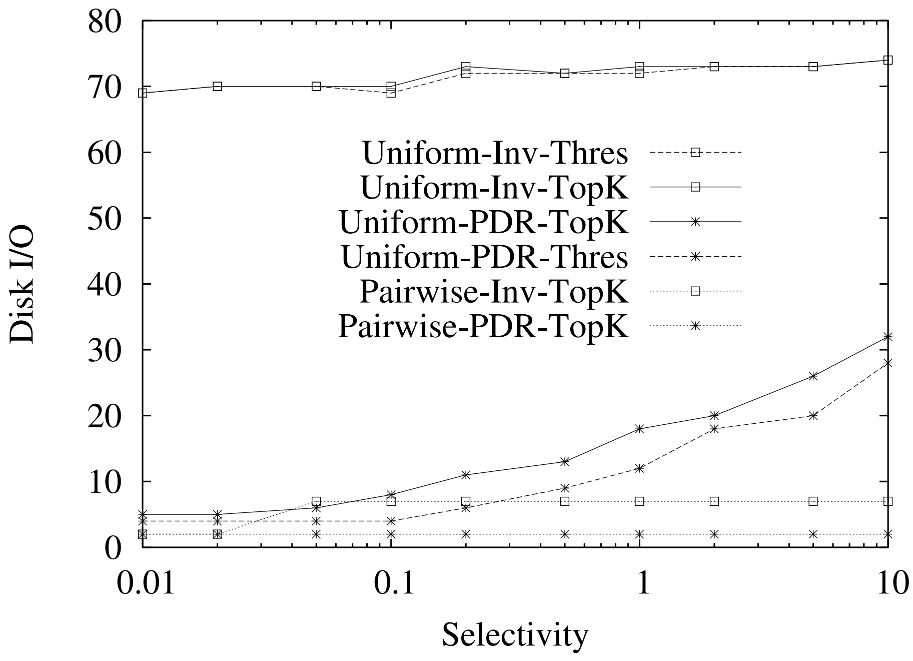

Figure 6.5. Inverted Index vs PDR-tree (synth).

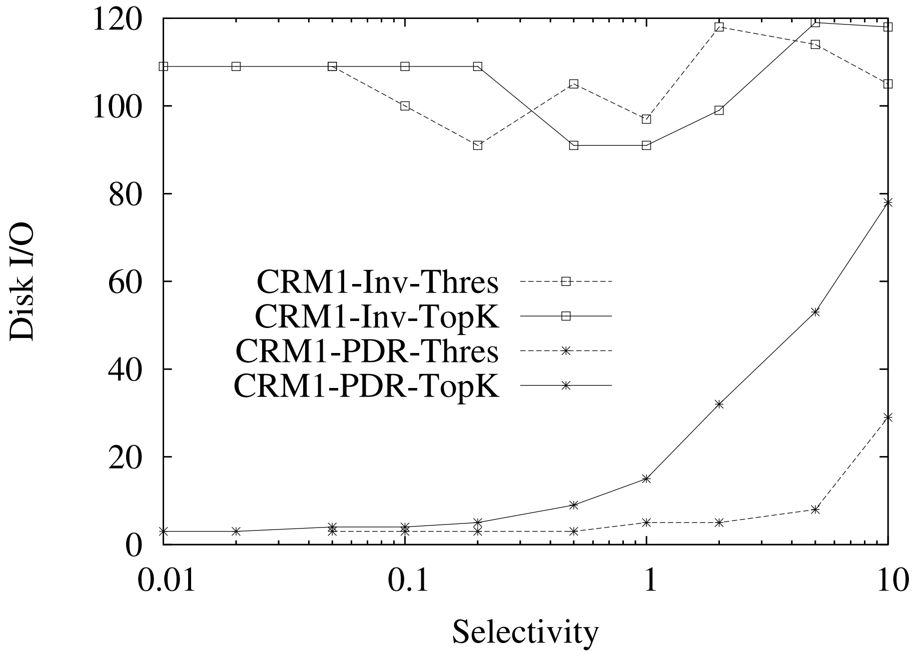

Figure 6.6. Inverted Index vs PDR-tree (CRM1).

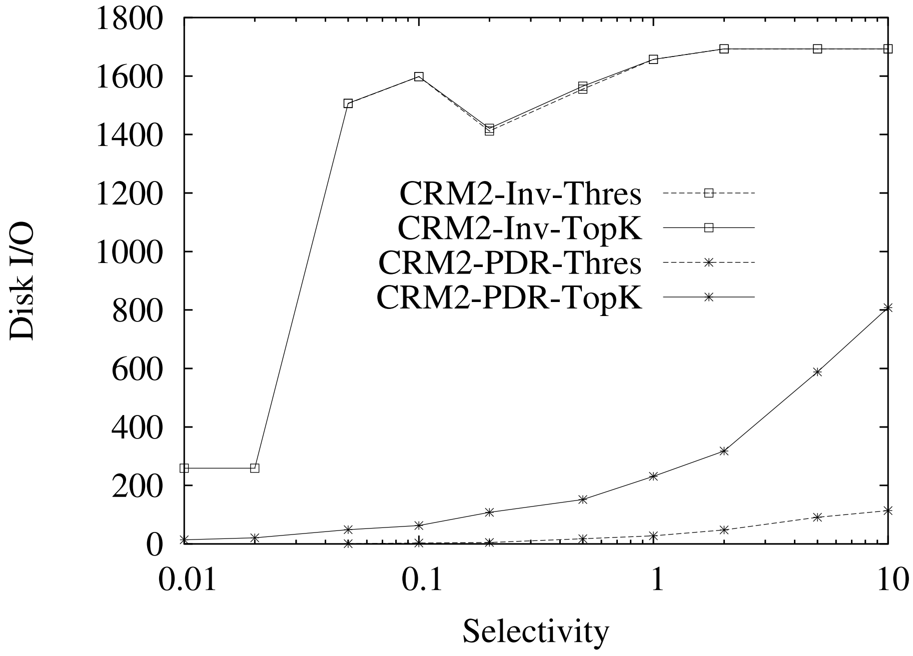

Figure 6.7. Inverted Index vs PDR-tree (CRM2).

and those for CRM2 are shown in Figure 6.7. The overall relative performance of is the same as that for the synthetic datasets. That is, the PDR-tree significantly outperforms the inverted index. Since CRM1 is classification-based data using a training set, it exhibits less uncertainty that CRM2 which is based on unsupervised clustering. Consequently, CRM1 is a sparse dataset while CRM2 is more dense. As a result, the performance for CRM1 is about 10 times better than that for CRM2.

Dataset Size

This experiment studies the scalability of the index structures as the size of the dataset is increased. The test is run using the CRM2 data by indexing differing numbers of tuples. Figure 6.8 shows the results. The x-axis plots the number of tuples in thousands, and the y-axis plots the number of disk I/O per query. As expected, the inverted index scales linearly with dataset size, while the PDR-tree scales sub-linearly.

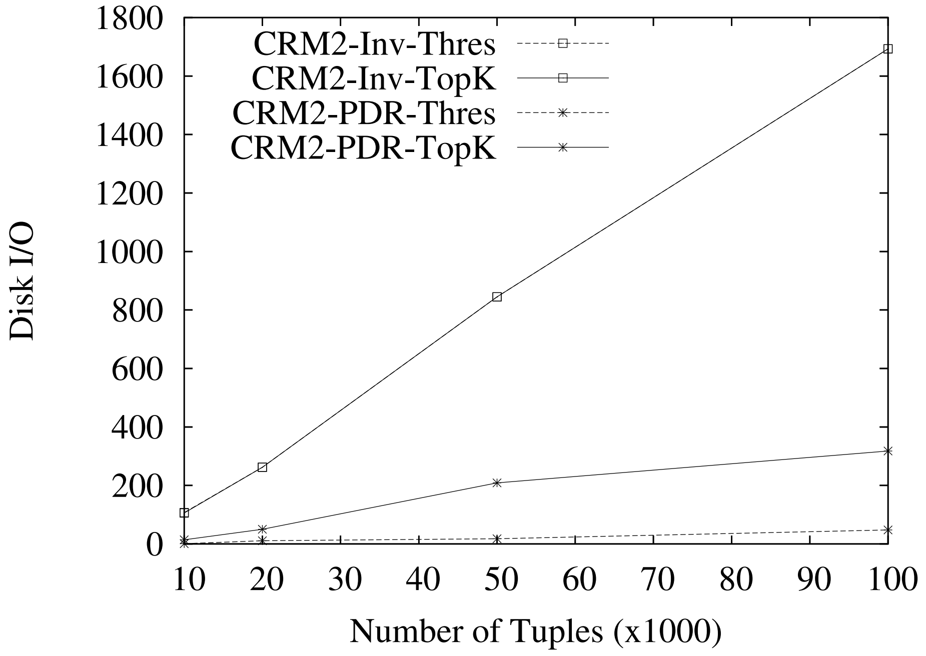

Figure 6.8. Scalability with Size of Data.

Domain Size

We now explore the impact of the domain size on index performance. In order to test this behavior, we generate another dataset, Gen3, for which we vary the number of items in the domain from 5 to 500. The number of non-zero entries is in the range of 3 to 10. The results are shown in Figure 6.9. As the domain size increases, the inverted index improves in performance. This can be attributed to the reduction in the average length of each list as the number of lists increases with domain size (since there is one list for each value in the domain). The charts for the PDR-tree show an initial increase followed by a decrease as the domain size increases. We believe this behavior is related to the data generation process. In particular, the relative number of non-zero entries at both ends of our experimental space are smaller than in the middle. This increase in the relative number of non-zero entries in the middle of the range results in poorer clustering for the PDR-tree.

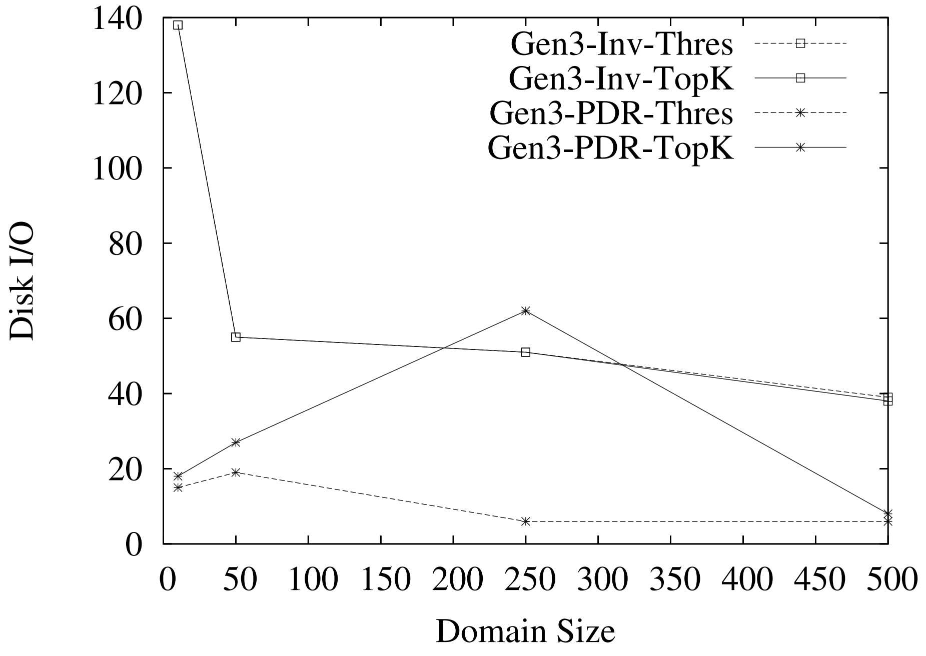

Figure 6.9. Scalability with Domain Size.

PDR Split Algorithm

The final experiment studies the relative performance of the *top-down* and *bottom-up* strategies for the split algorithm of the PDR-tree. Figure 6.10 shows the results with the Uniform dataset. We find that the top-down alternative gives worse performance than the bottom-up alternative. The performance of top-down is caused by outliers in the data that result in poor choices for the initial cluster seeds. A similar relative behavior was observed for the other datasets including the real data.

6.5 Chapter Summary

This chapter focused on indexing techniques for categorical uncertain data. Since such uncertainty can be considered an extension of set-values attributes, we proposed the extensions of signature trees and inverted indexes for this problem. Both index structures were shown to have good scalability with respect to dataset and domain

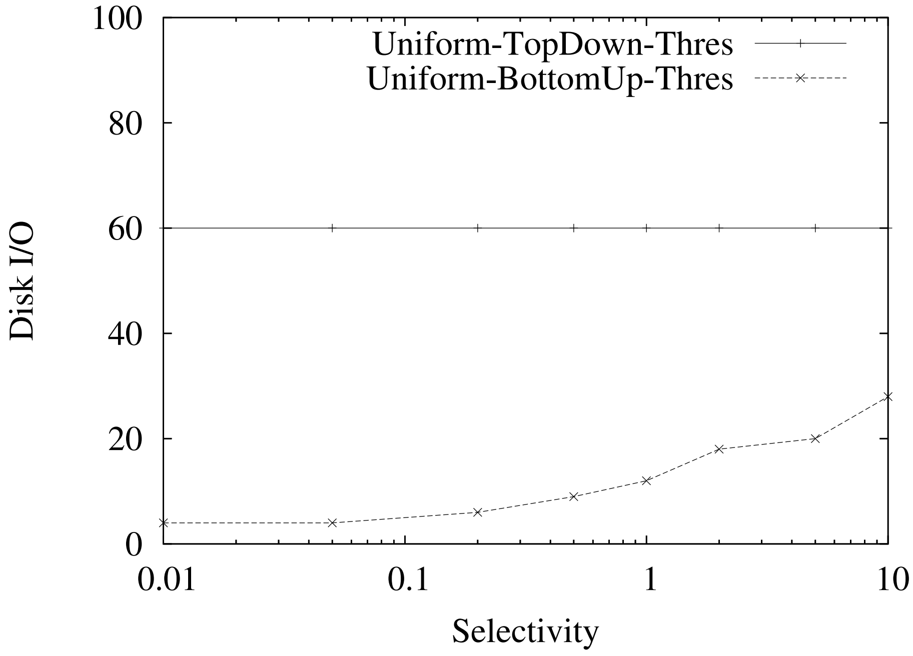

Figure 6.10. Top-down vs Bottom-up Approach.

size. Experimental results showed that each of these structures performed efficiently, but the nature of the data and query parameters appeared to determine their relative performance.

The results of this chapter can be found in [55]. In the next chapter, we discuss query selectivity estimation techniques, which are vital for query optimization in probabilistic databases.

7 QUERY SELECTIVITY ESTIMATION FOR UNCERTAIN DATA

As discussed in the previous chapters, applications requiring the handling of uncertain data have led to the development of database management systems extending the scope of relational databases to include uncertain (probabilistic) data as a native data type. New automatic query optimizations having the ability to estimate the cost of execution of a given query plan, as available in existing databases, need to be developed. For probabilistic data this involves providing selectivity estimations that can handle multiple values for each attribute and also new query types with threshold values. This chapter presents novel selectivity estimation functions for uncertain data and shows how these functions can be integrated into PostgreSQL to achieve query optimization for probabilistic queries over uncertain data. The proposed methods are able to handle both attribute- and tuple-uncertainty. Our experimental results show that our algorithms are efficient and give good selectivity estimates with low space-time overhead.

7.1 Introduction

An important step towards the development of practical uncertain data management systems, is the development of automatic query optimization as is available in existing databases. Toward this end, an essential ingredient is the ability to estimate the cost of execution of a given query plan. For probabilistic data this would involve providing selectivity estimates for probabilistic operators. Currently, there is no work on providing such selectivity estimation functions for probabilistic data. With the availability of these estimation functions it is possible to use existing query optimization techniques that are already built into databases to handle the case of probabilistic data.

In this chapter we address this problem and develop novel selectivity estimation functions for uncertain data. We also show how these functions can be integrated into PostgreSQL to achieve query optimization for probabilistic queries over uncertain data. Selectivity estimation for uncertain data needs to handle multiple values for each attribute and also novel query types with threshold values. Furthermore, an important type of uncertainty transforms a single attribute value to a continuous distribution – this is especially common in sensor databases [2]. The existing cost estimation methods are therefore not applicable for this domain.

The goal of this chapter is to handle selectivity estimation for the two main types of uncertainty that have been proposed in recent work: *tuple uncertainty* [1, 10] and *attribute uncertainty* [11]. In general, selectivity estimation can be used for query processing in addition to its traditional role in query optimization. In this chapter we show how the selectivity estimation algorithms can be used for efficiently processing probabilistic k-nearest neighbor queries. To demonstrate the effectiveness of our selectivity estimation techniques, we conduct experiments with Orion [45]. Although, Orion is tailored towards the attribute uncertainty model, as we will show in Section 7.2, our selectivity estimation techniques are equally applicable to both tuple and attribute uncertainty models.

To sum up, the major contributions of this chapter are as follows:

- We develop efficient algorithms for selectivity estimation of probabilistic threshold queries over uncertain data.

- Based upon an implementation in Orion, we experimentally show that the algorithms are efficient and provide good estimates for query selectivities.

The rest of this chapter is organized as follows. We formally describe the uncertainty model and probabilistic queries in Section 7.2. Our algorithms for selectivity estimation are presented in Section 7.3. Section 7.4 presents an application of selectivity estimation for nearest-neighbor queries. We present the experimental results in Section 7.5, and Section 7.6 concludes this chapter.

7.2 Uncertainty Model

The uncertainty model used in this chapter is presented in Chapter 3. The goal of this chapter is to propose estimation solutions that are applicable to both models of uncertainty: attribute and tuple. For our purposes, we are interested in a single attribute at a time, a (real-valued or integer), for which we are estimating the selectivity. Thus, we can ignore the intra-tuple dependencies. We assume that the uncertainty in the data can be captured in terms of attribute uncertainty. In other words, for the attribute in question, we are able to generate a pdf (f_a) and cdf (F_a) for each tuple of the relation. This is directly available from the attribute uncertainty model. For the case of tuple uncertainty, there are two cases to consider. The first is if there are no x-tuples. In this case, each tuple has a probability value associated with it and is independent of any other tuple. For this case, the pdf for each tuple is simply the single attribute value along with the associated tuple probability. In the second case, the x-tuple itself provides multiple alternatives for the given attribute along with associated probabilities. These are collapsed into a single attribute uncertainty (discrete) pdf.

7.2.1 Operators and Threshold Queries

A number of operators are defined in [47] for comparing uncertain values with both uncertain and certain (precise) values. This chapter focuses on selection queries that compare an uncertain value with precise values. For these queries, we present the definitions for comparing uncertain with certain data. Operators between an uncertain value a and a certain value $v \in \Re$ can be defined as:

$$\begin{aligned}
Pr(a < v) &= \int_{-\infty}^{v} f_a(x)dx = F_a(v) \\
Pr(a > v) &= 1 - F_a(v) \\
Pr(a =_c v) &= \int_{v-c}^{v+c} f_a(x)dx = F_a(v+c) - F_a(v-c) \\
Pr(a \neq_c v) &= 1 - P(a =_c v)
\end{aligned}$$

The probability that a continuous random variable is *exactly* equal to a precise value is 0. In order to circumvent this problem a parameter called *resolution* is used to relax the definition of equality [47]. Note that we can use the exact definition (without c) for defining equality for a discrete distribution, but in order to make things simpler we use the same definition for both cases. This enables us to treat both discrete and continuous data in a similar fashion. In case an exact equality is required the user can always select a small enough c to make sure that the approximate equality ($=_c$) becomes exact equality for discrete distributions.

The set of queries that we consider in the chapter are called *Probabilistic Threshold Range Queries* and were proposed in [30]. These queries are a variant of probabilistic queries where only answers with probability values over a certain threshold τ are returned. With this concept, all the operators discussed above can be changed into boolean predicates by adding a probability threshold to them.

