Query Processing on Uncertain Data

By Tingjian Ge

B.E., Tsinghua University, 1994

M.S., University of California, Davis, 1998

M.S., Brown University, 2006

A Dissertation Submitted in Partial Fulfillment of the

Requirements for the Degree of Doctor of Philosophy

in the Computer Science Department at Brown University

Providence, Rhode Island

May, 2009

This dissertation by Tingjian Ge is accepted in its present form
by the Computer Science Department as satisfying the dissertation
requirement for the degree of Doctor of Philosophy.

Date_____ _____
Stan Zdonik, Advisor

Recommended to the Graduate Council

Date_____ _____
Ugur Cetintemel, Reader

Date_____ _____
Eli Upfal, Reader

Approved by the Graduate Council

Date_____ _____
Dean of the Graduate School

Vita

Tingjian Ge was born on May 12th, 1972 in the beautiful city of Dalian, China. He received his Bachelor's degree in Computer Science from Tsinghua University, Beijing in 1994. In 1998, he received a Sc.M. degree in Computer Science from University of California, at Davis. After graduation from UC Davis, he worked as a Software Engineer at Informix Software, and then at IBM for a total of six years. He then joined the graduate program at Brown University, where he received his Sc.M. in Computer Science at Brown University, in 2006. He completed his Ph.D. in Computer Science at Brown University in 2009. He published six research papers as the first-author in recent three years at top-tier database conferences *ACM SIGMOD International Conference on Management of Data*, *International Conference on Very Large Data Bases* (*VLDB*) and *International Conference on Data Engineering* (*ICDE*).

ACKNOWLEDGMENTS

I owe my deepest gratitude to my advisor Stan Zdonik. My Ph.D. journey would have gone nowhere without his constant support, encouragements, and advices. Stan has helped me a great deal in presenting ideas clearly and succinctly. He has always been a great mentor who holds belief in me and encourages me to try any interesting ideas. Stan's technical insights benefited my thesis work a lot. My determination to work in the academia has so much to do with Stan's generous encouragements and advices.

I also wish to express sincere appreciation to Ugur Cetintemel. Ugur is simply always helpful. The very first graduate database class I took at Brown was with Ugur. From enlightening technical discussions, to inviting outside researchers to give a talk to us, Ugur contributed a lot to the well-being of my research work, as well as to improving my presentation skills.

My sincere thanks to Eli Upfal. Eli made me a big fan of probability. His book, "Probability and Computing: Randomized Algorithms and Probabilistic Analysis", has been in the reference list of nearly every paper of mine. This has proven to be the most useful mathematical tool in my research. I have either taken or audited as many as six of Eli's classes.

I am very grateful to Ihab Ilyas at University of Waterloo. Ihab's talk at Brown on top-k queries on uncertain data inspired me to work on this topic and eventually I published a paper at SIGMOD 2009, which is one of the key elements of this thesis. Ihab has also given me some valuable advices on my research, on my job application, and on my future career.

My gratitude also goes to Sam Madden at MIT. Sam has been helpful since the days I worked on the C-Store project. Recently, we worked on the SIGMOD paper together and had fruitful discussions about using his CarTel project dataset to perform the experiments. I learned a lot from Sam, not only the technical aspects, but also the presentation skills in a paper and in a research statement.

I am so much thankful to all the professors at Brown from whom I took classes. In particular, Anna Lysyanskaya's cryptography class and reading group drew so much of my interests that they give me

the foundation of my research on database security. So far, three of my papers on security are all based on the knowledge and techniques that I learned from Anna. I also have a great interest in the theory of distributed computing. For that reason, I learned a lot from Maurice Herlihy. Two of my research papers use techniques and theories I learned from Maurice's classes: skip-lists and the wait-free property. I am also grateful to Claire Mathieu, Roberto Tamassia, Franco Preparata, Shriram Krishnamurthi, and professors in the Applied Math department: Donald McClure and Stuart Geman. They had a great impact on my views and my research toolbox.

I can never overlook the effects of the education that I received from UC Davis during my Master's study. It is the main reason and inspiration that prompted me to pursue my Ph.D. study at Brown, even after six years in the industry. I can't forget the hours I spent in the library working on the problem sets from Dan Gusfield's advanced algorithm class. At that moment, I felt that I started to understand the meaning of computer science. Phil Rogaway's Introduction to Computer Science Theory class also made me appreciate the beauty of this field. Biswan Mukherjee's Network Performance Analysis class brought me more confidence in doing research. My work with Ron Olsson gave me understandings on concurrent computing.

The database group at Brown University has been a nurturing and stimulating environment. I am grateful to every member of the group.

This thesis would not have been a reality without the support of my family. First and foremost, my most heartfelt acknowledgment goes to my wife Li Li. Her constant support, endurance and sacrifice for me to pursue my dream will forever be my source of gratitude. My papers would not have been as good as they are today without Li's involvement in hearing my ideas, reading the papers, and commenting on them. Our upcoming son, Adam Ge, has certainly created a lot of motivations for me to finish my thesis. A penultimate thank-you must also go to my wonderful parents. For all the things they give me and their sacrifices without any hesitation or complaints, they deserve far more credit than I can ever give them.

(Funding for my graduate studies has been provided by a fellowship from Brown University and by the NSF, under the grants IIS-0086057 and IIS-0325838 given to Stan Zdonik.)

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

The need to manage uncertain data arises in many applications. Some examples include data cleaning, data integration, data extraction, sensor networks, pervasive computing, and scientific data management. For example, acoustic sensors (e.g., microphones) are often used to detect the presence of objects. Due to the nature of acoustic sensing, detections produced by microphones are often ambiguous, with an object possibly being at one of several locations. A common approach for storing such sensor data is to produce one record for each of the possible object locations, and assign a confidence (i.e., probability of existence in a table) to each record.

In the remainder of this chapter, we first discuss in more details about recent applications that require uncertain data management. Following that, we classify the types of uncertain data in database systems. We then present the possible world semantics that is commonly used in this context. Finally, for clarity, we give an overview of some open problems in this area and a brief highlight of our solutions to these problems.

1.1 Applications Where a DBMS Needs to Manage Uncertain Data

Managing large uncertain data repositories becomes an important and timely problem, with the explosion of the automatically generated data, inferred data, and data-by-the-masses in real systems. All these data are full of noise, missing values, errors and conflicts. Machine learning research has been trying to solve this very problem for decades. The explosion of the automatically generated data, inferred data, and data-by-the-masses in real systems requires a DBMS to efficiently handle large amount of data that has uncertainty. Here are some examples:

- *Sensor networks* can generate gigabytes of data every second, while sensor data are known to be low quality, because of the interference, noise, battery, etc.

- *Information extraction* systems automatically extract and classify entities, relationships and their attributes from web pages, where the extractor and classifier generate errors.

- *Data Integration* systems automatically try to infer schema mapping and record linkage from different data sources, which may result in mistakes in the integrated data.

- *Scientific databases* often have data that is imprecise in nature, due to the limitation of the instruments and the algorithms that derive the data. A simple example is that in astronomical databases, observed star locations are usually associated with error bars that specify a range of possible values.

- Lastly, *Social Networks* generate data by the masses, whose data is prone to be noisy and conflicting.

We illustrate using an example from an information extraction system. The Purple Sox [61] system at Yahoo! Research focuses on technologies to extract and manage structured information from the Web related to a specific community. An example is the DBLife system [27] that aggregates structured information about the database community from data on the Web. The system extracts lists of database researchers together with structured, related information such as publications they have authored, their co-author relationships, talks they have given, their current affiliations, and their professional services. Although most researchers have a single affiliation, the extracted affiliations are not unique. This occurs because outdated/erroneous information is often present on the Web, and even if the extractor is operating on an up-to-date webpage, the difficulty of the extraction problem forces the extractors to produce many alternative extractions or risk missing valuable data. Thus, each Name contains several possible affiliations. One can think of Affiliation as being an attribute with uncertain values; or equivalently, one can think of each row as being a separate uncertain tuple. There are two constraints on this data: tuples with the same Name but different Affiliation are mutually exclusive; and tuples with different values of Name are independent. The professional services can be extracted from conference web pages, and are also imprecise: in this example, each record in the table is an independent extraction and assumed to be independent.