7.2.2 Probabilistic Threshold Index

To efficiently evaluate the PTQ mentioned above, an indexing scheme known as probabilistic threshold index (PTI) was introduced in [30]. The PTI index is very similar to R-Trees but with extra information stored with each node. This information enables improved pruning for threshold queries on probabilistic data. The extra information stored is called *x-bounds*, which are tighter bounds calculated based on the properties of uncertainty pdfs stored within each node. An *x-bound* for a tree node N is defined as:

Definition 7.2.1 *Given* $0 \le x < 1$*, an* **x-bound** *of a node* N *consists of two values, called left-x-bound* ($L_N(x)$)*, and right-x-bound* ($R_N(x)$)*. For every uncertain value* a *contained in* N*, two conditions must hold:*

- *If* $l_a < L_N(x)$*, then* $\int_{l_a}^{L_N(x)} f_a(y)dy \le x$.
- *If* $r_a > R_N(x)$*, then* $\int_{R_N(x)}^{r_a} f_a(y)dy \le x$.

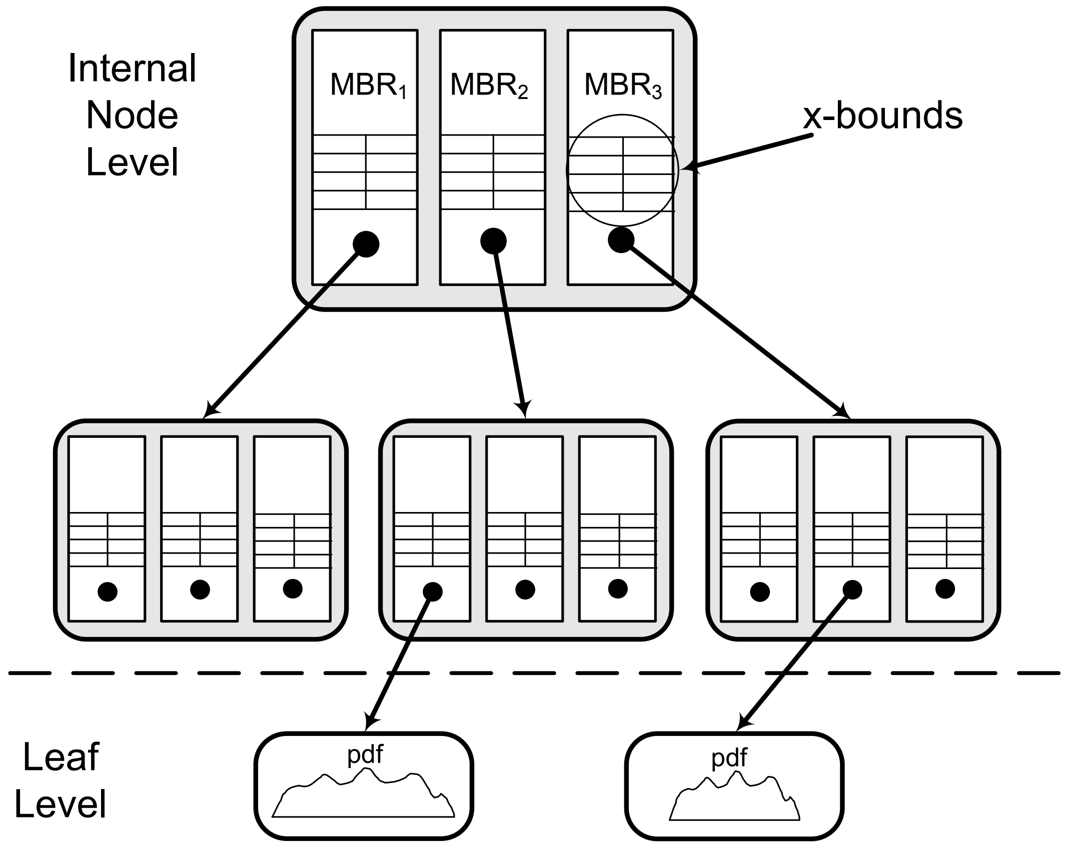

Figure 7.1. Structure of Probabilistic Threshold Index.

The x-bounds for different values of x (e.g. 0, 0.1,...,0.9) are calculated and stored inside the node N as shown in Figure 7.1. For a given threshold range query $[a, b]$

with threshold τ, if we know the x-bounds of a node N, we can eliminate N from further examination if the following two conditions hold:

1. $[a, b]$ does not intersect left-x-bound or right-x-bound of N i.e. $b < L_N(x)$ or $a > R_N(x)$ is true, and
2. $\tau \geq x$

Thus, the presence of x-bounds allows us to decide with ease whether an internal node contains any qualifying MBRs, without further probing into the subtrees of the node. We have used PTI index in our experiments to evaluate PTQ queries, but as discussed in Section 7.3 this index is not well suited for query selectivity estimation.

7.3 Selectivity Estimation

In this section we describe various techniques that can be used for estimating the selectivity for a given probabilistic threshold operator.

7.3.1 Selectivity Estimation using PTI

This is a naïve approach for getting an upper bound on result size. As discussed in Section 7.2, a PTI index can be used to evaluate a threshold range query over uncertain data. To get an estimate of result size, we can use the same index structure but we need to maintain extra information about the descendants of each node. For a node n, we represent the total number of descendants of n by D_n. Intuitively, D_n is the total number of uncertain items that are stored inside node n. For internal nodes, we can calculate D_n by summing up the number of descendants of all its children.

To obtain an estimate of result size, we can reuse the index-based range query processing algorithm with early termination: i.e. we do not evaluate the query beyond some depth d less than the height of the tree. The value of d determines the degree of overestimation of the result set size. If the algorithm returns $n_{m_1}, n_{m_2}, \ldots n_{m_k}$ as the nodes at depth d that are *candidates* satisfying the range query, an upper bound on *actual* number of items R is given by:

$$R \leq \sum_{i=0}^{n} D_{n_{m_i}}$$

To get a good estimate it may be desirable to proceed till the penultimate nodes (d = height of tree - 1). In that case, this naïve approach is not very different (in terms of I/O cost) from actually executing the query. This is because the PTI indexing scheme was originally developed for answering range queries and not for finding estimates. In the next section, we present another kind of structure based on histograms which is tailored for finding estimates for range queries.

7.3.2 Unbounded Range Queries

This approach is based on mapping the uncertain attribute values to a 2-D histogram and estimating the query result size by executing a 2-D box query on the histogram.

To understand the approach, let us consider an unbounded range query Q given by $a <_\tau x_0$, where τ is the probability threshold for the $>$ predicate. This query returns all uncertain items a such that $Pr(a < x_0) > \tau$. In terms of the cumulative distribution function $F_a(x)$, we get the following condition:

$$Pr(a < x_0) > \tau \Leftrightarrow \int_{-\infty}^{x_0} f_a(x)dx > \tau \Leftrightarrow F_a(x_0) > \tau \qquad (7.1)$$

This follows from the definition of pdf and cdf functions.

Let us consider a 2D graph where we plot the cdf function F of all uncertain items. Figure 7.2 shows an example of this graph. The cdfs for three data items a, b, and c are shown. The range query Q given by Equation 7.1 can be translated into a (unbounded) box query $x < x_0$ and $y > \tau$ over this 2D plot (the shaded region in Figure 7.2). Items a and b satisfy the query as they intersect the shaded region.

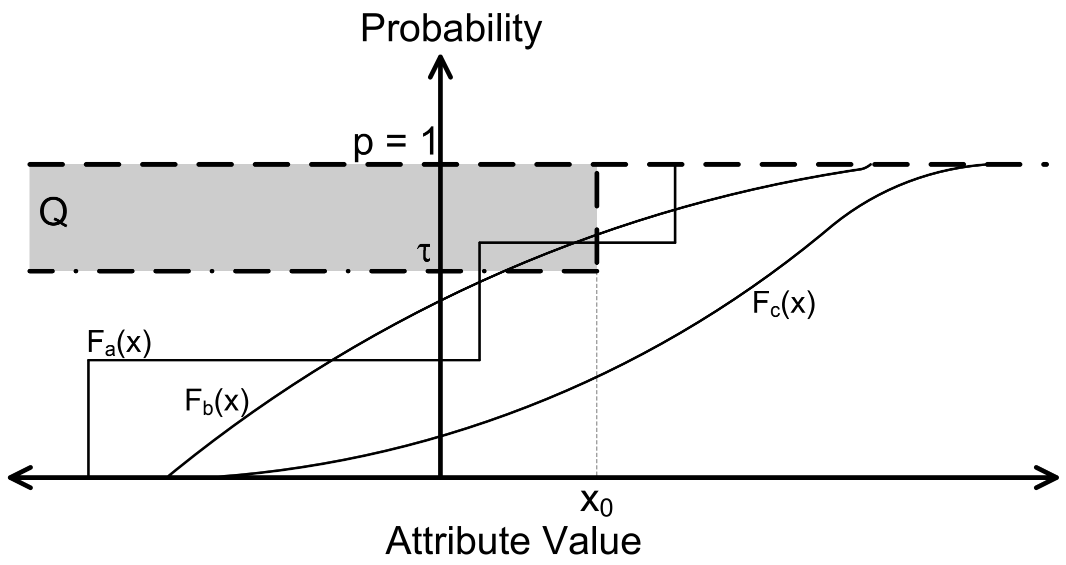

Figure 7.2. Example plot for query $Q(x_0, \tau)$.

Theorem 7.3.1 *All the items whose cdf function $F_a(x)$ lies in the box defined by query Q are part of the result of query Q. That is, $\forall a$, where the cdf function F_a lies in the box defined by query Q, we have $Pr(a < x_0) > \tau$.*

Proof We observe that for any cdf F_a that lies in the box of query Q, we have $F_a(x) > \tau$ for some $x < x_0$. As F_a is a monotonically increasing function, we can deduce that $F_a(x_0) > F_a(x) > \tau$. Using 7.1, $P(a < x_0) > \tau$. □

Now we state the following theorem without proof:

Theorem 7.3.2 *The total number of cdf lines that lie in the query box Q is equal to the number of lines crossing (intersecting) the vertical line-segment given by $\ell : x = x_0, \tau < y \leq 1$, which furthermore is equal to the number of lines crossing (intersecting) the horizontal ray $y = \tau, x < x_0$.*

The proof of this theorem follows from basic geometry and the monotonically increasing nature of cdf F.

Now finding all the items whose cdf function lies in the box defined by a query Q is equivalent to finding the total number of intersections of cdf lines with the vertical line-segment ℓ. To efficiently calculate this number we need to develop an approximation of the above technique. For this purpose, we define a *2-D grid* of histogram over the plot region. Given u_i, $0 \le i < m$ as all the uncertain data items, we define

$$l = \min_i (l_{u_i}), r = \max_i (r_{u_i})$$

where $[l_{u_i}, r_{u_i}]$ is the uncertainty interval of u_i. The plot region is bounded by 0 and 1 in the y (probability) direction and l, r in the x direction. The range R of the histogram is defined as $R = r - l$. The width of the histogram is given by the parameters δ_x and δ_p which represent the size of histogram along x and y (probability) axes respectively. A histogram bucket $H(x, y)$ covers the area given by the box $(x, y, x + \delta_x, y + \delta_p)$. The notations used are summarized in Table 7.1.

Table 7.1
Notations.

Symbol	**Meaning**
f_a	Probability distribution function (pdf) of uncertain item a
F_a	Cumulative distribution function (cdf) of a
l_a, r_a	Left and right bounds of a's interval.
R_a	Range of a, $R_a = r_a - l_a$
u_i	All the uncertain data items ($0 \le i \le m$)
l, r	Leftmost and rightmost limits of all the uncertain intervals
R	Range of input data, $R = r - l$
δ_x, δ_p	Width of histogram bucket along x and y (probability) axis
H	Histogram structure for cost estimation

Definition 7.3.1 *The height of a histogram bucket $H(x, y)$ is the total number of cdf lines of uncertain items intersecting the box $(x, y, x + \delta_x, y + \delta_p)$.*

With this definition, we can now informally describe the algorithm for calculating an approximation (upper-bound) of operator selectivity. Using Theorem 7.3.2 we see that the sum of individual histograms that cover the vertical line-segment ℓ gives a good approximation of the upper-bound of the result set size. The error in this approximation can be reduced by reducing the size of the histogram buckets. This extra accuracy comes at the cost of increased space overhead for storing the histogram structure.

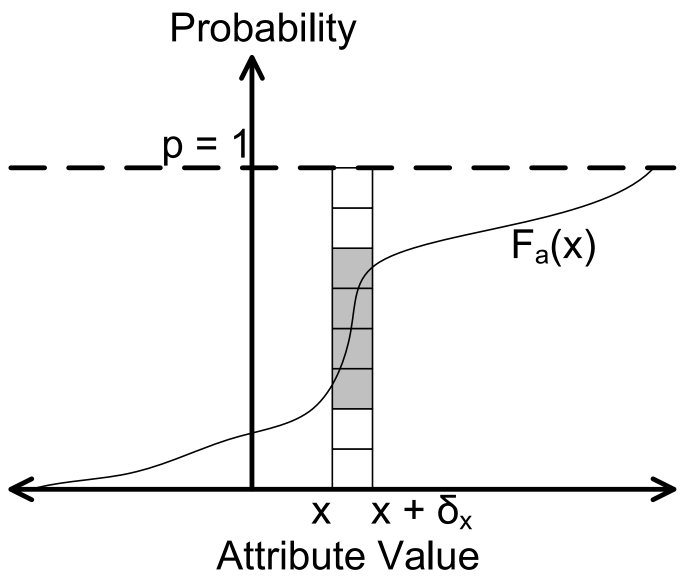

Figure 7.3. Plot showing the case when an item's cdf crosses more than one histogram bar in a vertical window due to its large slope.

As seen in Figure 7.3, if a cdf line has a large slope, it can contribute to more than one histogram in a given vertical window. This will result in over-estimation of the result size because the same cdf line will be counted multiple times. To prevent this, we propose a simple fix: If a cdf line intersects multiple (contiguous) histograms in a given vertical window, we only count its contribution in the *topmost* histogram. With this slight change, we will avoid counting the same line multiple times and obtain a tighter upper bound. Note that by adding the contribution of a given cdf line to the topmost histogram, we are guaranteed that there will be no false negatives. The algorithm for constructing this 2-D histogram is presented in Figure 7.4.

The algorithm presented in Figure 7.4 takes as input the uncertain data items from an attribute and the parameters δ_x and δ_p defining the width of each histogram inside the structure H. In addition to these values, it also takes the l and r values (defined earlier) which represent the spread of input data values. Depending on the attribute domain, these parameters can be provided by the user or the system can select them by random sampling. For a given uncertain item a, we start counting its contribution from its lower bound l_a and stop when we hit the upper-most bucket in the y-direction (Step 1(ii)). This small optimization saves a lot of computations as this step is repeated for all the input uncertain data items. Note that, for the correctness of our algorithm we *do* need to add the contributions to all the successive top buckets for item a. We take care of this *correction* in step 2 with just one pass over the entire histogram.

Given this histogram structure H, we can easily give an approximation for query result size. Figure 7.5 shows the algorithm for finding the selectivity estimate for query $Q(x, \tau) = a <_\tau x$.

Note that the above discussion applies to $a <_\tau x$ queries only. For unbounded range queries of the form $Q : a >_\tau x$, we have the following result:

$$a >_\tau x \Leftrightarrow Pr(a > x) > \tau \Leftrightarrow F_a(x) < 1 - \tau \tag{7.2}$$

Using Equation 7.2 we can see that if an uncertain item a *does not* satisfy the query $a <_{1-\tau} x$ (i.e. $F_a(x) \not> 1 - \tau$) then it will satisfy the query $a >_\tau x$. The algorithms presented in Figures 7.4 and 7.5 can therefore be used for $>_\tau$ queries with slight modifications. The selectivity of $>$ can be calculated by computing the selectivity of $<$ and using the fact that selectivity for $>_\tau$ is 1 - selectivity for $<_{1-\tau}$.

Theorem 7.3.3 *The time complexity of algorithm presented in Figure 7.4 is:*

$$\sum_{i=0}^{m-1} \left(\frac{R_{u_i}}{\delta_x} \right) + O\left(\frac{R}{\delta_x} \right)$$

Proof The first terms comes from Step (1) in which we go through each item once for each uncertain item. Finally we add up all the contributions in the top histogram buckets in Step (2) which gives us the second term in the above expression. □

```
Input
    $u_i, 0 \leq i < m$ : All the uncertain data items
    $\delta_x$, $\delta_p$ : Width of histogram along $x$ and $y$ axis
    $l$, $r$ : The left and right bounds for the histogram
Output
    $H$: The histogram structure for the input data
0. Initialize $H\,(\lfloor R/\delta_x \rfloor + 1, \lfloor 1/\delta_p \rfloor + 1)$ with all histogram bucket heights = 0
1. for $a = u_0, u_1 \ldots, u_{m-1}$ do
    (i) let $x = \lfloor (l_a - l)/\delta_x \rfloor$; $p = 0$
    (ii) while $p < (1 - \delta_p)$
        (a) $p = F_a(l + (x + 1)\delta_x)$
        (b) $H\,(x, \lfloor p/\delta_p \rfloor)$++
        (c) $x$++
2. for $x = 0, 1, \ldots, \lfloor R/\delta_x \rfloor$
    (i) $H(x, \lfloor 1/\delta_p \rfloor)$ += $H(x - 1, \lfloor 1/\delta_p \rfloor)$
3. return $H$
```

Figure 7.4. Algorithm for generating the histogram for unbounded range queries.

7.3.3 General Range Queries

As discussed earlier, a general range query Q is expressed as $Pr(x_1 < a < x_2) > \tau$. This query returns all tuples such that:

$$\begin{aligned} Pr(x_1 < a < x_2) > \tau \;&\Leftrightarrow\; \int_{x_1}^{x_2} f_a(x)dx > \tau \\ &\Leftrightarrow\; F_a(x_2) - F_a(x_1) > \tau \end{aligned}$$

The previous section on unbounded range queries is a special case of the general range query where $x_1 = -\infty$ (or l) or $x_2 = \infty$ (or r).

```
Input
    $x_0, \tau$ : Parameters of a query $Q$
    $H$ : Histogram structure
    $m$ : Total number of uncertain items
    $\delta_x$, $\delta_p$ : Width of histogram along $x$ and $y$ axis
    l, r : The left and right bounds for the histogram
Output
    An estimate (upper-bound) of query selectivity
1. if $x_0 < l$ return 0
2. if $x_0 > r$ return 1
3. $x = \lfloor (x_0 - l)/\delta_x \rfloor$
4. let $S = 0$
5. for $p = \lfloor \tau/\delta_p \rfloor, \ldots, \lfloor 1/\delta_p \rfloor$
    (i) $S = S + H(x, p)$
6. return $(S/m)$
```

Figure 7.5. Algorithm for estimating query selectivity for unbounded range queries.

We can extend the earlier solution to general range queries by adding another dimension to the histogram. In addition to the x-axis and y-axis representing x_2 (end-point of the range query) and the probability threshold τ respectively, we will now have a z-axis representing x_1 (or the beginning of range query).

The theoretical discussion of this selectivity estimation solution is similar to the unbounded case. In place of a 2-D curve, we will now have a 3-D curve for each uncertain item which is given by the function:

$$G_a(x_1, x_2) = \int_{x_1}^{x_2} f_a(x)dx = F_a(x_2) - F_a(x_1) \tag{7.3}$$

The range query Q will now translate to a box query given by $x < x_2$, $y > \tau$ and $z = x_1$. We can now state the following theorem for the 3-D curve:

Theorem 7.3.4 *Each item for which $G_a(x_1, x_2)$ intersects the box defined by query Q is part of the result of query Q. That is, $\forall a$, where the function G_a intersects the box defined by query Q, we have $Pr(x_1 < a < x_2) > \tau$.*

Proof We observe that for any cdf F_a that lies in the box of query Q, we know that $G_a(x_1, x) > \tau$ for some $x < x_2$. This gives us that $G_a(x_1, x_2) > G_a(x_1, x) > \tau$. Using 7.3, we have $P(x_1 < a < x_2) > \tau$. $\square$

Similar to Theorem 7.3.2, we can prove that we can count the total number of items in the result set by counting the total number of intersections of function G_a with the line-segment $x = x_2$, $\tau < y \leq 1$ in the $z = x_1$ plane. The definition and construction of 3-D histogram is similar to the 2-D counterpart and is presented in Figure 7.6. The algorithm for estimating the answer size for a given query $Q(x_1, x_2, \tau)$ is presented in Figure 7.7.