1.2 Classification of Uncertain Data

In the probabilistic databases literature, there are two types of data uncertainty: (1) tuple uncertainty and (2) attribute uncertainty.

Tuple ID	Soldier ID	Time	Location (x, y)	Score for Medical Needs	Conf.
T1	1	10:50	(10, 20)	49	0.4
T2	2	10:49	(10, 19)	60	0.4
T3	3	10:51	(9, 25)	110	0.4
T4	2	10:50	(10, 19)	80	0.3
T5	4	10:49	(12, 7)	56	1.0
T6	3	10:50	(9, 25)	58	0.5
T7	2	10:50	(11, 19)	125	0.3

Tuple Uncertainty

(a)

Object ID	Speed
28	Normal (90, 20)
11	Normal (62, 15)
72	Normal (78, 10)

distribution

Attribute Uncertainty

(b)

Figure 1.1: Illustrating two kinds of uncertain data: tuple uncertainty (a) and attribute uncertainty (b). The last column of (a) (Conf., i.e., confidence) indicates the probability that the tuple exists in the table. The highlighted green tuples are mutually exclusive (i.e., at most one of them can be true).

In tuple uncertainty, a probability number (sometimes called *confidence*) is associated with each tuple. An example is shown in Figure 1.1(a). Figure 1.1(a) is from an application in which various sensors are embedded in the uniforms of soldiers in a battle field. The sensors send out detections of the medical conditions of the soldier that wears the uniform. The second to last column is a score that indicates how much medical attention this soldier needs. The higher the score, the more urgent it is to send medical resources to this soldier. The last column (Conf.) is the probability that the tuple exists in the table. We may also specify mutual exclusion rules, which indicate that at most one of a set of tuples can exist in the table. In this way, we can encode a discrete PMF (probability mass function) by a set of mutually exclusive tuples. In more details, for a PMF $\{(v_1, p_1), (v_2, p_2), \ldots, (v_k, p_k)\}$, v_1 to v_k are values in a set of mutually exclusive tuples and p_1 to p_k are their probabilities. The sum of the probabilities is no more than 1. If the sum is less than 1, then with remaining probability, none of the mutually exclusive tuples exist in the table. In the example in Figure 1.1(a), the three highlighted tuples in green (T2, T4, and T7) are mutually exclusive. They are detections of the same soldier (same Soldier ID) at around the same time, and hence at most one of them can have the correct score.

Clearly, the tuple uncertainty model can be considered as a generalization of the data model without uncertainty, in which each tuple has probability one, and there are no mutual exclusion rules.

The second type of uncertainty is called *attribute uncertainty*. In this case, an attribute is uncertain and we model each value of the attribute as a probabilistic distribution. In the example of Figure 1.1(b),

the measurements of the *Speed* attribute can have errors and we model each speed value by a normal distribution. This is in contrast with the traditional deterministic model in which each value of an attribute is a fixed scalar value. Attribute uncertainty may also be considered as a generalization of the data model without uncertainty, in which each value in an attribute is some value with probability one (i.e., a discrete distribution).

SELECT *ObjectID, Speed* FROM *table*
WHERE *Speed* > 78

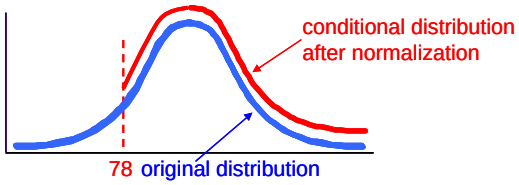
Result?

attribute uncertainty

tuple uncertainty

Object ID	Speed	Prob.
28	?	0.95
11	?	0.001
72	?	0.5

(a)



(b)

Figure 1.2: Illustrating tuple uncertainty and attribute uncertainty in a query result. We issue the query in (a) to the uncertain table in Fig. 1(b). Each of the three tuples has a non-zero probability to be in the result – this is tuple uncertainty (last column in (a)). The "Speed" in the result has attribute uncertainty – a conditional distribution shown in (b).

Not only do the two kinds of uncertainty exist in the source data, but they also exist in the query result. Let us look at an example.

We take a simple table that has attribute uncertainty as shown in Figure 1.1(b). We then issue a query as in Figure 1.2(a). What would the result be? Each of the three tuples has a non-zero probability to satisfy the predicate "Speed > 78". For example, the first tuple's Speed attribute has a normal distribution with mean 90 and variance 20, and thus has a high probability (say, 0.95) satisfying the predicate. The second tuple, on the other hand, has a normal distribution with a low mean (62) and has a tiny probability (say, 0.001) satisfying the predicate. Thus, we have tuple uncertainty in the query result (last column in Figure 1.2(a)).

Now about the selected "Speed" attribute in the result set? We know that only if the Speed is above 78 should the tuple be in the result at all. Hence, we can reason that the Speed attribute in the result should not be in its original form, but rather, a conditional distribution (conditioned on the predicate being true) based on the original distribution. We illustrate this in Figure 1.2(b), which shows the example for the first result tuple. We cut off the original distribution Normal (90, 20) at the value 78,

and only take the right side of the curve. Then, we need to normalize it (by multiplying a constant factor) so that the function still integrates to 1, as a probability density function. We can see that the Speed attribute in the result is still distributions, and we have attribute uncertainty in the result.

1.3 The Possible World Semantics

Possible world	Prob.	Top-2	Possible world	Prob.	Top-2
W1 = {T1, T2, T3, T5}	0.064	T3, T2	W10 = {T4, T5, T6}	0.09	T4, T6
W2 = {T2, T3, T5}	0.096	T3, T2	W11 = {T1, T4, T5}	0.012	T4, T5
W3 = {T1, T2, T5, T6}	0.08	T2, T6	W12 = {T4, T5}	0.018	T4, T5
W4 = {T2, T5, T6}	0.12	T2, T6	W13 = {T1, T3, T5, T7}	0.048	T7, T3
W5 = {T1, T2, T5}	0.016	T2, T5	W14 = {T3, T5, T7}	0.072	T7, T3
W6 = {T2, T5}	0.024	T2, T5	W15 = {T1, T5, T6, T7}	0.06	T7, T6
W7 = {T1, T3, T4, T5}	0.048	T3, T4	W16 = {T5, T6, T7}	0.09	T7, T6
W8 = {T3, T4, T5}	0.072	T3, T4	W17 = {T1, T5, T7}	0.012	T7, T5
W9 = {T1, T4, T5, T6}	0.06	T4, T6	W18 = {T5, T7}	0.018	T7, T5

Figure 1.3: Illustrating the possible world semantics. The uncertain table in Figure 1(a) has 18 possible worlds (W1 to W18). Each possible world is deterministic and has a fixed set of tuples (column 1). Each world also has a probability (column 2) that it is indeed the real world. The third column shows the results of a top-2 query based on the score attribute in the table. Overall, for the original uncertain table, the result is T2, T6 with probability 0.2 (i.e., the sum of the probability of W3 and W4).

For uncertain data management, the possible world semantics are commonly used. An uncertain relation may have many possible worlds. Each possible world is a deterministic world, and has a certain probability being the real world. Thus, we can think of an uncertain database as an aggregate of all the deterministic possible worlds. Query semantics under each deterministic possible world is well-known. Consequently, the total probability of all the possible worlds that bear some result tuple is the probability of that result tuple.

We now look at an example. Figure 1.3 lists the eighteen possible worlds of the uncertain table shown in Figure 1.1(a). We show the set of tuples in each world and the probability of each world in the first two columns of Figure 1.3. Note that Figure 1.3 respects the mutual exclusion rules of the uncertain table. For example, no world has more than one tuple from {T2, T4, T7}. Now suppose we want to answer a query that asks for top-2 tuples based on the score attribute in Figure 1.1(a). We can easily obtain the result for each possible world, shown in the third column of Figure 1.3. Then the final step is to "assemble" these results from the possible worlds, by summing on their probabilities. For example, the combination T2, T6 is and only is in W3 and W4, and therefore, T2,

T6 is in the result of the original uncertain table with probability Pr(W3) + Pr(W4) = 0.2. Repeating this for each possible result, we arrive at the semantics for the uncertain database.