We can apply an optimization similar to the algorithm in Figure 7.4 by modifying only the local histogram area which is affected by an uncertain item and then propagating the effects globally by adding a post-processing step. This optimization helps in bringing down the running time of the algorithm significantly. To achieve this goal we keep three temporary histogram tables H_x, H_z and H_{xz} along with the main histogram structure H. For an uncertain item a, Step 1 adds the contribution of the item to the main histogram H, along with adding the contributions that are to be propagated globally to the temporary histograms. H_z and H_x store the contribution to the bins corresponding to $z = l_a$ and $x = r_a$ respectively, while H_{xz} stores the contribution to the bin corresponding to $z = l_a$ and $x = r_a$. It is easy to see that the local contribution of the item a to H_z needs to be propagated to the plane given by $l_a \leq x < r_a$ and $z < l_a$ as for these values $Pr(z < a < x) = Pr(l_a < a < x)$ (Step 3a). Similarly, H_z needs to be propagated globally to the plane $l_a < z \leq r_a$ and $x > r_a$ as for this plane $Pr(z < a < x) = Pr(z < a < r_a)$ (Step 3b). In a similar fashion, H_{xz}

is propagated to $z < l_a$ and $x > r_a$ (Step 4 and 5). Finally, we add all the temporary histograms to the main histogram to get the final histogram structure (Step 6).

Theorem 7.3.5 *The time complexity of algorithm presented in Figure 7.6 is:*

$$\sum_{i=0}^{m-1}\left(\frac{R_{u_i}^2}{2\delta_x^2}\right) + O\left(\frac{R^2}{\delta_x^2\delta_p}\right)$$

Proof By counting the number of loops. All the steps in Figure 7.6, except for Step 1, touch the cells only constant number of times. The number of loops in Step 1 gives the first summation. □

Equality and Inequality Operators

We now discuss the selectivity estimation for $=_c$ and $\neq_c$ operators. Recall that:

$$\begin{aligned} a =_{c,\tau} v &\Leftrightarrow P(a =_c v) > \tau \\ &\Leftrightarrow F_a(v+c) - F_a(v-c) > \tau \\ &\Leftrightarrow P(v - c < a < v + c) > \tau \end{aligned}$$

Therefore, an equality query boils down to a simple range query. The selectivity estimate of $\neq_c$ is $1 -$ the selectivity estimate for $=_c$ operator. Hence, we can use the techniques discussed in the previous section for estimating selectivity for $=_c$ and $\neq_c$ operators also.

7.3.4 General Range Queries using Slabs

In Section 7.3.3 we discussed how the histogram construction technique can be extended to general range queries. While the accuracy of such an estimate is very good, the initial construction time and space trade-off is quadratic in terms of the range of the input data (R). In this section, we present another technique which has, in general, a lower accuracy than the previous technique but better space-time complexity.

```
Input
        $u_i, 0 \le i < m$ : All the uncertain items
        $\delta_x$, $\delta_p$ : Width of histogram along $x$,$z$ and $y$ axis
        l, r : The left and right bounds for the histogram
Output
        $H$: The histogram structure for the input data
0. Initialize $H, H_x, H_z, H_{xz}$ with all bucket heights $= 0$
1. for $a = u_0, u_1 \ldots, u_{m-1}$ do
        (i) let $x_{min} = \lfloor (l_a - l)/\delta_x \rfloor$, $x_{max} = \lfloor (r_a - l)/\delta_x \rfloor$
        (ii) for $z = x_{min}, \ldots, x_{max}$ and $x = z, \ldots, x_{max}$ do
                (a) $p = G_a(l + z\delta_x, l + (x+1)\delta_x)$
                (b) if $(z = x_{min}) \wedge (x = x_{max})$ $H_{xz}(x, \lfloor p/\delta_p \rfloor, z)$++
                (c) else if $(z = x_{min})$ $H_z(x, \lfloor p/\delta_p \rfloor, z)$++
                (d) else if $(x = x_{max})$ $H_x(x, \lfloor p/\delta_p \rfloor, z)$++
                (e) else $H(x, \lfloor p/\delta_p \rfloor, z)$++
2. let $x_{max} = \lfloor R/\delta_x \rfloor$
3. for $p = 0, \ldots, \lfloor 1/\delta_p \rfloor$
        (a) for $x = 0, \ldots, x_{max}$ and $z = x_{max} - 1, x_{max} - 2, \ldots, 0$ do
                $H_z(x, p, z)$ += $H_z(x, p, z+1)$
        (b) for $z = 0, \ldots, x_{max}$ and $x = 1, 2, \ldots, x_{max}$ do
                $H_x(x, p, z)$+ = $H_x(x-1, p, z)$
4. for $x = 0, \ldots, x_{max}$ and $z = x_{max} - 1, x_{max} - 2, \ldots, 0$ do
        $H_{xz}(x, \lfloor 1/\delta_p \rfloor, z)$ += $H_{xz}(x, \lfloor 1/\delta_p \rfloor, z+1)$
5. for $z = 0, \ldots, x_{max}$ and $x = 1, 2, \ldots, x_{max}$ do
        $H_{xz}(x, \lfloor 1/\delta_p \rfloor, z)$ += $H_{xz}(x-1, \lfloor 1/\delta_p \rfloor, z)$
6. for all $x, z, p$     $H(x, z, p)$ += $H_z(x, p, z) + H_x(x, p, z) + H_{xz}(x, p, z)$
7. return $H$
```

Figure 7.6. Algorithm for generating the histogram for general range queries.

```
Input
        $x_1, x_2, \tau$ : Parameters of a query $Q$
        $H$ : Histogram structure
        $m$ : Total number of uncertain items
        $\delta_x$, $\delta_p$ : Width of histogram bucket along $x$,$z$ and $y$ axis
        l, r : The left and right bounds for the histogram
Output
        An estimate (upper-bound) of query selectivity
1. if $x_1 < l$ $x_1 = l$
2. if $x_2 > r$ $x_2 = r$
3. let $x = \lfloor (x_2 - l)/\delta_x \rfloor, z = \lfloor (x_1 - l)/\delta_x \rfloor$
4. let $S = 0$
5. for $p = \lfloor \tau/\delta_p \rfloor, \ldots, \lfloor 1/\delta_p \rfloor$
        (i) $S = S + H(x, p, z)$
6. return $(S/m)$
```

Figure 7.7. Algorithm for estimating query selectivity for general range queries.

In this algorithm, we partition the entire range of input data into slabs. Similar to histograms, the length of a slab is controlled by the input parameter δ_x. Each slab stores estimates of query selectivity for different values of p. A slab with end-points at $x = x_1, x_2$ stores the selectivity of a bounded range query $Q(x_1, x_2, \tau)$ for different values of τ. Once again, the number of divisions (estimates) along the probability axis is controlled by δ_p. Note that, for a query that spans multiple slabs, we cannot just add the contributions of individual slabs. To solve this problem, we have a hierarchy of slabs. The size of slab at the bottom-most level of this hierarchy is exactly δ_x but as we go up the hierarchy the size increases exponentially until we reach the top-most slab, which encompasses the entire input region. At each level of the hierarchy there are two[1] sets of slabs, one starting at the midpoint of the other, so that we can get better estimates. We call these slabs A and B, respectively.

Formally, we have $\log(R/\delta_x)$ hierarchical levels, with each hierarchical level having two sets of slabs $A(i, j, p)$ and $B(i, j, p)$ where $j \leq \lceil \log_2(R/\delta_x) \rceil$.

Definition 7.3.2 *The slabs $A(i, j, p)$ and $B(i, j, p)$ cover the regions $\mathcal{R}_1 = [l + 2^j i\delta_x, l + 2^j(i+1)\delta_x]$ and $\mathcal{R}_2 = [l + 2^j(i+1/2)\delta_x,\ l + 2^j(i+3/2)\delta_x]$ respectively. The height of the slab $A(i, j, p)$ (or $B(i, j, p)$) is given by the number of uncertain items satisfying the bounded query $\mathcal{R}_1$ (or $\mathcal{R}_2$) with probability between $p\delta_p$ and $(p+1)\delta_p$.*

As mentioned earlier, each of these slabs stores the query answers for different values of query threshold τ. Thus, every $A(i, j)$ or $B(i, j)$ is an array of $\lfloor 1/\delta_p \rfloor$ values. The construction algorithm is presented in Figure 7.8. In Step 1, for each item, we find the slabs that are affected by the item and add the contribution of the item to the corresponding slabs.

Once we have this slab structure, we can get estimates by finding a pair of slabs that contains (over-estimate) and is contained (under-estimate) by the query region. With these estimates, we interpolate the estimates based on the the interval size to get the final estimate. The algorithm for finding the estimate is presented in Figure 7.9. In the algorithm, Step 1 picks j which corresponds to the slab size just smaller than the query. We have two additional functions *pickLB* and *pickUB*, which given the query limits and a level j, returns the slab that is contained inside and contains the query respectively. If these functions can not find any such slab at level j they return *null*. For $j < 0$, these functions simply return a slab with size 0 and all estimates are set to 0. In the case, these functions find more than one slab which satisfy the conditions of UB (LB) they return the one with minimum (maximum) estimate. This is done in order to get a tighter bound on the final estimate. The details of these functions are omitted due to space considerations. Steps 2 and 3 find the slabs and return them. Once we have a slab T_{LB} that bounds the answer from below and a slab

[1] In general, we can have more than two sets of slabs for each level of hierarchy which will further increase the accuracy of this technique.

```
Input
    $u_i, 0 \leq i < m$ : All the uncertain items
    $\delta_x$, $\delta_p$ : Parameters controlling width of divisions
    $l, r$ : The left and right bounds for the input region
Output
    The slab structure for the input data
0. Initialize $A$ and $B$ with all buckets heights = 0
1. for $a = u_0, u_1, \ldots, u_{m-1}$ do
    (i) for $j = 0, 1 \ldots, \lceil \log_2(R/\delta_x) \rceil$ do
        (a) let $x_{min} = \lfloor (l_a - l)/(2^j \delta_x) \rfloor$,
            $x_{max} = \lfloor (r_a - l)/(2^j \delta_x) \rfloor$
        (b) for $x = x_{min} \ldots x_{max}$ do
            (A) let $p = G_a(l + x2^j\delta_x, l + (x+1)2^j\delta_x)$,
            (B) $A(x, j, \lfloor p/\delta_p \rfloor)$++
        (c) let $x_{min} = \lfloor (l_a - (l + 2^{j-1}\delta_x))/(2^j \delta_x) \rfloor$,
            $x_{max} = \lfloor (r_a - (l + 2^{j-1}\delta_x))/(2^j \delta_x) \rfloor$
        (d) for $x = x_{min} \ldots x_{max}$ do
            (A) $p = G_a(l + 2^j(x + 1/2)\delta_x, l + 2^j(x + 3/2)\delta_x)$
            (B) $B(x, j, \lfloor p/\delta_p \rfloor)$++
2. return $A$,$B$
```

Figure 7.8. Algorithm for generating slabs.

T_{UB} that bounds the answer from above, we find the selectivity estimates of T_{LB} and T_{UB} in Step 6 and then finally in Step 7 we linearly interpolate the estimates based on the size of query and size of the two intervals returned. This gives us an estimate of the query result size.

Lemma 8 *For any query Q, the difference between the levels, from which T_{LB} and T_{UB} are picked up, is at most 2. Thus, the space covered by T_{UB} is at most 4 times that of T_{LB}.*

Proof It follows from the cases of Figure 7.9. It remains to show that the *e*lse cases in Step 2(b) and Step 3(a),(b) are always successful in finding a slab. Note that the size of the slab at level j is less than the query interval. So a slab at level j could fit in the query. If this happens with the A slab being contained, then there is a slab at level $j+2$ that surely contains the query. This is because, an A slab at level $j+1$ contains at least one end-point of the query, and hence at level $j+2$, since an A slab and a B slab extend this A slab at level $j+1$ in different directions, at least one of the A slabs at level $j+2$ or B slabs at level $j+2$ will cover the entire interval. If at level j, the query covers a B slab, then it cuts two consecutive A slabs at level j and hence it is covered in either an A slab or a B slab at level $j+1$. If the query does not cover any slab at level j, then it again cuts two consecutive A slabs at level j. This means it is covered by a slab at level $j+1$. Also, it cuts at least one of these A slabs by more than half at the level j. Thus, there is an A slab at level $j-1$ which is contained in the query. □

Theorem 7.3.6 *The time complexity of algorithm presented in Figure 7.8 is:*

$$O\left(\sum_{i=0}^{m-1}\left(\frac{R_{u_i}}{\delta_x}\right)+m\log\left(\frac{R}{\delta_x}\right)\right)$$

Proof The above result directly follows from the following expression which is the total cost of Step 1.

$$\sum_{i=0}^{m-1}\sum_{j=0}^{\log(R/\delta_x)}\left\lceil\frac{R_{u_i}}{2^j\delta_x}\right\rceil$$

□

Similarly, we can also show that the total space overhead is $O\left(R/\delta_x\right)$. Both these results are intuitive if we observe that the total cost/space is asymptotically bounded by number of slabs at the bottom-most level as the number of slabs at higher levels decrease exponentially.

```
Input
    $x_1, x_2, \tau$ : Parameters of a query $Q$
    $A, B$ : Slab structure
    $m$ : Total number of uncertain items
    $\delta_x$, $\delta_p$ : Parameters controlling width of divisions
    $l, r$ : The left and right bounds for the histogram
Output
    An estimate of the query selectivity
1. let $j = \lfloor \log_2((x_2 - x_1)/\delta_x) \rfloor$
2. if (T = pickLB($x_1, x_2, j$)) exists
    (a) $T_{LB} = T$
    (b) if ($T$ = pickUB($x_1, x_2, j+1$)) exists
            $T_{UB} = T$
      else $T_{UB}$ = pickUB($x_1, x_2, j+2$)
3. else
    (a) $T_{LB}$ = pickLB($x_1, x_2, j-1$)
    (b) $T_{UB}$ = pickUB($x_1, x_2, j+1$)
4. let $S_{min} = S_{max} = 0$, $t_1$ = length of $T_{LB}$,
    $t_2$ = length of $T_{UB}$
5. for $p = \lfloor \tau/\delta_p \rfloor, \ldots, \lfloor 1/\delta_p \rfloor$
    (a) $S_{min}$ += $T_{LB}(p)$, $S_{max}$ += $T_{UB}(p)$
6. $S = S_{min} + (S_{max} - S_{min}) \times (x_2 - x_1 - t_1)/(t_2 - t_1)$
7. return $(S/m)$
```

Figure 7.9. Algorithm for estimating query selectivity using slabs.

7.4 Nearest Neighbor Queries

Traditionally, selectivity estimation techniques have been useful for query optimization purpose. Here, we show an example where selectivity estimation can also benefit in indexing and retrieval problems. We consider the k-nearest neighbor (kNN) problem for uncertain data, which has received a lot of interest in recent years [31,56]. There are many different definitions and formulations of nearest neighbors when it comes to uncertain data. We consider here a slightly different variant than the one considered in [31, 56] and show how our selectivity estimation technique can help us design an efficient index for this variant of kNN query.

Let S be the set of tuples with uncertain attribute u. Let query Q consist of a point x, a threshold t, and a number k. Then, for an attribute u, its t-distance from x is the value r_u such that $Pr(x - r_u \leq u \leq x + r_u) = t$. Then, the nearest neighbor of x in S, is the tuple for which r_u is minimum over all values in S. The k-th nearest neighbor of x is the tuple with k-th minimum r_u. The value r_u of the k-th nearest neighbor is also called kNN radius $r(x, t, k)$ for the point x with threshold t. The query $Q = (x, t, k)$ when applied over S returns the set $Q(S)$ of k tuples which are nearest neighbors.

Note that if we know the value $r = r(x, t, k)$ before-hand then such a set $Q(S)$ can be obtained by probabilistic threshold range query (PTRQ) $([x - r, x + r], t)$. If we do not we could end-up in the case where we would need to examine a lot more candidates in order to make sure of which are the nearest k candidates. Note that the brute-force approach could possibly examine all the tuples. Thus, finding such an r is critical.

With our estimation techniques, we can quickly estimate such a radius r. Note that for two values $r_1 < r_2$, the $count(PTRQ([x - r_1, x + r_1], t)) < count(PTRQ([x - r_2, x + r_2], t))$. This monotonic behavior of *count* with respect to the radius r allows us to do a binary search to find the value of suitable r. We search for the value of r such that the estimate of $count(PTRQ([x - r, x + r], t))$ is equal to k. Notice that since these are estimates they may not be strictly monotonic and binary search can be affected by such an inversion. If we find such inversion for some range in our procedure, we stop the process and estimate the higher value of the range as r.

7.5 Experimental Evaluation

We have implemented our statistics collection and selectivity estimation algorithms in *Orion*. To efficiently evaluate the queries discussed in this chapter, Orion uses the *probabilistic threshold index* (PTI). This system not only allows us to validate the accuracy of our methods in a realistic runtime environment, it also gives additional insight into the overall effect our techniques have on query optimization in an industrial-strength DBMS.

7.5.1 Implementation

PostgreSQL measures the cost of query plans in disk page fetches (for simplicity, all CPU efforts are converted into disk I/Os). The optimizer generally estimates the cost of query plans by calculating the overall selectivity and multiplying it against the estimated cardinality. In the common case of multiple predicates, individual selectivities are multiplied together, except for range queries where the dependence between the lower and upper bounds is simple to evaluate.

Virtually every numeric data type in PostgreSQL shares the same source code for cost estimation. Using this code base, we have built our implementation of the algorithms in Figures 7.4, 7.6, 7.5, and 7.7. Using the elegant framework PostgreSQL provides for new data management techniques, our implementation extends the functionality of Orion's UNCERTAIN data type by registering the optional callbacks for collecting statistics and estimating selectivity.

7.5.2 Methodology

To ensure correctness, we ran each experiment on a variety of queries and datasets, and then averaged the results. After populating the database with each test dataset, we first used `VACUUM ANALYZE` to generate the statistics in advance. The following experiments were conducted on a 1.6 GHz Pentium CPU with 512 MB RAM, running

Linux 2.6.17, PostgreSQL 8.1.5, and Orion 0.1. Note that most of the resulting plots show the *relative error* of the selectivity estimates, i.e. the goal is to be as close to 0% as possible.

Synthetic Datasets

Each dataset consists of random "sensor readings," using a schema `Readings (rid, value)`. Without loss of generality, the uncertain values (i.e. reported from the sensors) are floating point numbers ranging from 0 to 1000, and the pdf for each uncertain value is a uniform distribution. The interval sizes are distributed normally, with midpoints distributed uniformly. We refer to our three main datasets as `Data-5`, `Data-50`, and `Data-100`; the numbers correspond to the average width of the uncertain value intervals.

Table 7.2 summarizes the control variables for the subsequent experiments. In particular, we show that our algorithms perform well without regard to dataset cardinality, and are reasonably robust to query selectivity and probabilistic threshold. In addition, we demonstrate the effect of increased precision as a trade-off between construction time and space versus the resulting accuracy of the selectivity estimates.

Table 7.2
Summary of Control Variables.

Variable	**Default Value**
Cardinality	250,000
Selectivity	2.5 %
Threshold	50 %
Precision	70 bins

Example Query Plan

To illustrate the impact that correct estimates have on query optimization, we present the following example output from PostgreSQL. When no selectivity estimation function is available for a given predicate, PostgreSQL simply returns the default value of 1/3 for estimating unbounded range queries, and 0.005 for general range queries. In practice this estimate favors the use of unclustered indexes, such as PTI [30], to improve I/O performance:

```
SELECT * FROM Readings WHERE value < 750;
-----------------------------------------
Bitmap Heap Scan on Readings
 (cost=742.33..4075.67 rows=66667 width=36)
 (actual=20379.348..20824.652 rows=153037)
 Recheck Cond: (value < 750::real)
->  Bitmap Index Scan on pti_value
 (cost=0.00..742.33 rows=66667 width=0)
 (actual=20378.677..20378.677 rows=153K)
 Index Cond: (value < 750::real)
```

With accurate estimates, the optimizer makes the correct decision, namely not to use the available PTI index:

```
(same query as before, but using our algorithms)
------------------------------------------------
Seq Scan on Readings
  (cost=0.00..5000.00 rows=164333 width=35)
  (actual=83.841..15545.401 rows=153037)
  Filter: (value < 750::real)
```

As shown in this example, accurate selectivity estimation saves the system thousands of disk fetches (i.e. 15545 total cost instead of 20825). In general, incorrect estimates may result in much higher losses of efficiency.

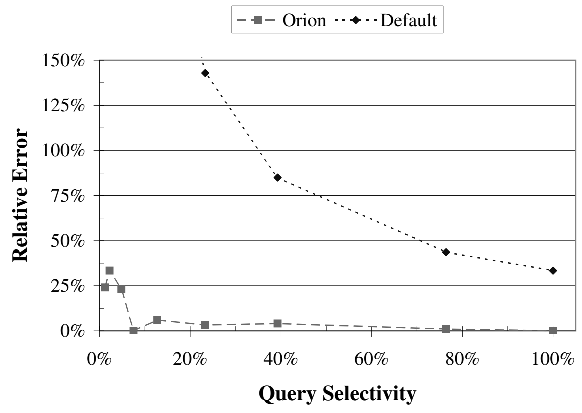

Figure 7.10. Selectivities (2D).

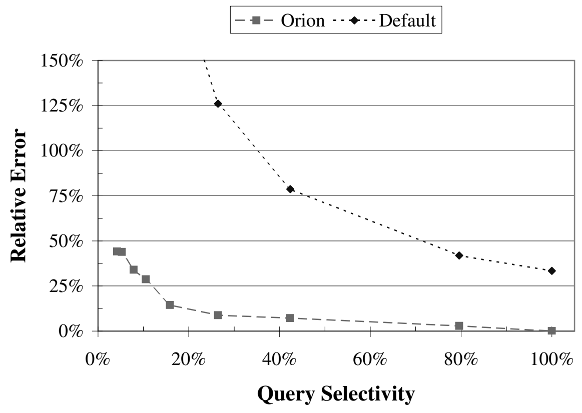

Figure 7.11. Selectivities (3D).

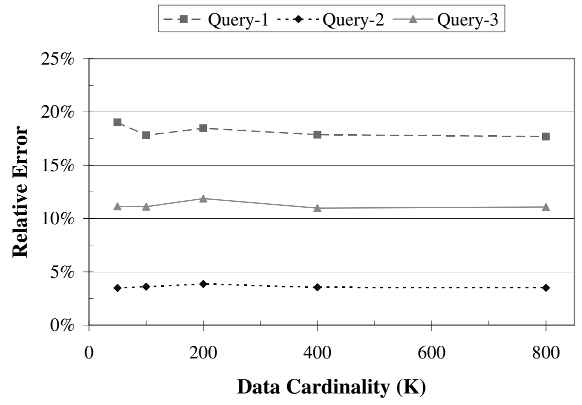

Figure 7.12. Cardinalities (2D).

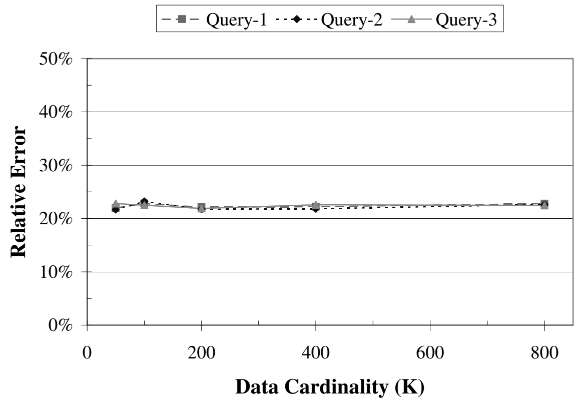

Figure 7.13. Cardinalities (3D).

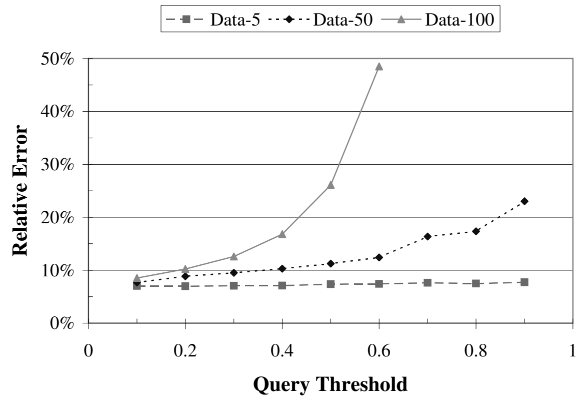

Figure 7.14. Thresholds (2D).

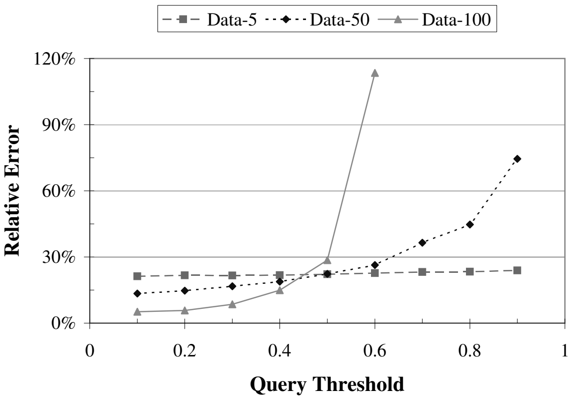

Figure 7.15. Thresholds (3D).

7.5.3 Results

We now evaluate the accuracy and performance of our cost estimation techniques for unbounded range queries using the 2D histogram structure introduced in Section

7.3.2 (see Figure 7.5), and general range queries using the 3D histogram discussed in Section 7.3.3 (see Figure 7.7).

Accuracy at Varying Selectivities

The first experiment verifies the accuracy of our algorithms, regardless of query selectivity. Figures 7.10 and 7.11 summarize the results using all three synthetic datasets. For clarity, we have only plotted one of them. The x-axis shows the selectivity of the query which was varied from high (1%) to low (100%). The y-axis shows the accuracy of the estimation as a percentage relative to the size of the exact result. Our algorithm significantly outperforms the baseline PostgreSQL estimate. As expected, high selectivity has a slight effect on the accuracy of our methods.

Accuracy at Varying Cardinalities

The next experiment studies the overall scalability of our algorithms, namely the impact of the size of the relation on the accuracy of the estimations. Figures 7.12 and 7.13 show the results for three representative queries. The x-axis shows the size of the table in number of tuples which was varied from 50,000 to 800,000. The results show that our approach is unaffected by the size of the dataset. This is in sharp contrast to the baseline PostgreSQL estimator (not shown) which is much more sensitive to the dataset size, particularly for smaller datasets.

Accuracy at Varying Thresholds

Figures 7.14 and 7.15 show the impact of query threshold on the accuracy of the estimates. The x-axis shows the threshold probability and the y-axis shows the relative accuracy with respect to the correct answer size. Once again, we observe that our algorithm is much more robust than the baseline PostgreSQL estimator (not shown) that simply returns a constant selectivity. Our implementation shows slightly

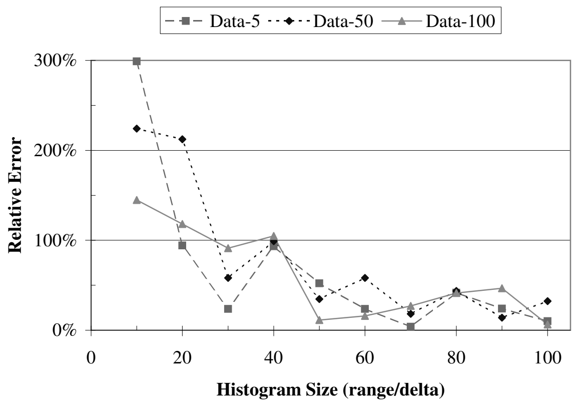

Figure 7.16. Precision (2D).

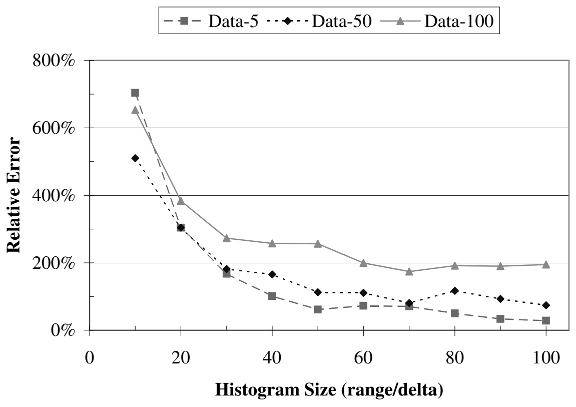

Figure 7.17. Precision (3D).

better accuracy for smaller thresholds, in part because larger thresholds result in additional tuples becoming part of the query answer, leading to overestimates. We can see that for highly selective queries, our algorithm is significantly better that the baseline and thus it is more likely to lead the optimizer into choosing a much more efficient plan.

Accuracy at Varying Precisions

Next we show the relationship between the size of the histograms and the resulting accuracy. Figures 7.16 and 7.17 summarize the results for each dataset. The x-axis shows the number of histogram buckets in each dimension, which was varied from 10 to 100. Clearly, both algorithms perform better with a more detailed histogram. Our algorithm outperforms the baseline for smaller histograms. As expected, we see that after a certain amount (i.e. 70, for these datasets and queries), larger histograms do not provide significant increase in accuracy.

Runtime Performance Overhead

The final set of experiments study the runtime performance of constructing the statistics and estimating the selectivity of a query. As expected, the estimation times are constant and almost negligible (on the order of 15 ms). The histogram construction times scale linearly with respect to data cardinality, and grow a little more than linear as the requested number of buckets increases. For the bulk of our experiments, histogram construction only amounted to several hundred milliseconds.

7.6 Chapter Summary

In this chapter, we developed algorithms for computing selectivity estimates of probabilistic queries over uncertain data. The estimation techniques can be applied both to tuple uncertainty and attribute uncertainty models. These techniques were implemented in Orion and found to provide accurate estimates for uncertain data. The algorithms presented can be further improved by combining them with standard cost estimation techniques such as equi-depth binning and sampling. We identified necessary and sufficient conditions and based on these conditions developed efficient method to process the tuples. We showed both theoretically and empirically that our histogram construction algorithms are fast. The experiments show that they give very accurate estimation especially for less selective queries. For more selective queries, the accuracy is not quite as good, but is still much better than the baseline estimator. Our estimation techniques are applicable not only in query optimization – they can also be used for other applications such as k-nearest neighbor queries. To the best of our knowledge, there is no prior work that deals with selectivity estimation over uncertain probabilistic data.

The results of this chapter are presented in [57]. In the next part of this thesis, we discuss a model for uncertainty management in databases which generalizes both tuple- and attribute-uncertainty models.

8 UNIFIED MODEL FOR PROBABILISTIC ATTRIBUTES AND TUPLES

This chapter presents a model for handling arbitrary probabilistic uncertain data (both discrete and continuous) natively at the database level. The presented approach leads to a natural and efficient representation for probabilistic data. We develop a model that is consistent with possible worlds semantics and closed under basic relational operators. This is the first model that accurately and efficiently handles both continuous and discrete uncertainty.

8.1 Introduction

In this chapter, we focus on probabilistic modeling. Recent work on the problem of handling uncertain data using probabilistic relational modeling can be divided into two main groups. One deals with modeling and the other with efficient execution of queries. Work on query processing over probabilistic data has assumed a simple model – a single (continuous or discrete) attribute that takes on probabilistic values [30–33, 55, 58]. Most of this work is focused on developing index structures for efficient query evaluation over probability distribution (or density) functions (pdf). While this work addresses specific queries (e.g. Range [45], nearest-neighbors [31]), it lacks a comprehensive model to handle complex database queries consisting of selects, projects and joins in a consistent manner. Most of the work is also focused on single table queries.

Recently proposed models for probabilistic relational data deal with the representation and management of *tuple uncertainty* (with the exception of [30]). These models are naturally well-suited for applications with categorical uncertainty. Under tuple uncertainty, the presence of a tuple in a relation is probabilistic, and multiple tuples can have constraints such as mutual exclusion among them. The recently proposed models [10, 17, 19] generalize most of the earlier models for probabilistic relational data. In contrast, *attribute uncertainty* models [12, 30] consider that a tuple is definitely part of the database, but one or more of its attributes is (are) not known with certainty. The model in [30] allows an uncertain value to take on a continuous ranges of values, but all other work has been focused on the case of discrete uncertainty (i.e. an enumerated list of alternative values with associated probabilities). Continuous uncertainty models easily capture the case of discrete uncertainty. Discrete uncertainty models can handle continuous uncertainty by sampling the continuous pdf, but are forced to trade-off accuracy (lots of samples) or efficiency (fewer samples).

This chapter presents a new model for representing probabilistic data that handles both continuous and discrete domains and allows uncertainty at the attribute and tuple level. To the best of our knowledge, this is the first model that handles continuous pdfs and is closed under possible worlds semantics (Section 8.1.1). The model can handle arbitrary correlations among attributes of a given tuple, and across tuples. Although this model is motivated by attribute uncertainty, it can directly handle tuple uncertainty, and thus is more general. The underlying representation for arbitrarily correlated uncertain data in our model is based upon multi-dimensional pdf attributes. Our approach results in a more natural representation for uncertain data primarily due to the fact that our chosen data representation better matches how uncertainty is modeled in applications. A second advantage of our model is its space efficient representation of uncertain data. This efficiency results in improved query result accuracy and lower processing time.

As discussed in Chapter 1, there are many real-world examples with continuous data where the uncertainty is naturally represented by continuous pdfs. (e.g., sensors, spatio-temporal databases, flight tracking, healthcare monitoring, financial analysis). Table 8.1 shows an example of values returned by the sensors. (*Gaus* represents

Table 8.1
Example: Sensor Database.

Sensor ID	**Location**
1	Gaus(20,5)
2	Gaus(25,4)
3	Gaus(13,1)

a Gaussian distribution followed by the parameters of the distribution – mean and variance).

Now consider the case where we use tuple uncertainty (i.e., discrete uncertainty) to model the sensor database in Table 8.1. Current tuple uncertainty models will be forced to make a discrete approximation of the pdf as they only support discrete uncertain data. This approach has a number of weaknesses. Firstly, such a representation is not efficient as we have to repeat certain attribute(s) (e.g., sensor id) along with each value instance of uncertain attribute(s). Secondly, either we have to sample many points (not practical) or sacrifice a great deal of accuracy (not desirable). On the other hand, if we use the symbolic form of a Gaussian distribution, obviously the answers will be more accurate as we are avoiding approximations. Furthermore, as we will see later, the usual database operations can be evaluated on symbolic pdfs in a more efficient manner. Note that this requires built-in support for symbolic pdfs (e.g., Gaussian) in the database. Our model provides this support, and for non-standard distributions, we support a *generic* pdf represented by histograms (*Hist*). Histograms give us an approximation for continuous pdfs, but this approximation is still more accurate than a discrete approximation. This issue is further explored in the experimental section.

In the previous example, a discrete sampling leads to loss in accuracy, but in many applications using discrete pdfs in place of continuous pdfs may be downright incorrect. Consider the example of location privacy where uncertainty can be used to "blur" the exact location of points to enhance their privacy. For these applications, a continuous uncertainty model will be better suited than a discrete uncertainty model. Joins over uncertain data would be another example – if we use discrete sampling to approximate a continuous distribution, we would miss many potential results. Another issue to consider is efficient query evaluation – using previous discrete models to model continuous uncertainty would be very inefficient (or infeasible) as the number of possible worlds are infinite in this case.

In addition, even in situations where the base uncertain data is discrete, some queries (e.g. aggregates) can produce results that are very expensive to represent using discrete pdfs. The main reason is that the resulting uncertain attribute can have an exponential number of possible values. In such cases, one can save space as well as time by approximating with a continuous pdf. This is exactly what our model proposes.

While our model is tailored towards representing continuous distributions, it is general enough to be used for modeling discrete uncertainty as well.

In summary, the salient features of our model are:

1. It handle both continuous and discrete uncertainty (with arbitrary correlations) natively at the database level, and is consistent and closed under possible worlds semantics.

2. The first model for uncertain data that can accurately handle continuous pdfs.

3. The pdf approach leads to a more natural and efficient representation and implementation than a tuple uncertainty based approach.

8.1.1 Possible Worlds Semantics

The definition of relational operators for this model is based upon the Possible Worlds Semantics (PWS) [59] that has been commonly used for other work on uncertain databases. Under these semantics, a probabilistic relation is defined over a set of probabilistic events. Depending upon the outcome of each of these events, a possible world is defined. Thus given a probabilistic relation, we get a set of possible worlds corresponding to all possible combinations of the outcomes of the events in the relation. Figure 8.1 shows a graphical view of the possible worlds semantics. Given a probabilistic database and query θ to be evaluated over this database, conceptually we first *expand* the database to produce the set of all possible worlds. The query is then executed on each possible world. The resulting probabilistic database is defined as the database obtained by *collapsing* the possible worlds in which the query is satisfied.

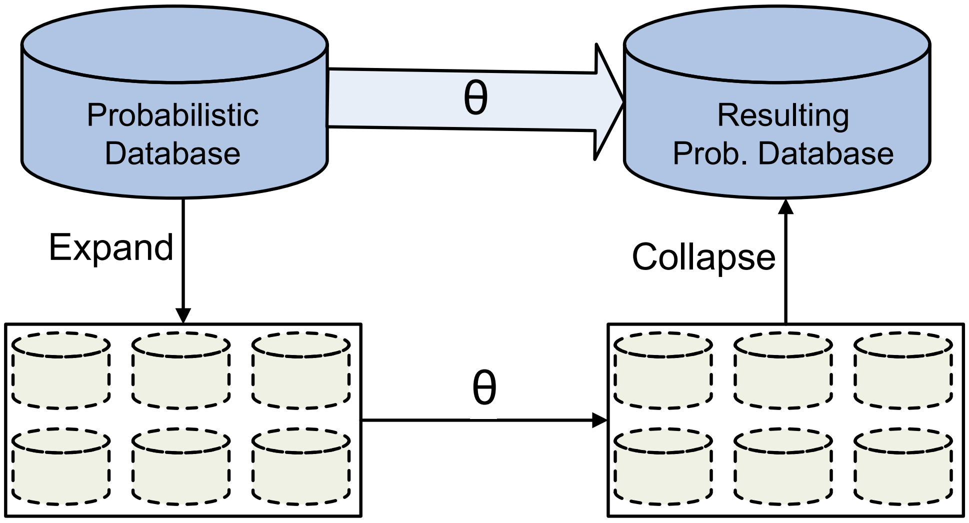

Figure 8.1. Possible Worlds Semantics.

Consider a database table with uncertain attributes a and b, as shown in Table 8.2. It consists of two probabilistic tuples. The first tuple represents a total of 4 possibilities: (i.e. $\{0,1\},\{0,2\},\{1,1\},\{1,2\}$) and a single (certain) value for the second tuple. The corresponding set of possible worlds are shown in Table 8.3 along with the associated probabilities for each world. The semantics of a query over this uncertain relation are defined as follows. The query is executed over each possible world (which has no uncertainty) to yield a set of possible results along with the probability of each result. The probability values of worlds that yield the same result are aggregated to yield the probability of that result for the overall query over the uncertain relation. Consider a selection query with predicate $a < b$, over the relation in Table 8.2. Conceptually, this query is evaluated over each possible world. The probability that a tuple satisfies the query criterion is equal to the sum of the probabilities of the possible worlds in which the tuple satisfies the query. In practice, the number of possible worlds can be very large (even infinite for continuous uncertainty). The goal of a practical model is to avoid enumerating all possible worlds while ensuring that the results are consistent with PWS. Section 8.3.4 shows how our model handles this particular example.