Note that although this gives an intuitive semantics (or what to expect) of the query results, in general it is highly inefficient to try to answer a query by enumerating all possible worlds. There are an exponential number of worlds and it is too costly to evaluate a query this way. Clearly, efficient query processing algorithms need to be developed.

1.4 The Open Problems in Managing Uncertain Data

Managing uncertain data is an area that has drawn considerably attention lately due to its wide applications. As we discussed, there are roughly two kinds of uncertainty in the database context: *attribute uncertainty* and *tuple uncertainty*. We start with the attribute uncertainty as commonly seen in sensor and scientific data management. There are many examples including sensor readings (e.g., temperature) and GPS location data [GPS08]. In many cases, the uncertainty increases with time as the readings become outdated.

1.4.1 Problem 1: Operations on Uncertain Data

In previous work (e.g., [CK03]), in the context of sensor networks, uncertain data is modeled as a continuous PDF (probability density function). Essentially each data value is a PDF describing its distribution. Queries produce results that are also uncertain and the resulting PDF is a function of the input PDF's. For example, in [CK03], to perform a simple "addition" on two uncertain values' PDFs, a convolution of the form $\int_{\max\{l_1, x-u_2\}}^{\min\{u_1, x-l_2\}} f_1(y) f_2(x-y) dy$ must be performed (resulting in a function on x), where f_1, l_1, and u_1 are the first value's PDF, lower bound, and upper bound, respectively, and f_2, l_2, and u_2 are the same for the second value. For adding more values (e.g., for SUM or AVG), the convolution is repeated n times, where n is the number of values being aggregated, to get the final distribution function. Scientific databases are typically huge (frequently terabytes) and the operations that they must support are complex. It is easy to see that this approach would not scale in our context: the intermediate PDFs are too costly to compute. Even if one applies numerical methods to approximate the intermediate PDFs, the expense can still grow arbitrarily [S01].

1.4.2 Problem 2: Modeling and Processing Correlated Uncertain Data

We first look at an example of uncertain data. Let's imagine a temperature sensor whose readings follow a Gaussian distribution with a known variance of one degree, and these sensor readings are

stored in a database. Imagine this database contains a tuple T1 which indicates that on Sunday the temperature was 79 degrees. Now someone queries the database for all rows with temperature greater or equal to 80. Should R1 be included in the result set? In a traditional database, the answer would be no. However, there is a 16% chance that the temperature represented by T1 was actually over 80 degrees because of the uncertainty in the sensor. Databases for scientific applications need to be able to handle this uncertainty by propagating it to query results.

The uncertainty problem is further complicated when the uncertainty between different values is correlated. Imagine that the database described above, when queried for temperatures above 80, will tell me that the probability that T1 is in the result set is 16%. Now imagine that the database contains another tuple T2 which indicates that on Monday the temperature was also 79 degrees. Now the database is queried for two consecutive days in which the temperature was above 80 degrees. Since T1 and T2 both have a probability of .16 of representing a temperature over 80 degrees, the probability of the result set containing the combination (T1, T2) is 0.16*0.16 = 0.0256, assuming the measurements are independent. However, the measurements are not independent; they come from the same sensor. What if the sensor is precise but inaccurate and all measurements have the same error due to the sensor itself? In that case, the database needs to handle this correlated uncertainty and report that the combination (T1, T2) should actually be in the result set with probability 0.16 rather than probability 0.0256.

Ignoring correlation of uncertain data in databases for the simplicity of representation and query processing is often unfounded and renders the results of queries wrong and useless. This is loosely analogous to previous work on query optimization, in which one has to consider attribute value correlation for selectivity estimation [PI97]. Compact model representation and efficient query processing are key ingredients of practical systems that handle uncertain data. These two elements are needed for correlated uncertain data.

1.4.3 Problem 3: Answering Queries of Uncertain Data in the Future

Answering queries about data in the future (i.e., prediction or forecasting) is a new direction in data management. Although the existing data may be deterministic, forecasting is over the uncertain future data. In other words, the result of a forecasting query is uncertain. Time series is the dominating data type in this domain, although it can be other data types. Scientific, financial, and business applications rely on time series data [WG93, WS01]. Decision making often requires forecasting over time series

data at different time scales. The following three example areas illustrate (1) short-, (2) medium-, and (3) long-term forecasting requirements respectively.

1. *Scheduling:* Forecasts of the level of demand for various products are an essential input to near-term scheduling of production, transportation, and personnel.
2. *Acquiring Resources:* Forecasting is needed to determine future resource requirements in order to plan for acquisition lead times that could span several months.
3. *Determining resource requirements:* Forecasts of financial, human, and technological requirements are helpful for determining what resources an organization will need in the long-term.

In these applications, the amount of data is often very large. Consider the time series of trades and quotes (called ticks). Stock quotes arrive every second. Financial analysts want to predict stock prices minutes ahead, hours ahead, days ahead, months ahead, or sometimes years ahead. A simple example of a forecasting query is the following:

SELECT * **FROM** ticks
WHERE symbol = "IBM" and time = **NOW** + 1 day

Clearly, excessive granularity of data is unnecessary and inefficient or even impractical for a given prediction interval. For example, to predict the stock price of some company one year from now, it is wise to use a history length of a certain number of years (say, 20 years). Too short a history may give a partial picture of the evolution of the stock data, thus making the prediction result inaccurate [MW98]. On the other hand, too long a history length may not offer more useful information for the prediction, and sometimes may even complicate and disturb the model building [YS00], thereby, also reducing accuracy.

A history length of 20 years with one tick per-second has 20*365*24*3600 = 630,720,000 values! A typical model selection and building process is expensive, and using this large number of data points is impractical. In fact, even for predicting 15 days from now (using, say, a 12-month history), the required history length would still be prohibitively large with over 30 million values.

1.4.4 Problem 4: Top-*k* Queries on Uncertain Data

The need to manage uncertain data arises in many applications. Some examples include data cleaning, data integration, sensor networks, pervasive computing, and scientific data management. In the mean time, top-k (i.e., ranking) queries have proved to be useful. Often, a query returns a large number of

result tuples. Users can choose the top ranked few tuples to look at, according to some scoring function that indicates their preference.

Consequently, answering top-k queries in uncertain databases has drawn some attention lately. The complication due to the interaction of scores and probabilities of tuples makes the semantics unclear. Thus, the very first problem is to define the semantics of top-k queries when the data is uncertain.

Recently, there has been some work on the semantics of top-k queries on uncertain data, starting from the inspiring work of Soliman, Ilyas, and Chang [SI07]. The proposed semantics roughly fall into two categories: (1) returning k tuples that can co-exist in a possible world (i.e., that must follow the generation rules) or (2) returning tuples according to their own marginal distribution in top-k results (e.g., the probability that a tuple is top-k or at a specific rank in all possible worlds). For example, the U-Topk [SI07] definition belongs to category (1) while the U-kRanks [SI07] and PT-k [HP08] definitions belong to (2). We build on this and propose an extension of the category (1) semantics.

In category (1) semantics, U-Topk chooses a k tuple vector based on its probability only. However, we observe three facts:

- Although a k-tuple vector has the highest probability p being in the top-k, p itself can be rather small (an obvious upper bound is that all those k tuples must all appear), or it is not much bigger than the probability of other vectors being top-k.
- The score distribution of the tuples is usually independent of the distribution of probability values of tuples.
- U-Topk does not take into consideration the distribution of the scores of all possible top-k tuple vectors.

As a result of the above three facts, the total scores of a U-Topk vector can be rather atypical. We note that this problem with U-Topk can be worse when k is bigger (i.e., $k > 2$). In other words, it is more likely to occur that U-Topk returns a vector with an atypical score for bigger k values. This is because for a specific k-tuple vector to be U-Topk, all the k uncertain tuples must appear in the first place, lowering the probability and increasing the likelihood that its score is atypical. More specifically, due to the "curse of dimensionality", no top-k vector likely dominates many possible worlds (or has a significant probability). Now suppose we arbitrarily increase the score of a tuple that is not in the most probable top-k vector, U-Topk result can be arbitrarily atypical. This dilemma is analogous to the "typical set" concept in information theory [CT91].

Let us step back and examine what the issue really is. The complete result of a top-k query on uncertain data, in fact, is a joint distribution on k-tuple vectors. If one were able to return such a joint distribution as the result, all available information would be there. Unfortunately, it is too expensive to compute, as well as to describe and return such a joint distribution as the result. All existing definitions try to provide some of the most important information of such a distribution. Category (1) and (2) definitions are useful in different situations. Category (1) definitions are needed for scenarios that seek "compatible" k tuples (e.g., further inference on the whole set of k tuples are performed, as in our examples). However, as we have observed, by simple selection of the highest probability, U-Topk may pick a k-tuple vector that is highly atypical in scores.

1.5 Overview of Our Results

1.5.1 Solution 1: A Discrete Approach to Modeling and Query Processing

We propose a simpler, scalable, and discrete treatment. Even after the discretization of input values, the cost of computing purely accurate result distributions can still be prohibitive. Consequently, it is imperative to have a good metric that tells us how far the result distribution is from the "ideal" distribution. We resort to a well-known metric from statistics, namely, variation distance [MU05]. It measures the "distance" of two discrete distributions. In order to use this metric, we propose a way to map continuous value intervals to discrete points in the state space. The "ideal" distribution is defined as the distribution one would get if given unlimited computing resources.

We give an algorithm called SERP (Statistical sampling for Equidepth Result distribution with a Provable error-bound) that has a provable upper bound on the variation distance between its result distribution and the "ideal" one. SERP contains a parameter that indicates the granularity of the discretization that balances efficiency and accuracy. SERP is a framework that can process general query types, and it is essentially based on Monte Carlo randomized algorithms.

For certain operations, such as those aggregating a large number of values (e.g., summing or averaging a few million uncertain values), it may be an unnecessary burden for the database system to compute a full distribution of the result. As the aggregation is performed on many uncertain values, the user is likely more concerned with a statistical summary of the result, such as the expected value and variance. Individual possible values or a full distribution is less interesting. Moreover, the database system may be able to compute "accurate" statistical information much more efficiently than trying to compute an approximated full distribution. For this reason, we propose the "statistical" mode of a value, which is comprised of the following components: expected value (E), variance (Var), an upper

bound (UB), the probability (p1) that the value is above this upper bound, a lower bound (LB), and the probability (p2) that the value is below this lower bound. The user may request the result to be in this statistical mode only. We have also studied predicate evaluation strategies using inequalities.

1.5.2 Solution 2: A Special Join Algorithm (S-Join) in the SERP Framework

One of the drawbacks of performing query evaluation through sampling is that one query on deterministic data can turn into one thousand queries on probabilistic data, which could be prohibitively expensive. JOIN is one of the costliest database operators. In the case where the JOIN attribute is uncertain, it may be necessary to sample that attribute and, in effect, perform thousands of JOINs, a daunting task. We develop a specialized JOIN algorithm under the Monte Carlo query evaluation that mitigates this problem. Our algorithm takes advantage of the structure of the sampling problem to provide a significant speedup over running a standard JOIN algorithm over and over. The algorithm is a modified sort-merge-join and we call it the S-Join algorithm.

The intuition behind the S-Join algorithm is that given a series of uncertain values, the order of samples drawn from their distributions should be similar to the order of their expected values since the correction parts are typically small compared to the gap between the original parts of any two values. Thus, once we sort the tuples according to their expected values and draw samples in that order, the samples themselves should be almost sorted (called pseudo-sorted). Sorting a pseudo-sorted list is much cheaper than a complete sort. If most of the values to be sorted are already in sorted order, insertion sort has a linear run-time.

1.5.3 Solution 3: Modeling and Query Processing of Correlated Uncertain Data

The first question we study is what the query semantics should be. Possible world semantics has been extensively studied in the probabilistic database literature (e.g., [G06, DS04]). However, it is unclear how one can apply it to the correlated continuous attribute uncertainty model. We present two ways to specify the query semantics (integral-based and sampling-based).

Compact model representation and efficient query processing are key ingredients of practical systems that handle uncertain data. To that end, *chunking* is typically employed in array database systems for efficient I/O [SS94]. We propose piecewise probabilistic graphical models (e.g., Markov Random Fields) [J98] with a slightly modified chunking scheme and adopt Markov Chain Monte Carlo (MCMC) algorithms to perform inference on these graphical models as a general query evaluation method.

We next observe some interesting properties of the entropy [CT91] of the probability distribution of result tuples and its relationship to the quality of the result. This relationship provides hints as to when a result set is relatively stable and Monte Carlo sampling can cease. These properties can also be exploited to selectively stop query evaluation for certain result tuples and only run more Monte Carlo rounds for those tuples that require more time. This optimization can be achieved with suitable lineage information of the result tuples.

1.5.4 Solution 4: A-tree: A New Data Structure to Model Correlated Multidimensional Array Data

In this work, we argue that by taking advantage of predictable and structured correlations of multidimensional data, we can provide a more efficient way of modeling and answering queries on large-scale array data. We propose a new data structure, called the A-tree (Array tree). The A-tree approach is based on the following interesting observation: data in a multi-dimensional array is usually correlated along some dimensions and the correlation is largely local. Thus, if we have to sacrifice precision by allowing approximate models, focusing on local correlation or using clustering is the best bet. An A-tree uses this fact and can automatically cluster data in a hierarchical manner. Within the clustering structure, the joint distributions are smaller scale and can be modeled efficiently.

There is a simple mapping from the graph structure of an A-tree (i.e., the storage model of an array) to its probabilistic graphical model. The graphical model of an A-tree is essentially a Bayesian Network. Physically, only the leaves of the tree-structured BN exist. The nodes (i.e., random variables) at upper levels are all derived from the leaves. Thus, the construction of an A-tree is bottom-up, yet the probabilistic inference (which is needed for processing queries [28, 31]) is top-down.

Because the graphical model has a natural correspondence with the physical spatial layout of the multidimensional array, probabilistic inference is very efficient by traversing the A-tree and following a logarithmic-length path directly to the needed cells of the array. We analyze and experimentally compare the performance, as well as modeling accuracy, with an alternative graphical model of a lattice-structure MRF. In this regard, A-tree behaves like a spatial index.

Sparse arrays are common for multidimensional arrays. An A-tree, by its nature, is a compact representation for sparse arrays. Missing subtrees correspond to empty regions of the array (i.e., cells

of a sparse array that have NULL values). We discuss its layout on disk. In this regard, A-tree is also a succinct storage structure.

Query processing is an integral part of a representation scheme. We study query processing techniques for A-trees. Specifically, the A-tree data structure facilitates an interesting optimization for COUNT, AVG, and SUM queries on arrays of arbitrary sparsity. We also study the problem of probabilistic inference for general queries.

1.5.5 Solution 5: Using Skip-lists in Answering Queries of Uncertain Future

For the prediction of a specified interval, we choose a subsequence embedded within the original time series as a "new" time series of a different "time granularity". In summary,

- We may use different "absolute history lengths" for different forecast intervals *f*.
- Given a history length *h(f)*, we determine the number of data points *n* to use for model building.

We use a skip list data structure [P90] to provide fast data access for different levels of granularity. In addition to supporting prediction, a skip list also supports searching (i.e., indexing). Each level of the skip list has a set of models (i.e., prediction functions) associated with it. We can also build models at the leaf level of a skip list to interpolate missing data values in the past. Note that the searching and interpolation aspects are straightforward and the focus of this paper is on prediction of various future intervals using data at different levels of the skip list.

The original skip list data structure is only meant to be in memory. To be scalable for large data sets, it needs to be stored on disk. We adopt it in our context and discuss its organization on the disk.