Table 8.2
Example of Probabilistic Table.

a	**Pr(a)**	**b**	**Pr(b)**
0	0.1	1	0.6
1	0.9	2	0.4
7	1.0	3	1.0

Table 8.3
Possible Worlds.

Possible Worlds		Probability
0	1	0.06
7	3	
0	2	0.04
7	3	
1	1	0.54
7	3	
1	2	0.36
7	3	

8.2 Model

In this section, we formally define our model for representing and querying a database with probabilistic data. We allow two kinds of attributes – *uncertain* (or pdf attributes) and *certain* (or precise) attributes. The model represents a set of database tables $\mathbf{T}$, with a set of probabilistic schemas $\{(\Sigma_T, \Delta_T) : \forall T \in \mathbf{T}\}$ and a history Λ for each dependent set of attributes in $\mathbf{T}$. A database table T is defined by a *probabilistic schema* (Σ_T, Δ_T) consisting of a *schema* (Σ_T) and *dependency information* (Δ_T). The schema Σ_T is similar to the regular relational schema and specifies the names and data types of the table attributes (both certain and uncertain). The dependency information Δ_T identifies the attributes in T that are jointly distributed (i.e., correlated). The uncertain attributes are represented by pdfs (or joint pdfs) in the table. In addition to pdfs, for each dependent group of uncertain attributes we store its *history* Λ. We will now describe each of these concepts in detail.

8.2.1 Uncertain Datatypes and Correlations

There are two major kinds of uncertain data types that our model supports – discrete and continuous. These data types are represented using their pdfs. The uncertainty model in many real applications can be expressed using standard distributions. Our model has built in support for many commonly used continuous (e.g., Gaussian, Uniform, Poisson) and discrete (e.g., Binomial, Bernoulli) distributions. These distributions are stored symbolically in the database. The major advantage of using these standard distributions is efficient representation and processing. When the underlying data distribution cannot be represented using the standard distributions we revert to *generic* distributions – Histogram and Discrete sampling. The histogram distribution consists of buckets over the data domain, along with the probability density in each bucket. The discrete sampling simply consists of multiple value-probability pairs. The bin size (or number of sampling points) is an important parameter that decides the trade-off between accuracy and efficiency.

The simple pdf distributions discussed above can be used to represent one dimensional pdfs. But in many cases, there are *intra-tuple* correlations present within the attributes. For example, in a location tracking application, the uncertainty between the x- and y-coordinates of an object is correlated. These more complex distributions are supported in our model using joint probability distributions *across* attributes.

For example, to represent the 2-D uncertainty in case of moving objects we represent the uncertainty by creating two uncertain attributes x and y which specify the x- and y-coordinates of the object, respectively. Instead of specifying two *independent pdfs* over x and y, we have a single *joint pdf* over these two attributes.

The information about intra-tuple dependencies is captured by the schema dependency information Δ_T. Δ_T is a partition of all the uncertain attributes present in the table T. It consists of multiple sets of attributes that are correlated within a tuple. These sets are called *dependency sets*. It also contains *singleton sets* containing attributes that are uncertain but are not dependent on any other attributes. The attributes not listed in Δ_T are assumed to be certain.

To illustrate, let us consider a table T with schema $\Sigma_T = (a_1{:}d_1, a_2{:}d_2, a_3{:}d_3, a_4{:}d_4)$, where d_i represents the data type of attribute a_i. If all the attributes in the table are certain, $\Delta_T = \phi$. On the other hand, if a_1, a_2 and a_3 are uncertain and a_1, a_2 are correlated, this information is represented by defining the dependency information as $\Delta_T = \{a_1, a_2\}, \{a_3\}$. For the example presented in Table 8.1, $\Sigma_T = \{id : int, x : real\}$ and $\Delta_T = \{x\}$ (x represents the 1-D location). To model the location as a jointly distributed 2-D attribute, $\Sigma_T = \{id : int, x : real, y : real\}$ and $\Delta_T = \{x, y\}$.

Consider the special case when all the attributes in a table T are jointly distributed (i.e. $\Delta_T = \{\Sigma_T\}$). This extreme case captures tuple uncertainty as the complete value of the tuple is uncertain. The joint pdf over the attributes implicitly represents a group of dependent tuples. In addition, we can define tuples which are continuous and thus an infinite number of alternatives are possible for each tuple. This representation is more powerful that the tuple uncertainty models in which each tuple can only have a finite number of alternatives.

We allow the dependency information Δ_T to contain *phantom attributes* which are not present in Σ_T. These extra attributes and their corresponding joint distribution are needed for ensuring that the correlation information of the attributes that are projected out is not lost during projections (See Section 8.3.3 for more information). However, only the attributes in Σ_T are visible to the user.

Definition 8.2.1 *A probabilistic tuple t of table $T(\Sigma_T, \Delta_T)$ is represented by values $t.a_j$ for all certain attributes a_j and pdf $f_t(S_i)$ for all sets of uncertain attributes $t.S_i \in \Delta_T$.*

To be precise, let us define $X^t_{S_i}$ to be the random variable for an attribute set $t.S_i$. Thus, $f_t(S_i)$ returns a pdf function that is defined over $X^t_{S_i}$. That is, $f_t : S_i \rightarrow f(X^t_{S_i})$. In the rest of this chapter, whenever we refer to $f_t(S_i)$, it is understood that we are referring to the underlying distribution $f(X^t_{S_i})$.

Given $f_t(S)$, we can easily calculate $f_t(C)$, where $C \subset S$ by marginalizing the pdf of S.

$$f_t(S) = \int_{S-C} f_t(C) \tag{8.1}$$

In case of discrete distributions, the integral is replaced by sum.

8.2.2 Partial PDFs

Table 8.4
Example: Missing Attributes Values vs Missing Tuples.

a	**b**	**c**	**Pr(b, c)**
1	2	3	0.8
	NULL	*NULL*	0.2
2	4	7	0.2
	4.1	3.7	0.6

In traditional databases, NULL is used to represent unknown or missing data. We also use NULL values in our model to signify *missing attribute values*. However, there is another way of representing missing data. The semantics of these two approaches differ from each other. To illustrate this point, let us consider the example presented in Table 8.4. The first tuple has missing (unknown) values for attribute b and c.

However, the presence of the tuple itself is certain as the probability $Pr(b, c)$ adds up to 1. The other approach for representing missing data uses a closed world assumption to represent unknown information with *partial pdfs*. The probability that the second tuple exists in the table is 0.8 ($=\sum Pr(b, c)$) and thus with 0.2 probability the tuple does not exist in the table. Although both these approaches signify missing data their probabilistic interpretations are quite different.

The usual definition of a pdf requires that it sums up (or integrates) to 1. We remove this restriction in our model in order to represent *missing tuples* with partial pdfs. The support for partial pdfs is crucial in our model to ensure that database operations such as selection are consistent with PWS. A partial pdf is a pdf where only the events associated with the existence of the tuple are explicitly represented. If the joint pdf of a tuple (defined formally in Section 8.3.2) sums to x, then $1 - x$ is the probability that the tuple does not exist, under a closed world assumption. In this chapter, we use the terms pdf and partial pdf interchangeably.

8.2.3 History

As discussed in the previous section, we allow multiple attributes to be jointly distributed in our model. This flexibility makes the model very powerful in terms of data representation, by allowing *intra-tuple dependencies* (i.e. correlation between attributes). But for the model to be closed and correct under the usual database operations, we need to handle *inter-tuple dependencies* as well. History captures dependencies among attribute sets as a result of prior database operations. It is used to ensure that the results of subsequent database operations are consistent with PWS. This is described in more detail in Section 8.3. A similar concept is used in many tuple uncertainty models to track correlations between tuples. [10] uses lineage and [15] uses factor tables to capture such dependencies. As we are interested in capturing historical dependencies between attributes of tuples, our concept of dependencies is different from this related work, which capture these dependencies on a per tuple basis.

We maintain the history of uncertain attributes by storing the top-level *ancestors* of each dependency set in a tuple. The function Λ maps each pdf $t.S$ of a tuple t, to a set of pdfs that are its ancestors.

Definition 8.2.2 *For a newly inserted tuple t in table T, $\Lambda(t.S) = t.S$, $\forall S \in \Delta_T$. If a new pdf $t'.S'$ is* derived *from pdfs $t.S_i$ via a database operation, then $\Lambda(t'.S') = \bigcup_i \Lambda(t.S_i)$.*

In other words, the ancestors are the base pdfs which are inserted in the database by the user. We assume that the base tuples are independent. All the *derived* attributes point back to the base pdfs from which they are derived.

Definition 8.2.3 *If $\Lambda(t.S_1) \cap \Lambda(t.S_2) \neq \phi$, then the nodes $t.S_1$ and $t.S_2$ are said to be* historically dependent.

Note that the deletion of a base tuple will cause dependency sets of its derived tuples to lose their ancestor information. Thus, while deleting a tuple from the base table, we first check if any other tuple in the database is referencing any dependency set within the tuple. If there is a reference, we delete the tuple but keep the dependency set and its pdf as a *phantom* node until its reference count falls to zero. Definition 8.2.2 assumes that the base tuples are historically independent. This is not limiting since a historical dependency between attribute sets of a base table, can be captured by creating a *phantom* ancestor and pointing the dependent attribute sets to this common phantom ancestor.

8.3 Probabilistic Operations

We begin by defining some basic operations on pdfs that underly the implementation of the usual database operations for our model. These operators are not directly accessible by users. One of the strengths of our model is that correctness with respect to PWS is achieved by manipulating the pdfs. Next, we present the usual relational operations under our model. The section concludes with a discussion of new operators that directly operate on the pdfs and are available to users as extensions to SQL.

8.3.1 Preliminaries

Here we describe some basic operations that are needed to define the usual relational database operations. These are basic tools which allow us to handle pdfs across the fundamental database operations of select, project and cartesian product. Corresponding to these, we describe the operations floor, collapse and product on pdfs. These operations are consistent with PWS interpretation of pdfs. Note that we distinguish between attribute set of a pdf and attribute value set pdf which is all possible combination of values for the attributes in the attribute set.

`marginalize`(f, A): Given a pdf f over attributes A_f, and a subset of attributes $A \subseteq A_f$: the operation produces the pdf function f' over attributes A. This is done by marginalizing the distribution f, i.e. $f' = \int_{A_f - A} f$. For discrete distributions, the integral is replaced by sum. It is easy to show the consistency wrt PWS because the probability of an event is the sum of probabilities of all the possible worlds in which the event occurs.

As an example, the results of marginalizing the pdf shown in Figure 8.2(c) over x and y is shown in Figure 8.2(a) and Figure 8.2(b), respectively.

`floor`(f, F): Given a pdf f, on a domain D and given a subset $F' \subseteq D$, operation `floor`(f, F) produces a new pdf f' such that values of $f'(x) = 0$ whenever $x \in F$ and $f'(x) = f(x)$ otherwise. This `floor` operation corresponds to a selection predicate. The values in F are those which do not pass the selection criteria and hence do not exist in the resulting pdf. Going by the PWS, this means that in the possible world where x takes the value in F, this tuple does not meet the selection criteria and hence it does not exist. Multiple `floor` operations can be successively applied over a pdf in any order and the result would be $\texttt{floor}(f, F_1 \cup ...F_k)$ regardless of the order in which they are applied.

The application of `floor` on a symbolic distribution (e.g. Gaus) will, in general, result in a non-standard partial pdf. This partial pdf could be potentially captured by a histogram representation. But, we can optimize the floor operation (and subsequent operations) significantly, if we store *symbolic floors* to represent the flooring operation along with the original (symbolic) distribution. Our model has built-in support for simple symbolic floors which result from some common selection predicates. To illustrate, if the distribution of an attribute x is given by Gaus(5,1) and we apply the selection predicate $x < 5$, the resulting pdf will be floored when $x \geq 5$ (and its value is given by Gaus(5,1) when $x < 5$). This resulting distribution is represented as [Gaus(5,1), Floor$\{[5, \infty]\}$] in our implementation.[1]

$\texttt{product}(f_1, f_2)$: Given two pdfs f_1 and f_2 over attribute value sets S_1 and S_2 (in a given tuple t) respectively, the operation `product` gives their joint pdf f (over $S' = S_1 \cup S_2$). We have to consider the following two cases:

f_1 and f_2 are historically independent: In this case, $f(x) = f_1(x_1)f_2(x_2)$ where $x \in S_1 \times S_2$ and $x = (x_1, x_2)$. To illustrate, assuming the pdfs shown in Figure 8.2(a), (b) are historically independent, the result of performing the product operation is shown in Figure 8.2(c).

f_1 and f_2 are historically dependent: Let $t_j.N_j, 1 \leq j \leq m$ be the common ancestors of $t.S_1$ and $t.S_2$ (i.e. $t_j.N_j \in \Lambda(t.S_1) \cap \Lambda(t.S_2)$). Each $t_j.N_j$ represents the distribution of an attribute set (N_j) of a given tuple (t_j). Thus N_j denotes the set of attributes in $t_j.N_j$. We define $C_j = N_j \cap S'$ and $D_i = S_i - \bigcup C_j$, $i = 1$ or 2. Thus C_j is the set of attributes that the ancestor $t_j.N_j$ shares with either S_1 or S_2 . D_1 (D_2) is the set of attributes in S_1 (S_2) that are not shared with any common ancestor. Let

[1]Similar implementation optimizations are possible for other operations presented in this chapter. We skip their discussion in this chapter due to space limitation.

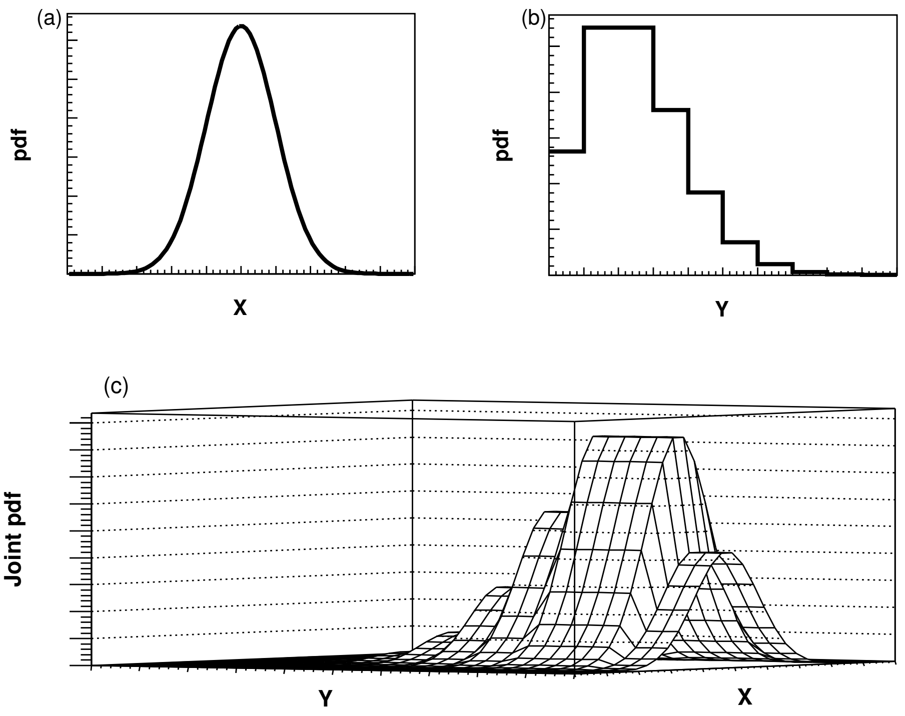

Figure 8.2. Example of product operation.

X_S^t be the random variable for an attribute set $t.S$. Let x_S^t be an instance of X_S^t. With these notations, the joint pdf of resulting set $t.S'$ is:

$$f(x_{S'}^t) = \begin{cases} 0, & \text{if } f(x_{S_1}^t) \text{ or } f(x_{S_2}^t) = 0 \\ f(x_{D_1}^t) f(x_{D_2}^t) \prod_{j=1}^{m} f(x_{C_j}^{t_j}), & \text{otherwise} \end{cases}$$

where, $x_{S'}^t \in X_{D_1}^t \times X_{D_2}^t \times X_{C_1}^{t_1} \times X_{C_1}^{t_1} \ldots \times X_{C_m}^{t_m} \times X_{C_m}^{t_m}$

In other words, we first find the group of attribute sets (D_1, D_2 and $C_j, \forall j$) that are independent of each other. We can multiply the distributions of these nodes as they are independent. But, that would ignore any `floors` that were applied during database operations from ancestor nodes $t_j.N_j$ to $t.S_1$ or $t.S_2$. One potential solution is to keep track of all the operations and re-apply them[2] but we observe that we can

[2]This method, though correct, is very inefficient and will not scale with database size and number of operations.

infer the final floors from the distributions of $t.S_1$ and $t.S_2$. The regions where they were floored are the regions whose corresponding possible worlds did not "survive" the selection conditions. Thus, we propagate the floors of $t.S_1$ and $t.S_2$ to the joint distribution. This operator is used for defining selection and is further discussed in Section 8.3.4. Note that this operator is associative and hence can be used over more than two pdfs as well.

8.3.2 Tuple Distribution

We now define the joint probability distribution over a probabilistic tuple. According to Definition 8.2.1, a probabilistic tuple consists of certain attributes and pdfs for all the attributes appearing together in the dependency set. We can consider the certain attributes as a pdf consisting of a single value with probability 1. Using this notion, we define the joint distribution of a tuple as the `product` of all the certain attributes and the dependency sets in that tuple. As discussed in Section 8.2.2, if the joint pdf of the tuple sums to x (less than 1), then with probability $1 - x$ the tuple will not exist in the database. Thus, in order for a tuple to definitely exist in the database, it should not contain any attribute sets with partial pdfs.

8.3.3 Projections

Given a table T, we define $R = \Pi_A(T)$ as the table which contains a tuple t' corresponding to *each* tuple $t \in R$ $(t \rightarrow t')$, such that the resulting schema $\Sigma_R = A$. The new dependency information Δ_R can contain some of the attributes that are projected away. These attributes and their corresponding distributes are kept to ensure that we do not loose any floors associated with the projected out attributes. $\forall S_i \in \Delta_T$, where $S_i \cap A \neq \phi$ or $\int f_t(S_i) \neq 1$, we keep $S_i \in \Delta_R$. A number of optimizations are possible to reduce the number of extra attributes that are kept in Δ_R. For example, instead of the complete set S_i, we can keep a subset S'_i such that for each tuple, S'_i functionally determine S_i.

The history of the new sets is updated to history of sets from which they are derived i.e. $\forall t' \in R$ and $\forall S_k \in \Delta_R$ where $t \rightarrow t'$ and $S_k \subseteq S_i$ ($S_i \in \Delta_T$), we have $\Lambda(t'.S_k) = \Lambda(t.S_i)$.

Similar to other models for uncertain data, we do not address the issue of duplicate elimination in projections in this chapter. This is because the concept of duplicate elimination for probabilistic data in general leads to complex historical dependencies. As part of our ongoing work, we are extending our model to address duplicate elimination.

8.3.4 Selections

Given a table T with attributes Σ_T and a boolean predicate $\Theta(A)$ defined over a subset of attributes A of table T, the result of the selection operator is $R = \sigma_{\Theta(A)}(T)$. If all the attributes in A are certain then we can simply use the "usual" definition of select operator to get the result. If not, selection will introduce new dependencies in the resulting set R, as explained below.

Case 1 All the attributes $a_i \in A$ are certain: The schema $\Sigma_R = \Sigma_T$ and the dependency information $\Delta_R = \Delta_T$. A tuple $t \in T$ maps to a tuple $t' \in R$ (i.e. $t \rightarrow t'$), if $\Theta(t.A)$ is true. That is, $t'.a_i = t.a_i$, $\forall$ certain a_i and, $f_{t'}(S_i) = f_t(S_i), \forall S_i \in \Delta_R$. The history is simply "copied over" for all the dependency sets i.e. $\forall S_i, \Lambda(t'.S_i) = \Lambda(t.S_i)$. As an example, the result of performing a selection $\sigma_{id=1}(T)$ on the relation T presented in Table 8.1 would give us a single tuple $t = [1, Gaus(20, 5)]$.

Case 2 At least one of the attributes $a_i \in A$ is uncertain: The schema $\Sigma_R = \Sigma_T$ and dependency information $\Delta_R = \Omega(\Delta_T \cup \{A\})$. The closure Ω is defined as follows:

Definition 8.3.1 *Given a set system $\{S_1, S_2, \ldots, S_m\}$ representing a hyper-graph, the closure $\Omega(\{S_1, S_2, \ldots, S_m\})$ produces a set system $\{S'_1, S'_2, \ldots, S'_{m'}\}$ such that $S'_1, S'_2, \ldots, S'_{m'}$ represent the hyper-graph produced by merging all the connected components of $\{S_1, S_2, \ldots, S_m\}$.*

To illustrate, if $\Delta_T = \{\{a, b\}, \{c, d\}, \{e, f\}\}$ and $A = \{b, c, g\}$ (g is certain), then $\Omega(\Delta_T \cup \{A\}) = \{\{a, b, c, d, g\}, \{e, f\}\}$. Note that the sets $\{a, b\}$ and $\{c, d\}$ were *merged* due to the condition on A. The dependency set $\{e, f\}$ was not affected as it is disjoint from A. Note that some of the certain attributes in T may become uncertain in R.

Let us assume that a tuple $t \in T$ maps to a tuple $t' \in R$ (i.e. $t \rightarrow t'$). For all the certain attributes a_j in R, we have $t'.a_j = t.a_j$ (i.e., they are copied over). For the dependency sets that were disjoint from A, we do not need to do anything special. For the merged sets, we need to evaluate the resulting pdf. Thus, for $\forall S_k \in \Delta_R$, we have the following cases:

Case 2(a) ($A \cap S_k = \phi$): This is the case when S_k does not share any attributes with the selection set A, and thus using Definition 8.3.1 and the fact that all $S_i \in \Delta_T$ are disjoint, we can see that S_k is derived from exactly one attribute set $S_i \in \Delta_T$, i.e. $f_{t'}(S_k) = f_t(S_i)$.

Case 2(b) ($A \cap S_k \neq \phi$): Using Definition 8.3.1 it is easy to see that ($A \subseteq S_k$). In this case, S_k can be potentially derived from multiple attribute sets $S_i \in \Delta_T$. These attribute sets S_i are the sets for which ($A \cap S_i \neq \phi$). Let us assume $f_i, 1 \leq i \leq n$ are their respective pdfs. S_k consists of all the attributes in such sets S_i and A. Let us assume that C is set of all certain attributes ($C \subset A$) and c is the value of C in t. We define the identify pdf f_0 over C as $f_0(c) = 1$ and 0 otherwise. Now, we can derive the resulting pdf of S_k by performing a `product` operation over $f_0, f_1, \ldots, f_m$ and `floor`ing the resulting pdf in the region where $\Theta(A)$ is false. If the pdf of S_k is completely floored (i.e. the resulting probability of the tuple becomes 0), we remove that tuple from the result.

Similar to the previous case, the histories of the new dependency sets are updated to the combined histories of sets from which they are derived i.e. $\forall t' \in R$ and $\forall S_k \in \Delta_R$ where $t \rightarrow t'$, we have:

$$\Lambda(t'.S_k) = \bigcup_{\forall S_i \subseteq S_k, S_i \in \Delta_T} \Lambda(t.S_i)$$

Consider the example shown in Table 8.2. The probabilistic schema of that relation in our model would be represented as $\Sigma = (a : int, b : int)$ and $\Delta = \{\{a\}, \{b\}\}$. There are two tuples t_1 and t_2 in that relation with pdfs $f_{t_1}(\{a\}) = Discrete(0 : 0.1, 1 : 0.9)$ and $f_{t_1}(\{b\}) = Discrete(1 : 0.6, 2 : 0.4)$ (this notation represents a discrete pdf, whose parameters $x_i : y_i$ denote the probability y_i for value x_i). Similarly, we can write the pdfs of t_2 as $f_{t_2}(\{a\}) = Discrete(7 : 1.0)$ and $f_{t_2}(\{b\}) = Discrete(3 : 1.0)$. Applying a selection predicate $\sigma_{a<b}$ results in a table with schema $\Sigma = (a : int, b : int)$ and $\Delta = \{\{a, b\}\}$. This table consists of a single tuple t' with the joint distribution $f_{t'}(\{a, b\}) = Discrete(\{0, 1\} : 0.06, \{0, 2\} : 0.04, \{1, 2\} : 0.36)$. The history $\Lambda(t'.\{a, b\}) = \{t_1.\{a\}, t_1.\{b\}\}$.

Theorem 8.3.1 *The new pdf generated by selection operation is consistent with Possible Worlds Semantics.*

Proof This follows from PWS consistency for the operators `product` and `floor`. The `product` operation on contributing pdfs results in a joint pdf which is consistent with the PWS semantics for all the non-zero values of the new pdf. Now, the various selection criteria can be considered as multiple applications of the `floor` operation which set the pdf to zero for all possible worlds where the corresponding attribute values do not pass the selection criteria. In these possible worlds, the tuple containing this pdf will not exist. Since operation `floor` can be applied in any order, one does not need to re-apply selection criteria which were already captured by some dependency set S_i. □

8.3.5 Joins

The join of two tables $T_1 \bowtie_{\Theta(A)} T_2$ can be written as $\sigma_{\Theta(A)}(T_1 \times T_2)$. Thus, to define the semantics of joins, we can use the semantics of selection and cross-product. We have already seen selection, the cross-product $R = T_1 \times T_2$ is defined as follows. $\Sigma_R = \Sigma_{T_1} \cup \Sigma_{T_2}$ and $\Delta_R = \Delta_{T_1} \cup \Delta_{T_2}$. Let us assume a tuple $t \in R$ is derived from tuples $t_1 \in T_1$ and $t_2 \in T_2$ (i.e. $(t_1, t_2) \rightarrow t$). $\forall S_k \in \Delta_R$ and the corresponding

$S_i \in \Delta_{T_c}, c = 1$ or 2 we have, $f_t(S_k) = f_{t_c}(S_i)$. Similarly, the history is also copied over for the new sets, $\Lambda(t'.S_k) = \Lambda(t_c.S_i)$.

Thus, conceptually joins are an application of cross-product followed by selection (as defined in Section 8.3.4). The tuples that are produced as a result of join may contain some dependencies (implied by history Λ) which are not captured by the attribute dependencies (implied by Δ_T). We can, in principle, apply the algorithm explained in Section 8.3.4 to collapse the intra-tuple dependencies implied by Λ into Δ_T. This decision will not affect the correctness or the semantics of the operations defined in this section but will have a significant effect on performance. The definition of the operations in this section assumes a lazy merging of dependencies and evaluation of joint pdfs. In practice, a combination of these techniques can be used to improve performance. Thus, the decision of whether to merge the intra-tuple dependencies eagerly or lazily is left to the implementation.

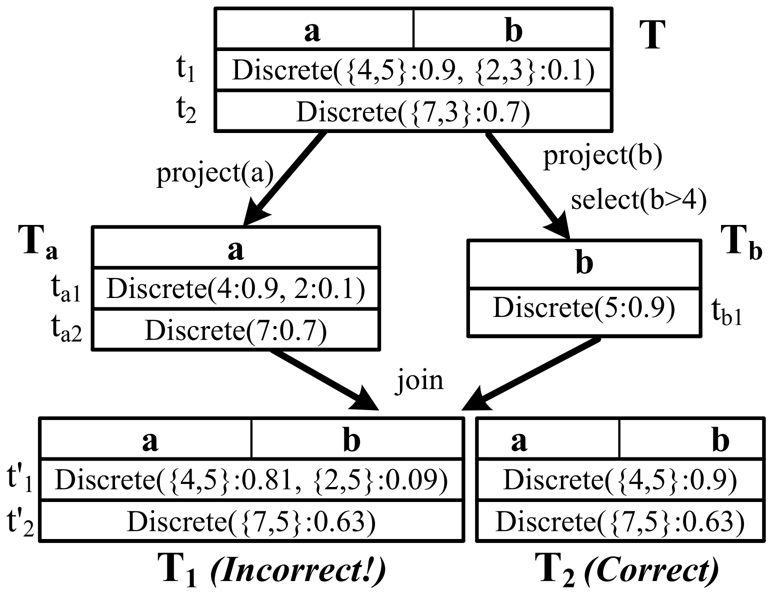

Figure 8.3. Example illustrating histories.

Consider as an example, a table T with $\Sigma_T = (a : int, b : int)$ and $\Delta_T = \{\{a, b\}\}$ as shown in Figure 8.3. We perform operations $\Pi_a(T)$ and $\Pi_b(\sigma_{b>4}(T))$ to obtain the tables T_a and T_b (In this example, we do not need to keep the projected out attributes, as both the attributes a and b functionally determine each other in both the tuples). Clearly, $\Sigma_{T_a} = (a : int)$ and $\Delta_{T_a} = \{\{a\}\}$ for T_a; and $\Sigma_{T_b} = (b : int)$ and $\Delta_{T_b} = \{\{b\}\}$ for T_b. Now, if we join T_a and T_b *without* considering historical dependencies we would get an incorrect result T_1. The tuple $(2, 5)$ in t'_1 can never exist because it do not exist in any possible world corresponding to table T. Similarly, the probability of tuple $(4, 5)$ in T_1 is incorrect as the pdfs of t_{a1} and t_{b1} share common ancestor $t_1.\{a, b\}$ and thus the two events cannot be considered independent. Our model detects the historical dependency between tuples t_{a1} and t_{b1} and uses that information to correctly calculate the distribution of tuple t'_1 in the final table T_2 by considering the joint distribution of attributes a and b in T. In addition, as part of the tuple value $(2, 3)$ $(\in T)$ was floored in table T_b, we correctly floored that value in the distribution of $t'_1.\{a, b\}$.

The correctness of the project and join operations with respect to the possible world semantics follows from the correctness of the selection operation and are thus omitted. Given the definition and the correctness of the selection, project, and join operations, we obtain the following theorem.

Theorem 8.3.2 *Our model is closed under selection, projection, and join operations.*

8.3.6 Operations on Probability Values

We also allow queries based on the probability values of the tuples in our model. One example of such queries are threshold queries. Given a table T with probabilistic schema (Σ_T, Δ_T), a threshold query $R = \sigma_{Pr(A)>p}(T)$, where $A \subseteq \Sigma_T$ and p is the probability threshold, returns all tuples whose probability over the attribute set A is greater than p. As the operations on probability values act on the probabilistic model instead of a *possible world*, the possible worlds semantics described in Section 8.1 is not be used to define the semantics of these operations.

In general, consider the boolean predicate given by $\Theta(S)$, where $S = \{Pr(s_1), Pr(s_2), \ldots, Pr(s_m)\}$ and $s_i \subseteq \Sigma_T$. The result R of applying this selection on T consists of all tuples $t \in T$ such that t satisfies $\Theta(S)$. The semantics of this operation and effect on histories is similar to Case 1 defined in Section 8.3.4.

8.4 Chapter Summary

In this chapter, we presented a new model for handling arbitrary pdf (both discrete and continuous) attributes natively at the database level. Our approach allows a more natural and efficient representation and implementation for continuous domains. The model can handle arbitrary intra- and inter-tuple correlations. We show that our model is complete and closed under the fundamental relational operations of selection, projection, and join. In our previous work we have developed Orion – an extension of PostgreSQL that provides native support for attribute uncertainty with procedural semantics. We have extended Orion to support our new model. The experiments presented in Chapter 9 show the effectiveness and efficiency of our approach.

The results of this chapter can be found in [60]. We shift our focus to the implementation details of Orion in the next chapter.

9 ORION IMPLEMENTATION

This chapter discusses the implementation of Orion system [45], which is a state-of-the-art uncertain database management system with built-in support for probabilistic data as first class data types.

Uncertainty is prevalent in numerous application domains, ranging from information extraction and integration to scientific data management and sensor databases. Orion is a general-purpose uncertain DBMS that unifies the modeling of probabilistic data across applications. This in turn provides additional opportunities to the query engine for indexing and optimization.

One motivating example is a data cleaning system that automatically detects and corrects errors. Since conventional database management systems assume data to be certain and precise, the software must either construct its own probabilistic model for the data, or simply pick one of the alternative values to store in the underlying database. This leads to a no-win situation: the first option significantly complicates the queries, while the second technique results in a substantial loss of information.

The Orion system provides a better solution: built-in support for uncertainty at the database level. By extending the query processing engine of PostgreSQL, Orion natively manages uncertain data.

There are two major versions of Orion system. The first system, called Orion 1.0 was developed based on the attribute uncertainty model presented in Chapter 3. The major focus of Orion 1.0 was on Probabilistic Threshold Queries. Orion 2.0 was later developed using the ideas presented in Chapter 8 and supports Possible Worlds Semantics [59]. Both these systems are implemented inside PostgreSQL system [44]. We next provide a brief overview of these two systems.

9.1 Orion 1.0

Orion 1.0 implements the Uncertainty model presented in [11] and discussed in Chapter 3. This model assumes that each data item can be represented by a range of possible values and their distributions. The current version is publicly available under the Purdue Free License and can be downloaded from [45].

9.1.1 System Architecture

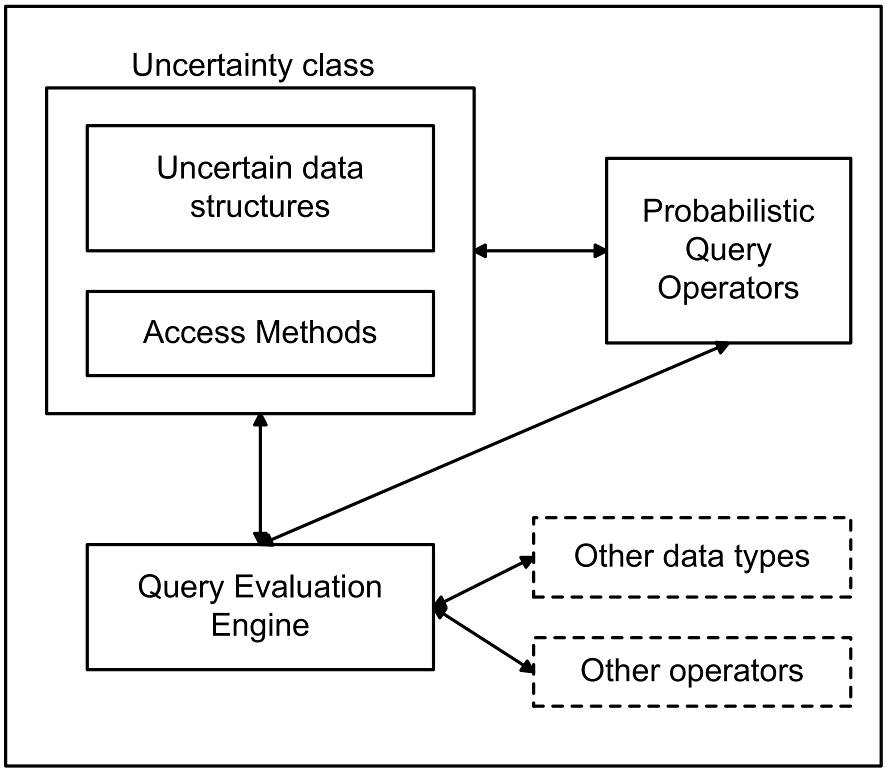

Figure 9.1. Architecture of Orion 1.0.

We develop our system on PostgreSQL [44] because it is an open-source system. Also, its object-oriented design allows us to extend the functionalities easily without modifying its internal code. We define new data types and queries through developing external C libraries, and linking them with the PostgreSQL source code. Another advantage is that the uncertainty functionalities do not interfere with the original database; instead, uncertain and certain data can be "blended" together, and they can be used by database queries at the same time. The high level architecture of Orion is shown in Figure 9.1.

As shown in Figure 9.1, we have added a new class, called *uncertainty class*, which stores the data structures and access methods of uncertain data. We have also implemented a set of probabilistic query operators to manipulate the uncertain data. The interface of the query evaluation engine is modified to interact with both the uncertainty class and query operators. All other existing data types and operators in the PostgreSQL system (dotted-line boxes) remain intact.

9.1.2 Supporting Uncertain Data

We support four types of data uncertainty: (1) Gaussian, (2) uniform, (3) histogram and (4) discrete. While Gaussian and uniform distributions are commonly found in applications, we want to develop a system that is general enough to support any kind of pdfs (e.g., *Zipf* and *Poisson* (for describing the frequency of events)). Moreover, arbitrary operations on an uncertain item with standard distribution can render a non-standard distribution. For example, the sum of two uniform distribution is a triangular distribution. A histogram allows us more flexibility in query operator implementation. Internal functions that convert different pdf types to histogram pdf are also implemented.

In order to represent these data types, we define the *uncertain* class (with keyword **UNCERTAIN**), as shown in Figure 9.1. It is a variable-length data type, which can store an uncertain value (Gaussian, uniform, histogram or discrete). The design of this class is flexible, and other kinds of uncertainty pdf (e.g. Poisson) can be added to it with minimal change.

Like other relational database systems, PostgreSQL stores internal bookkeeping information in catalogs (which are internally represented as tables). One key difference is that PostgreSQL stores much more information in these catalogs, such as data types, access methods and functions. Thus, PostgreSQL can be modified or extended by changing these catalogs. Moreover, the PostgreSQL server can incorporate user-written codes through dynamic loading. Thus, the user can specify a shared library that implements a new type or function, and these will be incorporated into the server automatically.

To create a new uncertain data type, we use shared C libraries to specify the internal representation of the data type, along with "helper functions" that operate on the data types. These access functions are specified by the interface that PostgreSQL uses to interact with a data type. The query engine interacts with the uncertain data type through these access functions, as shown in Figure 9.1. Once these helper functions are properly set, the uncertain data type becomes one of the data types in PostgreSQL.

9.1.3 Probabilistic Queries

To support probabilistic queries, we provide PostgreSQL with the semantics of operations like $=$, $\neq$, $>$, $<$ on each uncertainty type, using compiled C functions (Figure 9.1).Once these operations are defined, PostgreSQL automatically supports queries like joins and selections for uncertain data by interacting with the probabilistic operators as shown in 9.1.

We emphasize that only one uncertainty type, specified by the **UNCERTAIN** keyword (with parameters describing uncertainty pdf type), is used. We choose not to provide one keyword for each pdf type. A user should not have to think, for example, what the resulting pdf is when a Gaussian pdf is multiplied with a uniform pdf. In Orion, the user only needs to specify the result is **UNCERTAIN**. The system decides the most appropriate resulting pdf.

Inserting Uncertain data. The following statement shows how a table with two attributes (k, a) is created, where k and a are primary key and uncertain values, respectively. The keyword `uncertain` specifies that a is uncertain.

```
CREATE TABLE T (
k INTEGER PRIMARY KEY,
a UNCERTAIN);
```

This schema is used for further discussion in this section. An uncertain value is inserted as follows:

```
INSERT INTO T VALUES (1, '(g, 0, 5)');
```

where, g specifies that the Gaussian distribution is used and 0, 5 are its mean and variance respectively. Similarly, we can insert a histogram or a discrete distribution in Orion 1.0.