Different levels of a skip list have different data densities. For a given query interval f, as we discussed earlier, we can determine a proper history length h(f) to use and the number of subsequence data points n to use within h(f) for model building. Thus, n/h(f) gives a data density which we use to select a level of the skip list that has the closest density.

If characteristics of the workload are known, we can pre-build a set of models for prediction queries using our skip list technique. If the workload is unknown, we can build the models on the fly. We must also consider the maintenance costs for updating the pre-built models as new data comes in. It is worth noting that on-line performance will be improved using our skip-lists when we must either dynamically build models or frequently maintain (rebuild) the models under update.

We present a randomized algorithm called ChoosePMSet to select a set of models to pre-build subject to a maintenance cost constraint. This constraint is based on query interval workload information described as a PMF (Probability Mass Function). A prediction query is hence answered by picking the "closest" pre-built model (PM) to use. We measure how well the set of PM's "serves" the workload by computing the expected model distance of a prediction query. The PM for prediction queries are analogous to materialized views (MV) for traditional queries. The key difference is that an MV materializes the data tuples while a PM only "materializes" the parameters of a model (e.g., coefficients of a polynomial), which is highly compact.

Using PM's for query processing is more straightforward for point queries than for more complex query types. We discuss query processing techniques using PM's for interesting query types, namely, range queries, aggregations, and join queries. We avoid materializing future data points for efficiency.

1.5.6 Solution 6: Novel Semantics for Top-k query on Uncertain Data and Efficient Query Processing Algorithms Based on Dynamic Programming

The complete result of a top-k query on uncertain data, in fact, is a joint distribution on k-tuple vectors. If one were able to return such a joint distribution, it would represent a complete answer, and would provide users with a convenient representation of the tradeoff between probability and score from which they could select the results of interest. Unfortunately, a complete distribution is too expensive to compute, as well as to describe and return as the result. All existing definitions try to provide the most important information of such a distribution. Category (1) and (2) definitions are useful in different situations. Category (1) definitions are needed for scenarios that seek "compatible" k tuples (i.e., they can co-exist), which is required when, for instance, further inferences on the whole set of k tuples are performed, as in our examples. However, as we have observed, by simple selection of the highest probability, U-Topk may pick a k-tuple vector that has a highly atypical score. What we propose in this work is a simple two-fold solution:

(1) The application program can optionally retrieve the score distribution of top-k vectors at any granularity of precision (e.g., histograms of any bucket width).

(2) We propose a new definition c-Typical-Topk which returns c typical top-k tuple vectors according to the score distribution, where c is a parameter specified by queries. Intuitively, the actual top-k's score should be close to one of the c vectors' score.

We then address the computational challenge of obtaining the score distribution of top-k vectors and selecting c typical vectors. For the score distribution, we first give two simple and naive algorithms that either explore the state space to reach top-k tuple vectors (StateExpansion algorithm) or iterate through all k-tuple combinations within a bounded set of tuples (k-Combo algorithm). These two algorithms establish a baseline for comparisons. We then present our main algorithm which is based on dynamic programming and is much more efficient than the naive algorithms. The presentation of the main algorithm starts with the basic framework and is then extended to handle more complex and realistic scenarios, namely mutually exclusive tuples and score ties for tuples. Score ties are common when the score is based on an attribute that does not have many distinct values, e.g., year of publication, number of citations, or even non-numeric attributes [7]. Note that extending the semantics and algorithms to score ties (i.e., non-injective scoring functions) for uncertain data can be non-trivial [22] (because a single possible world can now have multiple top-k vectors) and is not dealt with in previous work. Once we obtain the score distribution of top-k, using ideas similar to [8], we apply a two-function recursive approach resulting in another efficient dynamic programming algorithm to select c typical vectors for c-Typical-Topk.

We conducted systematic experiments on a real dataset of road delays in the greater Boston area as measured by the CarTel project team [10, 14], as well as a synthetic dataset. Through the experiments, we verify our motivation, study the performance of our algorithms, and observe interesting behaviors of the results with different characteristics of data.

Chapter 2

A MONTE CARLO QUERY PROCESSING FRAMEWORK (SERP) AND S-JOIN

In this chapter, we describe in details (Section 2.1) on the algorithm we developed to answer an arbitrary query on uncertain data. The algorithm is called Statistical sampling for Equidepth Result distribution with Provable error-bounds, or SERP. SERP is essentially an Monte Carlo randomized algorithm. We also an alternative "Statistical Mode" for query results. In Section 2.2, we introduce a special efficient JOIN algorithm (S-JOIN) under the SERP framework.

2.1 SERP

2.1.1 Discrete Treatment of Imprecise Data

Propagating continuous PDFs across complex mathematical operations and large data sets can easily become intractable. Instead, we take a systematic and rigorous approach to the use of discrete PDFs for this purpose.

Consider the lifetime of an uncertain value in ASAP. It "flows" through a graph of mathematical or query operators, the output of one operator box is the input of another, and finally the output of the whole query graph is the result to the end user. We model an uncertain value as a general discrete probability density function. We first look at an intuitive and commonly used form of discretization. We choose a set of points in the value range (frequently they are equally spaced), and assign a probability value to each point. The probabilities add up to 1. Thus a distribution is modeled by a set of (v_i, p_i) pairs, indicating that the probability of the value being v_i is p_i.

Under this representation, we look into the problem of computing the output distribution of a primitive mathematical operator. For ease of presentation, we discuss the case of two uncertain input values and one output (i.e., a binary operator). This can be easily extended to the general case. More formally, suppose that one input is (v_{1i}, p_{1i}), and the other input is (v_{2i}, p_{2i}), with $i=\{1,\ldots, k\}$. We denote the binary operator as $\otimes$. We look at the complexity of computing the output distribution under the independence assumption of inputs (from different tuples), as followed by most work in

this area (e.g., [CK03] and the "x-tuples" in [BS06]). Note that the SERP algorithm that we will present does not have to use this assumption. Clearly the probability of the result being $v_{1i} \otimes v_{2j}\ (1 \le i, j \le k)$ is $p_{1i} \cdot p_{2j}$. In general, each $v_{1i} \otimes v_{2j}$ can be distinct, hence the cost of describing and computing the output distribution precisely is $O(k^2)$. In the same manner, if we perform the same binary operation n-1 times for n values (e.g., for SUM or AVG), the complexity of computing the output distribution precisely is $O(k^n)$, a prohibitive exponential cost for a large value n.

A standard way to handle this dilemma is to use some form of approximation and to have a systematic way of measuring how much precision we lose to gain the needed efficiency. Towards this end, we first give three simple, intuitive (and rather naive) heuristic algorithms for approximating the output distribution.

Perhaps the most intuitive and simple algorithm is to uniformly at random pick $O(\sqrt{k})$ pairs of (v_i, p_i) from each of the two inputs; iterating on all combinations of these pairs gives an O(k) cost for one operation. Doing this binary operation n-1 times on n values gives O(kn) cost. We call this algorithm RAND. Clearly we need a final normalization step to multiply the computed probabilities by a constant factor so that they add up to 1.

The next heuristic is a greedy algorithm. Observe that as each input has k pairs of (v_i, p_i), an exhaustive algorithm would compute the result for all k^2 combinations. However, not all combinations are "equal". If we only have the resources to compute k combinations, we tend to gain "more information" about the result distribution by picking the combinations that occur with higher probability. For a simple example, let two inputs of an addition operator be {(8, 0.8), (4, 0.2)} and {(10, 0.9), (6, 0.1)} respectively. Out of the four combinations, the one that has result value 8+10=18 and probability 0.8*0.9=0.72 has the highest probability (0.72) of occurrence. Computing it would give us the most "information" about the result, if we only had the resources to compute one combination. Thus, in this algorithm we greedily pick k combinations (out of k^2) that have the top probabilities. We can accomplish this without computing the probability of all k^2 combinations, by always maintaining an order of the (v_i, p_i) pairs sorted on p_i in a discrete PDF. Through a "merging" process which only inspects a subset of "top candidates", we can obtain the k top combinations without computing them all. We omit the details due to space constraints. We call this algorithm K-TOP.