Extracting Uncertainty Information. Orion 1.0 allows the details of uncertain attributes, like the lower bound of the uncertainty interval, the uncertainty pdf, and data quality (e.g. mean and variance), to be extracted. For example, the following query obtains the lower bound of a.U.

```
SELECT u_lower(a) FROM T;
```

SPJ Queries over uncertain data. Orion supports a number of queries over uncertain data. Some examples of such queries are selections, projections, joins, entity minimum/maximum query and value-minimum query [11]. All the probabilistic conditions are converted into a boolean predicate using a probabilistic threshold. For example, consider a SPJ query:

```
SELECT R.k, S.k
FROM R,S
WHERE R.a = S.a;
```

The uncertain attribute a is used to perform an equality join between tables R and S. The probabilistic condition $R.a = S.a$ is converted into a boolean predicate by converting it into a threshold condition $Pr(R.a = S.a) > p$, where p is a system defined threshold.

Quality. In addition to the usual database operations, Orion 1.0 supports quality queries to compute the quality (preciseness) of uncertain values and query results [11].

Indexing and Cost Estimation. For efficient query execution, Orion uses Probabilistic Threshold Index (PTI) introduced in [30]. The PostgreSQL query optimizer uses this index automatically when it estimates that the cost of an index scan will be lower than sequential scan. For this purpose, it uses the cost estimation techniques presented in Chapter 7.

A complete discussion of the internals of Orion 1.0 is beyond the scope of this thesis and can be found in the Orion 1.0 documentation [45].

9.2 Orion 2.0

In this section, we describe Orion 2.0, which is the current iteration of Orion. Although there are superficial similarities, Orion 2.0 fundamentally differs from the earlier version of Orion.

In contrast to other uncertain databases, Orion 2.0 supports both attribute and tuple uncertainty with *arbitrary correlations*. This enables the database engine to handle both discrete and continuous pdfs in a natural and accurate manner. The underlying model is closed under the basic relational operators and is consistent with Possible Worlds Semantics [59].

Orion 2.0 includes the following new and innovative contributions:

- An integrated implementation (*within* PostgreSQL) of the “PDF Attributes” data model, which is consistent with *Possible Worlds Semantics* (PWS) and supports both continuous and discrete uncertainty (See Chapter 8).

- Efficient access methods for querying uncertain data, including three index structures based on R-trees, signature trees, and inverted indexes (See [30] and Chapter 6).
- Improved query optimization, join algorithms, and selectivity estimation by gathering and exploiting additional statistics over probabilistic data types (See Chapter 5 and Chapter 7).
- Integration with PL/R for graphical visualization of and statistical inference over uncertain data [61].

Orion 2.0 is an implementation of the new uncertainty model described in Chapter 8.

9.2.1 System Implementation

Orion is primarily written in C, with some portions at the user level in PL/pgSQL. Figure 9.2 gives a high level overview of the system architecture. The shaded regions represent new components that correspond to the primary features of the Orion data model. Partially shaded boxes highlight portions of the PostgreSQL backend we extended to support queries over uncertain data. Clear boxes (which include the majority of the PostgreSQL backend) indicate components that have not been modified.

Query interface

One underlying goal in the design of Orion was to support uncertain data with minimal changes to SQL. The resulting user interface is standard SQL plus a handful of data types and built-in functions for manipulating probabilistic data. These include, for example, evaluating the cdf of an uncertain attribute, and converting symbolic pdfs into approximations. In addition, we have integrated our system with PL/R [61], an extension to PostgreSQL that allows the user to write SQL statements and functions in the R programming language. “R is a free software environment for

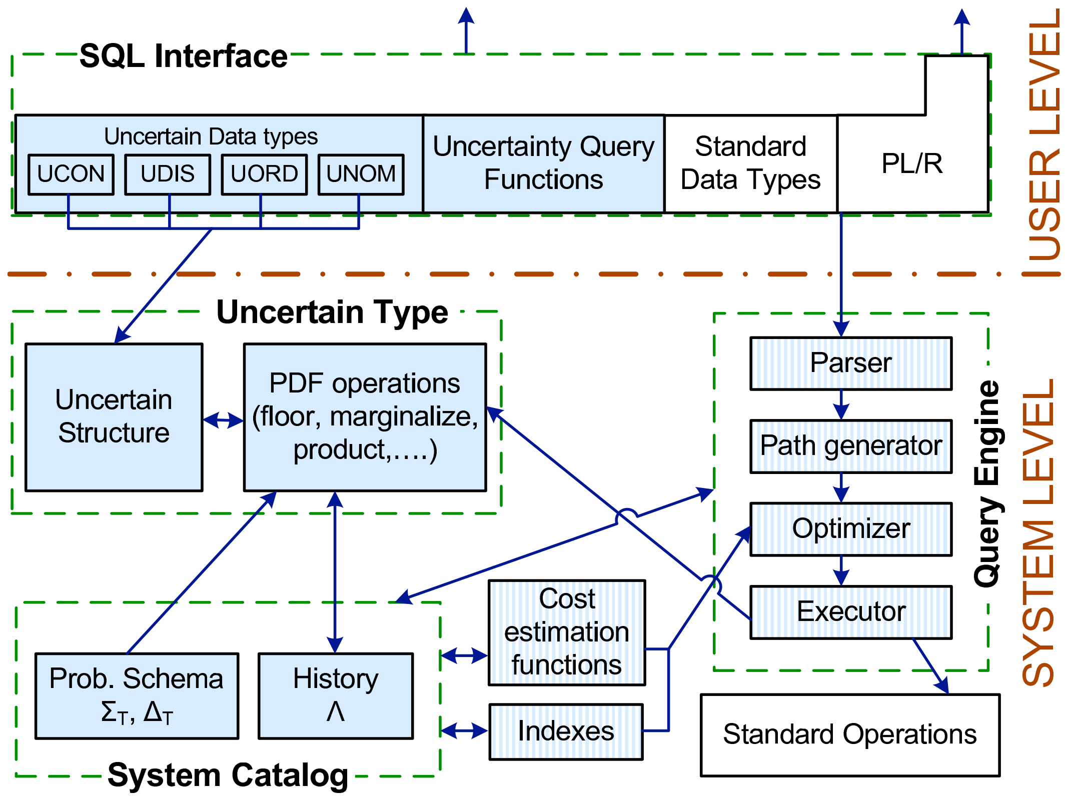

Figure 9.2. Architecture of Orion 2.0.

statistical computing and graphics,"[1] and provides elegant visualization of uncertain pdfs in the Orion client.

Uncertain data types

Orion supports four main types of uncertain data attributes:

1. Continuous Numeric (`ucon`) – Each data item has an associated probability density function for evaluating the probability of any given value.
   Example: Temperature or voltage from a sensor.

[1]See `http://www.r-project.org/`

2. Discrete Numeric (`udis`) – Each data item has a probability distribution function, which stores the frequencies of the alternative values.
   Example: Number of neighbors in a mobile network.

3. Ordered Categorical (`uord`) – Similar to discrete numeric, each data type comes with a pdf that stores probabilities for each category.
   Example: Fuzzy data value, e.g. low or high.

4. Unordered Categorical (`unom`) – Same as above, except there is no logical ordering between categories.
   Example: Document classification or generic type.

Internal representation

All the uncertain attributes are stored internally using a data structure called `Uncertain`. This type is hidden from the user, and is only accessible through the four SQL data types listed above. Consequently, the data structure is generic and represents all possible types of uncertainty pdfs. In particular, it can represent both independent and joint distributions. When multiple attributes are correlated, the system automatically stores the number of dimensions, the type of each dimension, and the resulting joint pdf in a single data instance.

In addition to the pdf, `Uncertain` also maintains a list of floored regions and historical dependencies that are due to operations on pdfs. Probabilistic schemas (i.e. dependency sets) for each table are stored in the system catalog. All of this information is used by internal functions to detect correlations while performing pdf operations.

Query examples

The following examples show how probabilistic tables are created, populated and queried. Note that apart from a few additional keywords, the interface is identical to standard SQL.

```
-- Query1
CREATE TABLE location (
  id integer, ts time,
  xloc ucon, yloc ucon, room udis,    -- unc. types
  PRIMARY KEY (id, ts),
  DEPENDENT (xloc, yloc) );           -- prob. schema

-- Query2
INSERT INTO location VALUES (
  1, '2008-06-09 14:05:27',
  'prod( norm(5,3) , norm(7,3) )',    -- 2D pdf
  'dist( 2 : 0.75 , 3 : 0.25 )' );    -- 1D pdf

-- Query3
SELECT xloc, room FROM location       -- marginalized and floored pdfs
  WHERE xloc > 5 and yloc < 5;        -- with history
```

Query Rewriting

To support the uncertainty model described in Chapter 8, we used query rewriting techniques. The other option was to introduce the probabilistic queries natively in PostgreSQL. This would have involved major changes to PostgreSQL query engine including the parser, path generator, optimizer and executor. The intrusive changes would have hindered the power of PostgreSQL to handle certain (or precise) attribute in the tables. On the other hand, as this approach required the implementation of probabilistic queries from scratch, it would have given us a lot of flexibility for the implementation of the model.

After a careful examination of the query discussed in Chapter 8, we found out that all the queries can be implemented by using query rewriting techniques. Although, this limits our flexibility, this technique allowed us to leverage the existing PostgreSQL infrastructure for query optimization and execution with minimal changes. As described in Chapter 8, internally all SQL queries are rewritten in terms of the three basic functions on pdfs: floor, product and marginalize.

To illustrate this point, consider Query1. Both `xloc` and `yloc` are stored together as one joint pdf as they are jointly distributed (specified by the keyword `DEPENDENT`). Further, all uncertain data types are stored internally as `UNCERTAIN`. Therefore, Orion rewrites this query as:

```
-- Query1 rewritten
CREATE TABLE location (
  id integer, ts time,
  xloc uncertain, room uncertain,
  PRIMARY KEY (id, ts));
```

Additionally, it stores the information specifying the original data types (of `xloc` and `room`) and their dependency sets in its schema.

Similarly, Query2 is rewritten to enforce type checking. Using the schema information created during Query1, it checks if the 2D pdf that is being inserted into `xloc` corresponds to (`ucon`, `ucon`). Similarly, it checks if the pdf that is being inserted into `room` is of type `udis`.

Query3 consists of a selection followed by projection. The selection conditions are converted into corresponding floor calls to `xloc` and `yloc` according the the rules discussed in Chapter 8. This is followed by calls to marginalize function to generate the final result. The final rewritten query that gets executed is:

```
-- Query3 rewritten
```

```
SELECT marginalize(floor(floor(xloc, !(xloc > 5)), !(yloc < 5)), 0)
  as xloc, room FROM location
  WHERE nonzero(xloc);
```

The second argument of `floor(.,.)` function specifies the pdf regions that is to be floored. In this case, the application of the two floor functions in sequence corresponds to the selection condition `(xloc > 5 and yloc < 5)`. The fact that only the first dimension (corresponding to `xloc`) is to be retained is expressed by the second argument of `marginalize(.,.)`. The final condition guarantees that all the resulting tuples have non-zero probability of being present in the result.

Indexes and query optimization

The standard cost estimation and indexing techniques built into PostgreSQL are not appropriate for uncertain data. Orion provides novel query cost estimation techniques that are used for optimizing the generated query plans involving uncertain data (See Chapter 7). In addition to cost estimation, Orion also includes a number of uncertainty indexing methods and join algorithms for efficient execution of specialized queries (See Chapter 5 and Chapter 6).

Minimal overhead

One major advantage with the design and implementation of Orion is that there is virtually no system overhead in the absence of uncertain data. The modifications for uncertain data support are for the most part self-contained, and operate side by side with the standard indexing and query optimization components.

9.3 Comparison with other Systems

Due to the importance of uncertainty management in real-world applications, several database systems for managing uncertain data have been proposed (See Chapter 2). Table 9.1 shows the comparison of Orion with some of the recently proposed uncertainty management systems. As described earlier, Orion is the only DBMS with support for continuous uncertain data. Although all the models support the tuple uncertainty model, only MayBMS and Orion support the attribute uncertainty model. With a recent change, Trio has added limited support for attribute uncertainty, as it can now store a joint distribution for all the uncertain attributes in a tuple. This is opposed to our model, in which, multiple uncertain attributes can be stored independently. All databases, except MystiQ, support an exact query evaluation technique. MystiQ uses approximations for the queries for which it cannot find a safe plan [17]. As regards to implementation, Orion and MayBMS are both implemented as an extension to PostgreSQL, whereas the other database managements systems are implemented as a wrapper to a relational DBMS. Finally, only Orion supports indexing and query optimization for uncertain data.

9.4 Experimental Evaluation of Orion 2.0 Model

This section presents the experiments performed using Orion 2.0 to validate the efficiency of the model presented in Chapter 8. Orion not only allows us to validate the accuracy of our methods in a realistic runtime environment, it also gives additional insight into the overall effect our techniques have on probabilistic query processing in an industrial-strength DBMS. The following experiments were conducted on a Sun-Blade-1000 workstation with 2 GB RAM, running SunOS 5.8, PostgreSQL 8.2.4, and Orion 2.0.

Using a series of synthetically generated datasets, we explore the performance and accuracy of our model's operations over pdfs. Each dataset consists of random "sensor readings," using the schema `Readings(rid, value)`. The uncertain pdfs (e.g. reported from the sensors) are Gaussians, with their means distributed uniformly from 0 to 100, and their standard deviations distributed normally using $\mu = 2$ and $\sigma = 0.5$. We also generate numerous range queries, with midpoints distributed uniformly

Table 9.1
Comparison of Orion 2.0 with other Uncertainty Management Systems.

	Trio [3]	**MayBMS [43]**	**MystiQ [17]**	**Orion 2.0**
Continuous uncertain data	No	No	No	Yes
Tuple uncertainty	Yes	Yes	Yes	Yes
Attribute uncertainty	Limited	Yes	No	Yes
Correlations	Yes	Yes	Yes	Yes
Evaluation technique	Exact	Exact	Sometimes approximate	Exact
Implementation technique	Outside DBMS	Inside DBMS	Outside DBMS	Inside DBMS
Indexes and query optimization	No	No	No	Yes

between 0 and 100, but with interval lengths distributed normally using $\mu = 10$ and $\sigma = 3$.

For simplicity, we omit the initial results of evaluating pdfs symbolically because they produce no approximation error and incur negligible overhead. Instead, our results focus on the relative performance of approximating symbolic pdfs with histograms as opposed to discrete sampling. Although it's obvious theoretically that histograms will generally outperform discrete representations, we wish to quantify the observed difference of these two approximations in our actual implementation.

9.4.1 Accuracy vs Sample Size

The first experiment shows the average error when answering range queries over histogram and discrete approximations of symbolic pdfs. We first discretize our dataset of random Gaussian pdfs, varying the number of sample points. Figure 9.3 shows the average approximation error of the cdf values returned at each sample size. The standard error over these averages is negligible. As expected, the histogram representation outperforms the discrete, even in the worst case (not shown). With only five sampling points, the accuracy is around ± 0.01 probability mass. A discrete approximation requires over twenty-five sampling points, which greatly increases the size of each tuple and thus the overall I/O cost. Of course, a symbolic representation is both ideal in storage size and accuracy.

We also show the standard deviation of the error values themselves, at each sample size, plotted only in the positive direction for clarity. As expected, a discrete representation has a considerably higher variance in approximation error than a histogram. Sometimes the error is quite large, for example in boundary cases when the query barely misses a discrete point. Continuous representations (including histograms) avoid this issue altogether because they can accurately estimate probability mass at arbitrary points. The difference in error is likely to be even greater in more complex pdfs.

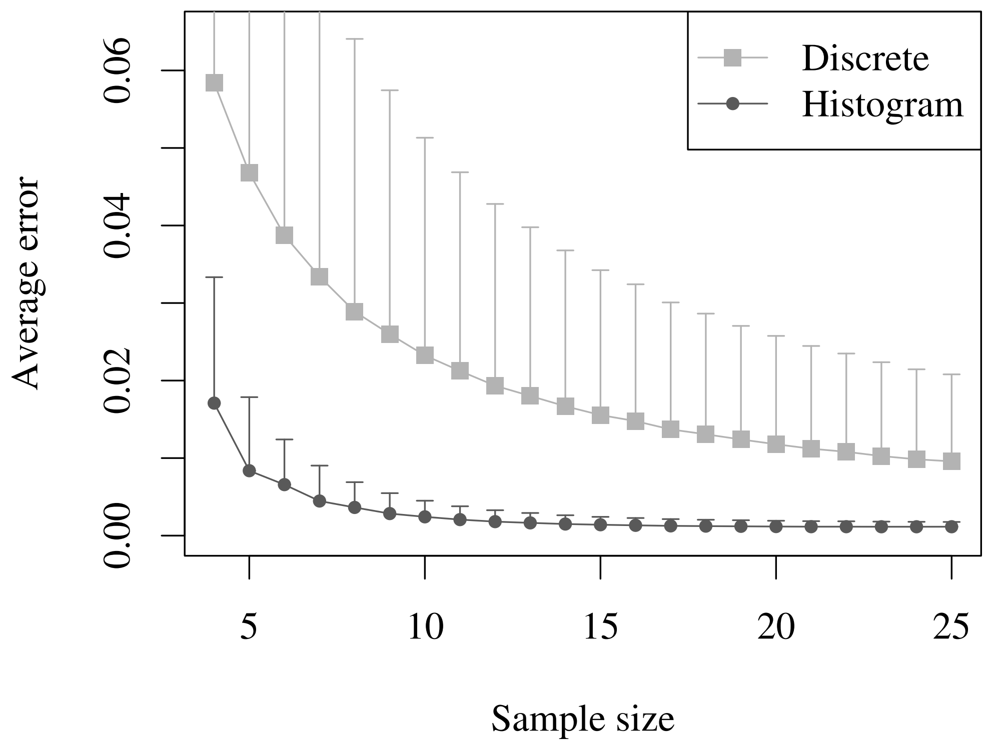

Figure 9.3. Accuracy vs Sample Size.

9.4.2 Performance of Discretized PDFs

For this experiment, we compare the performance of the aforementioned approximate representations. We fix the number of histogram bins at five and the number of discrete sample points at twenty-five, in order to compare runtimes at an equivalent level of accuracy. As shown in Figure 9.4, discretizing the data not only takes additional processing time, but also incurs more disk reads, yielding a steeper rise in cost. Runtimes for the symbolic representation are just under the five-bin histogram times, but we do not show these here since they give an even higher level of accuracy.

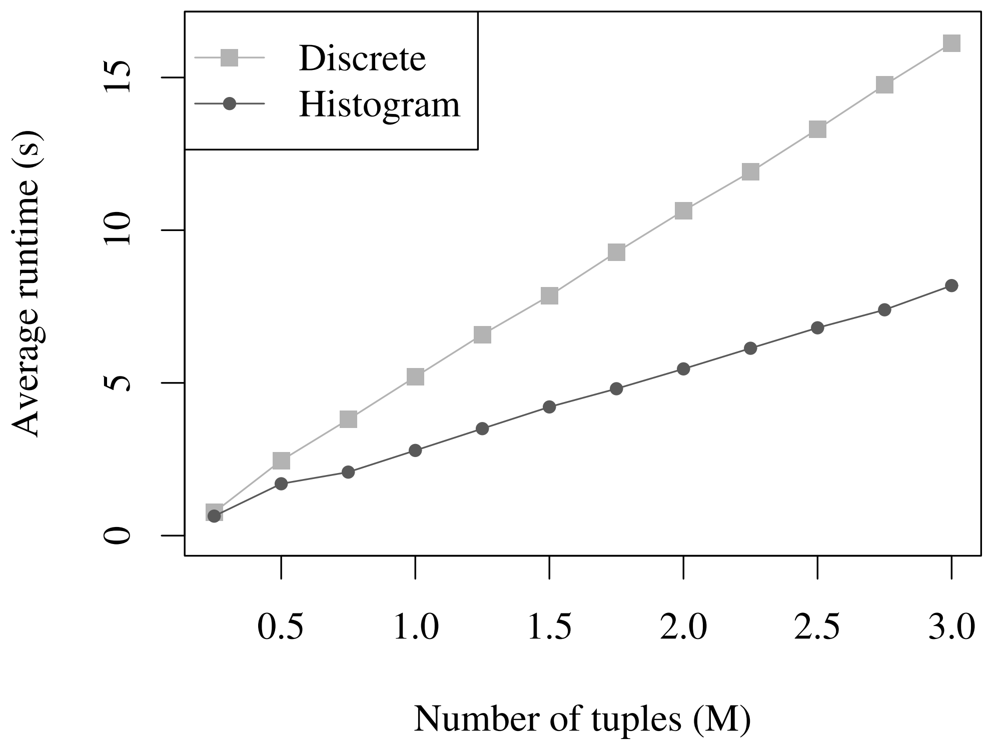

Figure 9.4. Performance of Discretized PDFs.