In contrast, the third heuristic algorithm computes all k^2 combinations but sorts the result values and "condenses" them into k pairs of the form (v_i, p_i) in which v_i is the average of the i'th run of k contiguous values, and p_i is the sum of the probabilities of these k values'. This algorithm is very intuitive, although a bit more costly, with the cost $O(k^2 \log k)$ for two values and $O(n \cdot k^2 \log k)$ for n values. We call this algorithm CONDENSE.

2.1.2 The SERP Algorithm

2.1.2.1 A Different Way of Discretization

We propose to use a different form of discretization. We partition the value range of the continuous PDF into *k* intervals, such that for each interval *I*, it holds that $\int_{x \in I} f(x)dx = \frac{1}{k}$, where *f(x)* is the continuous PDF. In other words, each interval has overall probability *1/k*. Thus, a distribution is "described" by *k* contiguous intervals and can be succinctly represented as *k+1* values indicating the boundaries of the *k* intervals: $(v_0, v_1, \ldots, v_k)$, where $[v_i, v_{i+1})$ is the *i'th* interval. We assume a uniform distribution within an interval. This is reminiscent of "equidepth" histograms widely used in query optimizers, and reflects the idea that the exact distribution of "high density areas" is more important and should be given higher "resolution". However, note the important difference that each bucket of an *equidepth histogram* contains a number of *actual* column values, whereas an *equidepth distribution* specifies the PDF of one scalar entity (random variable). This representation is quite compact, only needing *k+1* values to describe a distribution. In contrast, the discretization scheme earlier requires both values and the associated probabilities.

2.1.2.2 A Weighted Sampling Method

We next propose a simple method that samples a random variable according to an arbitrary equidepth discrete PDF, as follows.

> *Input*: A discrete PDF: $(v_0, v_1, \ldots, v_k)$, in equidepth form.
> *Output*: A random point $v_0 \le s \le v_k$ that is a weighted sample according to the input discrete PDF.
> (1) Pick a number *i* uniformly at random from the set $\{0, 1, \ldots, k-1\}$.
> (2) Choose the output value *s* uniformly at random from the interval $[v_i, v_{i+1})$.

Theorem 2.1: *The weighted sampling algorithm WS indeed accomplishes the task: it returns a random sample weighted according to the input discrete PDF.* □

2.1.2.3 The SERP Algorithm

We now introduce the SERP algorithm which uses the WS algorithm for statistical sampling to compute the output distribution of a mathematical operator. We model the operator as "*n* input values and one output value" without loss of generality. For example, for SUM or AVG, the inputs may be *n* values in *n* tuples and the output is the result. The algorithm is shown in the text box.

In the algorithm, μ is a parameter that balances accuracy with performance, as we shall investigate in the theoretical analysis and the empirical study. Note that we model all inputs as uncertain. In reality, some input values can be certain. It is straightforward to extend the algorithm to the mixed case. Also note that from one execution on the *n* samples to the next, to be more efficient, we can share the query plan (i.e., the query is compiled only once, and executed many times for each loop). Further, among different executions, sub-results of parts of the query plan that only refer to data without uncertainty can be shared. Another key optimization is on I/O cost. The database engine can try to read the data from the disk only once, and incrementally carry out the multiple rounds of computation in parallel. It is easy to see that SERP is *scalable*. The cost is no more than a constant factor of that of the same operation on data *without uncertainty*, regardless of the number of tuples.

Input: n uncertain values as inputs to operator *Op*, each described as an equidepth discrete PDF.
Output: The result value distribution for operator *Op* applied to the *n* inputs as an equidepth discrete PDF with *k* intervals.

(1) **Repeat** the following steps $k \cdot \mu$ times, where *k* is the intended number of intervals of the result distribution and μ is a parameter to be determined later.

(2) For each of the *n* inputs, apply the WS algorithm to get a sample value. Let's say we get $s_1, s_2, \ldots, s_n$.

(3) Feed $s_1, s_2, \ldots, s_n$ as deterministic inputs into the operator *Op* and compute the output value *o*.

(4) **End Repeat** loop.

(5) Sort the output values obtained above as $o_1, o_2, \ldots, o_{k\mu} (where\ o_i \le o_{i+1})$.

(6) Get *k* contiguous value intervals, each containing μ output values. That is, the 1st interval contains $o_1, o_2, \ldots, o_\mu$, and the 2nd contains $o_{\mu+1}, o_{\mu+2}, \ldots, o_{2\mu}$, and so on. More precisely, let $v_0, v_1, \ldots, v_k$ be the boundaries of the *k* contiguous intervals, where $v_i = (o_{i\mu} + o_{i\mu+1})/2, (1 \le i \le k-1)$, and $v_0 = 2o_1 - o_2$, $v_k = 2o_{k\mu} - o_{k\mu-1}$.

(7) Return the *k* contiguous intervals above as the result distribution.

Note that SERP is similar in spirit to the classical Monte Carlo method [H70]. However, technically, SERP extends it in a nontrivial way. Through repeated sampling, the classical Monte Carlo method approximates some value (e.g., computes an integral) that is equal to the expectation of a random variable. By the law of large numbers, one can show the result converges to the true value. To the best of our knowledge, SERP is the first one that computes an equidepth result distribution (PDF)

which, as we further prove, has a bounded distance from an "ideal" PDF using the variation distance metric.

We stress that unlike the algorithms earlier, SERP works even if there is correlation between different inputs. We just need to carry out the sampling from the joint distribution. For example, if an array stores a 3-D image or temperatures in a space, it may be partitioned into "distribution chunks" such that cell values in each chunk exhibit high positive correlation. One can assign one cell in each chunk as the "leader", whose distribution represents that of the whole chunk. We know the differences between each cell and its leader. Thus, we only sample on the leader of each chunk, and derive other cell values.

2.1.3 A Metric on Judging Results and Provable Error Bounds of SERP

2.1.3.1 A Distance Metric and Its Adoption

We measure the distance between the discrete result PDF computed by some algorithm and an "ideal" one based on the same input distributions, but given as much computing resource as needed. We use a well-known distance metric: *variation distance*.

Definition 2.1 [MU06]**:** The *variation distance* between two distributions D_1 and D_2 (each being a PDF) on a countable state space S is given by $VD(D_1, D_2) = \frac{1}{2}\sum_{x \in S} | D_1(x) - D_2(x) |$

We first give some insights on the variation distance metric, as we will be using it for analysis and experiments.

Lemma 2.1 [MU06]: Consider two distributions D_1 and D_2. For a state x in the state space S, if $D_2(x) > D_1(x)$, then we say D_2 ***overflows*** at x (relative to D_1) by an amount of $D_2(x) - D_1(x)$. Likewise, if $D_2(x) < D_1(x)$, then we say D_2 ***underflows*** at x (relative to D_1) by an amount of $D_1(x) - D_2(x)$. We denote the total amount that D_2 overflows (and underflows, respectively) as P_{over} (and P_{under}, respectively). Then, $P_{over}=P_{under}=VD(D_1, D_2)$. □

2.1.3.2 A Provable Error-Bound of SERP

We are now ready to present a novel proof that SERP has a nice bound on the variation distance between its result distribution and the *ideal one*, even though we do not know the exact form of the ideal result distribution, nor do we make any assumption on how to obtain it.

Theorem 2.2: *In the SERP algorithm, let k and* μ *be parameters as described in the algorithm. Then, with probability at least* $1-k\cdot\left[\left(\frac{e^{2\delta}}{(1+2\delta)^{1+2\delta}}\right)^{\mu}+\left(\frac{e^{-2\delta}}{(1-2\delta)^{1-2\delta}}\right)^{\mu}\right]$*, the variation distance between the result distribution and the ideal one is no more than* $\delta\ (0<\delta<0.5)$.

Proof: Consider any *one* interval *I* of the *ideal* distribution. Define $k\cdot\mu$ random variables $X_i\ (1\le i\le k\mu)$:

$$X_i=\begin{cases}1 & \text{if the output from i'th repeat loop of SERP falls in } I\\ 0 & \text{if the output from i'th repeat loop of SERP is not in } I\end{cases}$$

Because *I* is an interval of the ideal distribution, from the definition of the equidepth partition, we have $\Pr(X_i=1)=\frac{1}{k}$, thus $E(X_i)=\frac{1}{k}$. Next define random variable $X=\sum_{i=1}^{k\mu}X_i$, indicating the number of result points that fall in *I*. From linearity of expectation, we have $E(X)=k\mu\cdot\frac{1}{k}=\mu$. As X is the sum of independent 0/1 random variables, we can apply Chernoff bounds [MU06] that for any $0<\delta<0.5$, we have

$$\Pr[X>(1+2\delta)\mu]<\left(\frac{e^{2\delta}}{(1+2\delta)^{1+2\delta}}\right)^{\mu}\qquad \Pr[X<(1-2\delta)\mu]<\left(\frac{e^{-2\delta}}{(1-2\delta)^{1-2\delta}}\right)^{\mu}$$

Then from union bound [MU06],

$$\Pr[X>(1+2\delta)\mu \text{ or } X<(1-2\delta)\mu]<\left(\frac{e^{2\delta}}{(1+2\delta)^{1+2\delta}}\right)^{\mu}+\left(\frac{e^{-2\delta}}{(1-2\delta)^{1-2\delta}}\right)^{\mu}$$

Now consider all *k* intervals and apply union bound again,

$$\Pr[\exists \text{interval st.}\ |X-\mu|>2\delta\mu]<k\cdot\left[\left(\frac{e^{2\delta}}{(1+2\delta)^{1+2\delta}}\right)^{\mu}+\left(\frac{e^{-2\delta}}{(1-2\delta)^{1-2\delta}}\right)^{\mu}\right]$$

Hence,

$$\Pr[\forall \text{interval},\ |X-\mu|\le 2\delta\mu]\ge 1-k\cdot\left[\left(\frac{e^{2\delta}}{(1+2\delta)^{1+2\delta}}\right)^{\mu}+\left(\frac{e^{-2\delta}}{(1-2\delta)^{1-2\delta}}\right)^{\mu}\right]$$

Thus, with probability at least $1-k\cdot\left[\left(\frac{e^{2\delta}}{(1+2\delta)^{1+2\delta}}\right)^{\mu}+\left(\frac{e^{-2\delta}}{(1-2\delta)^{1-2\delta}}\right)^{\mu}\right]$, all intervals contain sample result points whose number differs from the expected value by no more than $2\delta\mu$. As each such point carries weight $\frac{1}{k\mu}$ into the probability, and there are either no more than $k/2$ overflow intervals (holding more points than μ) or no more than $k/2$ underflow intervals, from Lemma 1, we get that the variation distance is no more than $2\delta\mu\cdot\frac{1}{k\mu}\cdot\frac{k}{2}=\delta$. □

To get a numerical sense about the bound, we take $k=5$, $\delta=0.2, \mu=60$. Then from Theorem 2.2, using 300 sample points (rounds), with probability at least 0.91, the variation distance between the result of the SERP algorithm and the ideal distribution is no more than 0.2. This is a (rather conservative) theoretical guarantee, and as we shall show, in practice, one can obtain a small variation distance with significantly fewer rounds. On the other hand, theoretical guarantees are important as they hold for any dataset while the result of a particular experiment depends on its data.

2.1.4 Statistical Model

As individual data items are already uncertain and imprecise (even their distributions are estimated), statistical information about the result of an operation is frequently more desirable than its full distribution. It is well known that scientific databases typically require operations on huge volumes of data. For example, the full result distribution for SUM or AVG on a few million tuples is unnecessary. Reporting an *expected value* and the *variance* is often sufficient and more useful.

Moreover, it is much more efficient for the database system to merely compute the statistical information about the result, rather than the full distribution. Often, statistical information can be computed not only much faster, but also more accurately (i.e., without the approximation needed in computing the full distribution). For example, if the database system first computes the full distribution of SUM or AVG which requires approximation and then calculates the expected value and the variance from the full distribution, it would be less accurate than if the database system ran in statistical mode and returned the expected value and variance directly.

The structure of the statistical information consists of six parts: $\{E, Var, UB, p_1, LB, p_2\}$. Here, E and Var are the expected value and the variance, respectively. UB and probability p_1 express an upper bound satisfying $\Pr[X > UB] \le p_1$, where X is the result random variable. Likewise, LB and p_2 indicate a

lower bound such that $\Pr[X < LB] \le p_2$. To make it meaningful, typically p_1 and p_2 are small; hence, with high probability X is between LB and UB. Note that not all six parts are required, as some of them may be difficult to obtain in some cases. To be more concrete, we look at a few examples.

- **SUM and AVG.** For clarity we only discuss AVG; SUM is similar. Let the result random variable be $X = \frac{1}{n}\sum_{i=1}^{n} X_i$, where X_i is the random variable for each value and there are n values to be aggregated. From linearity of expectation, we have $E[X] = \frac{1}{n}\sum_{i=1}^{n} E[X_i]$. For variance, we have $Var[X] = \frac{1}{n^2}\sum_{i=1}^{n} Var[X_i]$. Thus, solely from the expectation and variance of the base values (X_i's), we can calculate the E and Var components of the result. The expectation and variance of the base values can be easily calculated from the discrete PDF, be it in the form of (v_i, p_i) pairs or equidepth intervals. We omit the details due to space constraints. The simple operations on the base value expectations and variances make the statistical information computation very efficient.

- **Arithmetic operators.** Specifically, we look at addition, subtraction, multiplication and division. As SUM uses addition, computing E and Var for the result of addition and subtraction are similar. For multiplication, due to the independence assumption, we have

$$E[X \cdot Y] = E[X] \cdot E[Y]$$

$$Var[X \cdot Y] = Var[X] \cdot Var[Y] + E[Y]^2 Var[X] + E[X]^2 Var[Y]$$

Thus, we can get the expectation and variance of the product. Division is the reverse of multiplication, and the formulas can be derived accordingly.

- **COUNT.** We consider a simple SQL statement "*SELECT COUNT(*) FROM A WHERE X > 5.6*" as an example, where X is an uncertain attribute. The result count is a random variable and we wish to compute its statistics. We can reduce this case to SUM on boolean random variables B_i's (one per tuple) that bears the value 1 if the predicate is true and 0 otherwise. That is, $C = \sum_{i=1}^{n} B_i$. Knowing X's distribution, we can get the probability that the predicate is true, and hence the complete distribution of each B_i (and certainly its expectation and variance). We can obtain the result C's expected value and variance in a manner similar to SUM.

However, there is an additional interesting technique we can apply here. As C is the sum of independent 0/1 random variables, we can use *Chernoff bounds* to obtain good upper and lower bounds. For the case of the upper bound, we have $\Pr[C > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$ where $\mu = E[C]$. In particular, if $\mu = 10{,}000$ and we let $UB = (1+\delta)\mu = 10{,}500$, then $p_1 = 4.57 \times 10^{-6}$, a really tight bound. We can use a similar formula for the lower bound.

- **SUM and AVG of correlated values.** In certain applications, one may *not* be able to assume *independence* between the values being summed or averaged. If this is the case, we can use Azuma-Hoeffding's inequality [MU06] to establish upper and lower bounds for the result of SUM and AVG. This involves the concept of "martingales" which allow the underlying random variables to be *dependent*. We model successive partial sums (i.e., $S_t = \sum_{i=1}^{t} X_i$) as a "Doob martingale". We omit the details here due to space constraints. Finally, we get the upper bound $\Pr[S - E[S] \geq \lambda] \leq \exp(-\frac{2\lambda^2}{\sum_{i=1}^{n}(b_i - a_i)^2})$

where random variable S is the sum of n uncertain values X_i $(i=1,\ldots,n)$ for which each X_i is within $[a_i, b_i]$. This gives an upper bound (UB and p_1) for SUM and likewise for AVG. A similar formula exists for the lower bound (LB and p_2).