9.4.3 Overhead of Histories

The final experiment shows the overall performance of the implementation of our proposed model inside PostgreSQL. We run two types of queries: joins over range queries (which involve floors and products), and projections of the resulting correlated data (triggering a collapse of the 2D pdfs). Figure 9.5 compares the average runtime of these queries with and without the overhead of maintaining histories for correctness. Note that ignoring histories will result in incorrect answers. The overhead shown in this figure ranges between 5-20%. Thus, although the proposed model is complex, it is efficient to implement and we pay a small overhead for correctness.

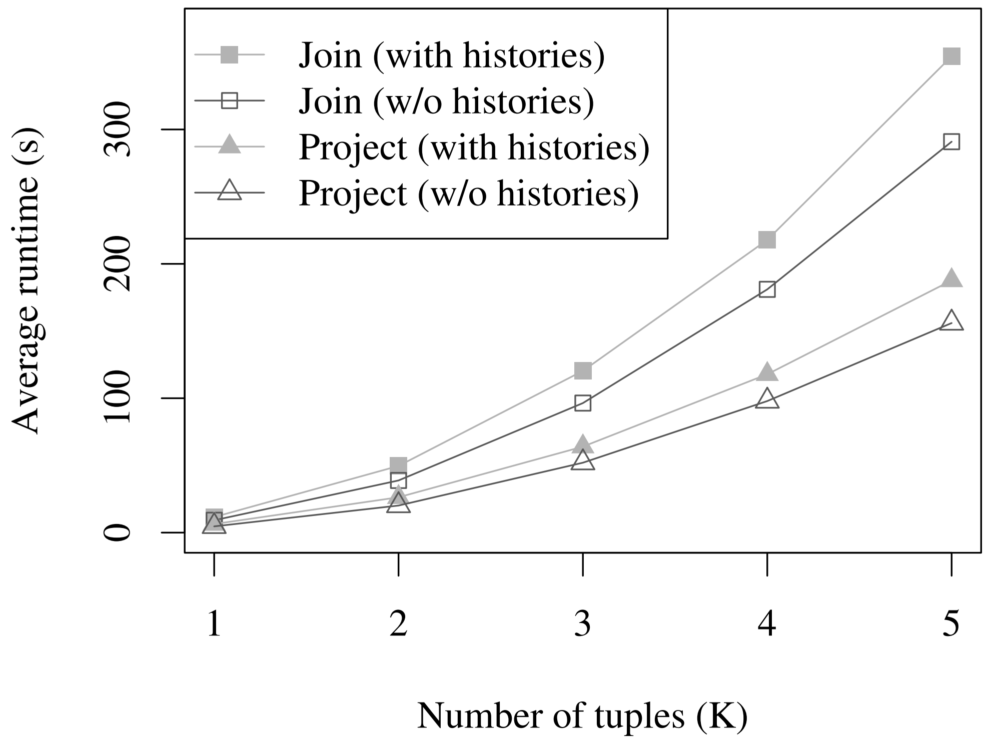

Figure 9.5. Overhead of Histories.

9.5 Chapter Summary

The Orion project aims to build a general-purpose uncertain DBMS to support both current and forthcoming applications. Research and development of a database system that supports uncertain data will advance scientific understanding and enable future work in a variety of fields. But whether emerging applications use databases simply as an information storage technology rather than an effective data management solution depends on to what extent they can reason about and make use of the uncertainty of data directly.

The Orion system was presented in numerous conferences [62–64] and workshops and received considerable interest from the research community. In the next chapter, we present the future work in the field of uncertain data management.

10 FUTURE WORK

This chapter presents our ongoing and future work in the field of uncertainty management in databases.

10.1 Modeling and Approximations

The current model described in Chapter 8 unifies both tuple and attribute uncertainty and is closed under the simple database operations of selections, projections and joins. If we extend the same model to handle operations like projections with duplicate elimination and aggregates, the history graphs become very complex. A future goal is to define the semantic meaning of these operations according to PWS and extend the current model to handle them efficiently.

Many applications require enforcing of constraints on probabilistic data. Some examples are primary/foreign key constraints and functional dependencies. We are currently expanding our current model to express these constraints.

There are many useful operations on probabilistic data that can not be explained through Possible Worlds Semantics as they involve conditions on aggregate probabilities of possible worlds. Threshold queries and distributional similarity are examples of such operations. We plan to extend the PWS to define the semantics of such operations.

There are many cases when the user is not concerned about the *exact* probabilities of query results. In these cases, the model should be able to generate approximate probabilities along with some guarantees over them. We would like to include the ability to do such approximations in the model. Some possible solutions would be to use Monte Carlo simulations and/or approximate inferencing. These approximations will be especially useful when the historical dependencies are complex and exact probability calculations would be computationally very expensive for the system.

10.2 Nearest Neighbor Queries

There has been some work on supporting Nearest Neighbor (NN) queries on uncertain data [31]. In case of certain data, the definitions of NN and k-NN are obvious, but the semantics of these queries over uncertain data is not very clear. We present some of the possible semantics of NN queries in the following definitions.

Definition 10.2.1 *Given a set U of uncertain data, a Nearest Neighbor query Q on this data returns the tuple $a \in U$ which is most likely to be the nearest neighbor of query point Q.*

This definition is the most widely used definition of nearest neighbor queries in the current literature. A k-NN of above query just returns k tuples sorted according to their probability values which are the most likely contender for the NN.

Note that this is not the only possible way of defining NN queries. Below we present an alternate formulation of the NN queries that is based on the notion of τ-radius.

Definition 10.2.2 *Given a query point q, a threshold τ and an uncertain item a, τ-radius r of a is defined as the distance from a query point q such that the probability that a lies in a radius of r around q exceeds τ.*

Given this notion of τ-radius, we can define the alternate notion NN of a query point q as:

Definition 10.2.3 *Given a query point q, a threshold τ and a set of uncertain items U, the NN of q is the tuple $a \in U$ which minimizes the τ-radius over all the uncertain items in U.*

Similarly, a k-NN of q would a data item which has the k-th minimum τ-radius. We explored some of these aspects in [56]. We plan to future investigate the NN queries based on above definitions and derive efficient solutions for evaluating them.

10.3 Query Optimization

As we have seen earlier, there has been a recent interest in pushing uncertainty management to the database level. Once we have database systems capable of managing uncertain data, we would need to efficiently execute queries over the uncertain data. For certain data, various techniques for indexing and cost estimation have been developed. Similar techniques for uncertain data need to be developed. Our indexing and selectivity work was the first step in this direction. We would like to continue in this direction by developing novel indexing and cost estimation techniques for uncertain data.

While the certain data in databases is usually uncorrelated, correlations are naturally present in uncertain data. This important distinction can be used to develop query optimization techniques that can suggest alternate query plans based on correlations that are observed in the input data.

Another important dimension in Query optimization is the issue of query plans. For certain data, there has been a lot of work for generating and transforming query plans into near optimal query plans. Many operations over uncertain data involve threshold predicates and/or top-k queries. We need to develop new algorithms and heuristics for pushing these predicates down the query plan. We have already started work in this direction and plan to extend our indexing and cost estimation solutions for it. The final goal would be to implement these techniques inside the PostgreSQL query optimization engine.

10.4 Data Mining and Information Retrieval

In recent years there has been a surge of interest in problems related to Data Mining and Information Retrieval. There has been a lot of work for developing algorithms that either work on certain data or assume that the uncertainty in the data can be removed (e.g. by taking the most probable value). The underlying data that these algorithms often work on can have uncertainties inherent in them. We plan to extend the current algorithms (e.g. associating rule mining, classification, clustering) in the fields of Data mining and Information Retrieval to handle uncertain data. An important challenge is to develop efficient algorithms that are able to produce returns that show *significant* improvement over the base case when uncertainties are ignored, without too much computational overhead.

10.5 Privacy and Anonymity

Uncertainty can be used to guarantee privacy and anonymity. The basic idea is to "blur" the certain data points (e.g. location, salary) by adding uncertainty to them. While tuple-uncertainty (or discrete uncertainty pdf) can be used for this purpose, the concept of continuous uncertainty pdfs is more useful for privacy. Our model already handles continuous uncertainty and correlations and it would be interesting to see how we can extend the current model for privacy and anonymization, while providing some useful guarantees.

10.6 Reliability

While the notion of reliability looks very similar to uncertain data with tuple uncertainty (i.e. if you have probabilities of existence associated with each tuple), the actual problem is much more complex. The reliability of data can change after the data is inserted into the database, hinting at using histories for maintaining reliability. Further, reliability can be very subjective with each user having his/her own notion of data reliability. In a sense, the concept of reliability is complementary to the concept of data accuracy and quality. We would like to explore this domain further and develop techniques based on uncertain data management which can handle data reliability issues.

11 CONCLUSION

Due to the presence of numerous applications that handle probabilistic data, uncertainty management in databases has attracted considerable research interest in recent years. The ultimate goal of this research is to take the burden of managing uncertain data away from the applications to the database systems. In this dissertation, we identified and solved important issues for managing uncertain data natively at the database level.

We proposed the semantics of join operation in presence of attribute uncertainty. Joining uncertain data can be very costly and we discussed three pruning techniques to reduce this cost. The experiments show that with only a small overhead these techniques can improve the join performance significantly. We presented two index structure for indexing categorical (discrete) uncertain data. Since such uncertainty can be considered as an extension of set-values attributes, we proposed the extension of signature trees and inverted indexes for this problem. These index structures were shown to be efficient and have good scalability with respect to the dataset and domain size. For query optimization of probabilistic queries, we presented novel selectivity estimation techniques. These techniques were shown to be efficient and gave good estimates for threshold queries.

A new model for handling arbitrary pdf (both discrete and continuous) attributes natively at the database level was also presented. Our approach allows a more natural and efficient representation and implementation for continuous domains. The model can handle arbitrary intra- and inter-tuple correlations. Our model is consistent with Possible Worlds Semantics and is closed under fundamental relation operations of selection, projection and join.

We also presented and discussed the implementation of Orion – an extension of PostgreSQL that provides native support for uncertain data. We have extended Orion to support our new model. The experiments performed in Orion show the effectiveness and efficiency of our approach.

Finally, we presented our ongoing and future work in the field of uncertain data management. Despite the significant gains already made, numerous interesting open problems remain. The new model presented in this thesis raises a number of interesting problems that have not been solved yet. We believe that this dissertation is an important step towards the realization of our goal of managing uncertainty natively at the database level.

LIST OF REFERENCES

LIST OF REFERENCES

[1] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2004.

[2] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2004.

[3] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings Conference on Innovative Data Systems Research (CIDR)*, 2005.

[4] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[5] A. Hendrich, M. Chow, B. Skierczynski, and Z. Lu. A 36-hospital time and motion study: How do medical-surgical nurses spend their time? *The Permanente Journal*, 12(3), 2008.

[6] N. Khoussainova, M. Balazinska, and D. Suciu. Towards correcting input data errors probabilistically using integrity constraints. In *Proceedings of the 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 2006.

[7] R. McCann, P. DeRose, A. Doan, and R. Ramakrishnan. SLIC: On-the-fly extraction and querying of web data. Technical Report TR-1558, Computer Sciences Department, University of Wisconsin-Madison, 2006.

[8] D. Burdick, P. Deshpande, T. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP over uncertain and imprecise data. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2005.

[9] S. Prabhakar. Tutorial on probabilistic queries and uncertain data. In *Proceedings of Internation Conference on Management of Data (COMAD)*, 2005.

[10] O. Benjelloun, A. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[11] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 551–562, San Diego, California, June 2003.

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

[13] D. Dey and S. Sarkar. A probabilistic relational model and algebra. *ACM Transactions of Database Systems*, 21(3):339–369, 1996.

[14] L. Lakshmanan, N. Leone, R. Ross, and V. Subrahmanina. Probview: A flexible probabilistic database system. *ACM Transactions on Database Systems*, 22(3):419–469, 1997.

[15] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *Proceedings of 23rd International Conference on Data Engineering (ICDE)*, 2007.

[16] L. Antova, C. Koch, and D. Olteanu. 10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information. In *Proceedings of 23rd International Conference on Data Engineering (ICDE)*, 2007.

[17] 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 ACM Special Interest Group on Management Of Data (SIGMOD)*, 2005.

[18] A. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *Proceedings IEEE Internation Conference on Data Engineering (ICDE)*, 2006.

[19] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *Proceedings of ACM Special Interest Group on Management Of Data (SIGMOD)*, pages 73–84, 2006.

[20] D. Pfoser and C.S. Jensen. Capturing the uncertainty of moving-objects representations. In *Proceedings of the Scientific and Statistical Database Management Conference (SSDBM)*, pages 123–132, 1999.

[21] A. Yazici, A. Soysal, B. Buckles, and F. Petry. Uncertainty in a nested relational database model. *Elsevier Data and Knowledge Engineering*, 30, 1999.

[22] A. Nierman and H. V. Jagadish. ProTDB: Probabilistic Data in XML. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2002.

[23] E. Hung, L. Getoor, and V. S. Subrahmanian. PXML: A probabilistic semistructured data model and algebra. In *Proceedings IEEE Internation Conference on Data Engineering (ICDE)*, 2003.

[24] J. Galindo, A. Urrutia, and M. Piattini. *Fuzzy Databases: Modeling, Design, and Implementation.* Idea Group Publishing, 2006.

[25] M. Boughanem, F. Crestani, and G. Pasi. Management of uncertainty and imprecision in multimedia information systems: Introducing this special issue. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 11(1):1–4, 2003.

[26] P. Bosc and O. Pivert. About projection-selection-join queries addressed to possibilistic relational databases. *IEEE Transactions on Fuzzy Systems*, 13(1), 2005.

[27] B. Boss and S. Helmer. Index structures for efficiently accessing fuzzy data including cost models and measurements. *Fuzzy Sets and Systems*, 108(1), 1999.

[28] S. Helmer. Evaluating different approaches for indexing fuzzy sets. *Fuzzy Sets and Systems*, 140(1), 2003.

[29] P. Bosc and M. Galibourg. Indexing principles for a fuzzy data base. *Information Systems*, 14(6), 1989.

[30] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2004.

[31] V. Ljosa and A. Singh. APLA: Indexing arbitrary probability distributions. In *Proceedings of 23rd International Conference on Data Engineering (ICDE)*, 2007.

[32] C. Böhm, A. Pryakhin, and M. Schubert. The gauss-tree: Efficient object identification in databases of probabilistic feature vectors. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2006.

[33] A. Faradjian, J. Gehrke, and P. Bonnet. GADT: A probability space ADT for representing and querying physical world. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2002.

[34] H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *Proceedings IEEE Internation Conference on Data Engineering (ICDE)*, 1991.

[35] D. Pfoser and C. Jensen. Incremental join of time-oriented data. In *Proceedings of the Scientific and Statistical Database Management Conference (SSDBM)*, 1999.

[36] M. Soo, R. Snodgrass, and C. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings IEEE Internation Conference on Data Engineering (ICDE)*, 1994.

[37] D. Zhang, V. Tsotras, and B. Seeger. Efficient temporal join processing using indicies. In *Proceedings IEEE Internation Conference on Data Engineering (ICDE)*, 2002.

[38] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2004.

[39] C. Faloutsos. Signature files. In *Information Retrieval: Data Structures & Algorithms*, pages 44–65. Prentice-Hall, 1992.

[40] N. Mamoulis, D. Cheung, and W. Lian. Similarity search in sets and categorical data using signature tree. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2003.

[41] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2003.

[42] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 1996.

[43] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *Proceedings of 24th International Conference on Data Engineering (ICDE)*, 2008.

[44] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, volume 2, pages 125–142, March 1990.

[45] Orion: A database system for managing uncertain data. `http://orion.cs.purdue.edu/`, 2008.

[46] T. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[47] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *Proceedings of ACM 15th Conference on Information and Knowledge Management (CIKM)*, 2006.

[48] F. Pereira, N. Tishby, and L. Lee. Distributional clustering of english words. In *Meeting of the Association for Computational Linguistics*, 1993.

[49] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[50] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of ACM Symposium on Principles of Database Systems*, 2001.

[51] I. Ilyas, W. Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2003.

[52] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[53] K. Kummamuru, R. Lotlikar, S. Roy, K. Singal, and R. Krishnapuram. A hierarchical monothetic document clustering algorithm for summarization and browsing search results. In *Proceedings of the 13th international conference on World Wide Web*, 2004.

[54] C. Oh, K. Honda, and H. Ichihashi. Fuzzy clustering for categorical multivariate data. In *IFSA World Congress and 20th NAFIPS International Conference*, 2001.

[55] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing uncertain categorical data. In *Proceedings of 23rd International Conference on Data Engineering (ICDE)*, 2007.

[56] Y. Qi, S. Singh, R. Shah, and S. Prabhakar. Indexing probabilistic nearest-neighbor threshold queries. In *Proceedings of Workshop on Management of Uncertain Data (MUD, VLDB)*, 2008.

[57] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Query selectivity estimation for uncertain data. In *Proceedings of 20th International Conference on Scientific and Statistical Database Management (SSDBM 2008)*, 2008.

[58] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Querying imprecise data in moving object databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(7), 2004.

[59] Joseph Y. Halpern. *Reasoning about Uncertainty*. The MIT Press, 2003.

[60] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, S. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *Proceedings of 23rd International Conference on Data Engineering (ICDE)*, 2008.

[61] Joseph E Conway. PL/R - R Procedural Language for PostgreSQL. `http://www.joeconway.com/plr/`, 2008.

[62] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *Proceedings of Very Large Databases Conference (VLDB Demo)*, 2005.

[63] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. Orion 2.0: Native support for uncertain data. In *Proceedings of Special Interest Group on Management of Data (SIGMOD Demo)*, 2008.

[64] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. The orion uncertain data management system. In *Proceedings of the 14th International Conference on Management of Data (COMAD Demo)*, 2008.

VITA

VITA

Sarvjeet Singh was a Ph.D. student in the Department of Computer Science at Purdue University since fall of 2003. He research interests lie in problems related to uncertain data management, privacy and security in databases and artificial intelligence.

Prior to joining Purdue, Sarvjeet obtained his B.Tech. in Computer Science and Engineering from the Indian Institute of Technology (IIT), Mumbai, India. During his B.Tech., he worked on various projects in the areas of artificial intelligence, databases and data mining. For his B.Tech. final project, he developed and implemented a multilingual and meaning-based search engine, with the goal of eliminating the language barrier and improving the recall and precision of current keyword search engines. During summer of 2001, he worked as a research associate at the Center of Studies in Resources Engineering (CSRE), IIT Bombay, where he developed and implemented algorithms for registration and analysis of satellite images. He received his M.S. in Computer Science from Purdue University in May 2005. He worked as summer intern at Amazon.com in the summer of 2005, where he developed a high performance fault-tolerant business application utilizing distributed caching and databases. In 2006, he worked as graduate research professional in the Rosen Center for Advanced Computing at Purdue, where he developed mobile applications for deployment in intelligent environments using technologies such as J2ME, and conducted research on issues related to communication technologies such as RFID and Bluetooth. He worked at Google as an intern from June to September 2008, where he proposed and implemented a solution for automatic schema change detection and reconciliation of constantly evolving data sources. He graduated in May 2009 from Purdue University with a Ph.D. degree in computer science.