- **Computing bounds from *E* and *Var*.** Once we obtain the result's E and Var, without other knowledge about the distribution, we can obtain an upper and a lower bound by applying Chebyshev's inequality [MU06]. The one-sided version of Chebyshev's inequality is $\Pr[X \geq E[X] + a] \leq \frac{Var[X]}{Var[X] + a^2}$. This gives an upper bound (UB and p_1) for a result X. A similar version of inequality exists for the lower bound.

There may be multiple ways to compute the bounds. For example, from E and Var one can compute bounds using Chebyshev's inequality; as introduced earlier, for certain operations we may use Azuma-Hoeffding's inequality to obtain bounds. The database system can explore multiple ways and return the best or most applicable bounds to the user. For any application that requires an answer within some deadline, like real-time processing, we can use a cost model that estimates the execution time to compute the full distribution. If the estimated time to return a full distribution with a small variation distance is too long, then the optimizer can quickly compute and return the result in several different ways. It can use a smaller value for k (i.e., number of intervals), it can use fewer *rounds* (i.e., REPEAT

loops in SERP), or it can use statistical mode. Our cost model would be used to make this decision relative to the known deadline, and, in this way, trades off precision for shorter latency.

2.1.5 Empirical Study

2.1.5.1 Setup and Datasets

In this section we perform a comprehensive empirical study on two real world data sets. We extend the ASAP array database system with a data model that captures uncertainty and with the algorithms that we have introduced in this paper. The experiments were conducted on a 1.6GHz AMD Turion 64 machine with 1GB physical memory and a TOSHIBA MK8040GSX disk. We performed the experiments on two sets of real world data:

1. The positions of ships measured with GPS during one week between March 1st and March 7th of 2006 in the East and West coasts of U.S., obtained by the *United States Coast Guard* [PA07]. The position data is recorded once per several seconds, with latitude and longitude.

2. The global temperature records from the year 1850 to 2006 obtained by the *Climatic Research Unit* of Univ. of East Anglia in U.K. [T07]. The data records air temperature anomalies on a 5° by 5° (latitude and longitude) grid-box basis. The anomalies (in °C) are with regard to the mean value (of *that same* location) during the normal period between 1961 and 1990.

Both data sets have inherent uncertainty due to many factors [BK06]. Different parts of the data in the multidimensional arrays can have different levels of uncertainty. For example, temperature readings in the winter have larger errors than in the summer, and earlier years have larger errors. We omit the detailed description due to space constraints.

2.1.5.2 Ship Positions Dataset: Result Accuracy

We run SERP and the heuristic algorithms to compute the total angle a particular ship has turned in a period of 7 days. In this section, for SERP, the number of intervals (k) of the result distribution is 5, unless specified otherwise. The variation distance (with the ideal one) of SERP with different rounds and that of the heuristic algorithms is shown in Figure 2.1(a) (in which "20-r" is shorthand for 20-round SERP and so on). We can see that in this case a 20-round SERP already gives us very good accuracy with variation distance from the ideal less than 0.1. A 100-round one would further improve it while we can see that the rate of improvement drops as we do not see much improvement for 200 rounds. This verifies our theoretical proof of Theorem 2.2, and is in fact showing that even as few as

20 rounds gives a good accuracy in practice as the theoretical proof is a safe guarantee. In addition, the equidepth discretized ideal distribution and the result of the 20, 100, and 200 round SERP are shown in Figure 2.1(b). This shows pictorially how close the distributions are.

Figure 2.1(a) also shows that heuristic algorithms have big variation distances, with RAND being the worst. For distributions that are relatively far from the ideal distribution, it would also be helpful to compare the "coarser-grained" statistics such as simply the expected value and variance. We show these in Figure 2.1(c, d). We can see that the heuristic algorithms can satisfy the most basic property of being close in expectation, but are far in variance (note that the ideal distribution has the biggest variance), indicating the detailed distribution is far off. In retrospect this is reasonable as with heuristic algorithms the errors can accumulate arbitrarily with each intermediate step; hence they do not scale well. This is not the case with SERP as we compute the final result distribution directly.

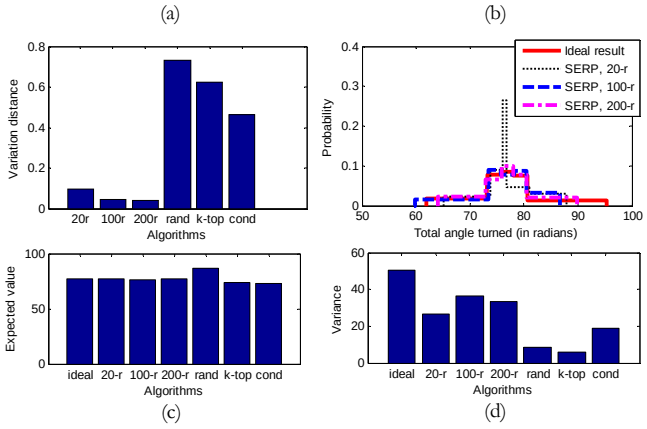


Fig. 2.1: (a) Variation distances of results of the algorithms from the ideal one; PDFs (b), expectations (c), and variances (d) of the ideal and the results of various algorithms.

2.1.5.2 Ship Positions Dataset: Speed

We now look at the CPU cost of our algorithms. We also compare it with the I/O cost of just reading the position data for a particular ship from disk. Note that for SERP, as discussed earlier, we have the optimization that we only need to do I/O in one pass, carrying out sampling and computation for multiple rounds in parallel. The result is shown in Figure 2.2(a). We also measure the CPU cost of simply doing the operation on the data without any uncertainty (i.e., just the mean values), shown as the last bar (labeled "none") in the figure.

The result indicates that the CPU cost of SERP is roughly proportional to the number-of-rounds parameter. In the case of 20 rounds, which gives a variation distance less than 0.1 as shown earlier, its CPU time is well below the I/O cost. Some heuristic algorithms run faster, but they are inaccurate as we have shown. All these algorithms have a much greater CPU cost than computing on the non-probabilistic data directly. There seems to be an inherent cost of computing a probabilistic distribution of the result.

We next vary the number-of-result-intervals parameter. We look into the cases of k = 3, 5, 7, 9. For each k value, we compute the discrete ideal distribution with k intervals. Then we record the (minimum) running time of SERP such that its variation distance is no more than 0.1 from the ideal for each k value. The result is shown in Figure 2.2(b). As expected, as k increases, the CPU cost increases as well leading to more information about the result distribution, thereby trading off performance for accuracy.

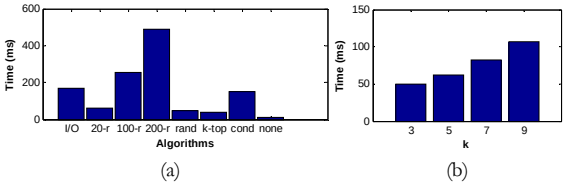


Fig. 2.2: (a) I/O time and CPU times of various algorithms. (b) CPU time vs. different k values.

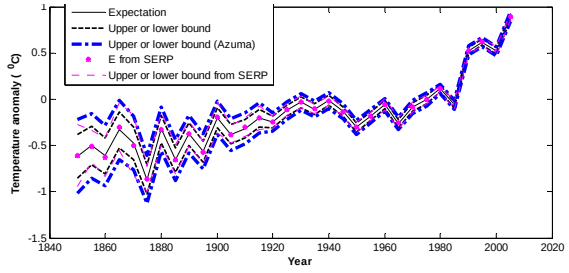


Fig. 2.3: Expectation and bounds of average temperature anomalies, obtained with different methods.

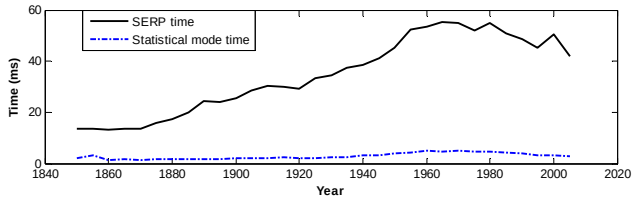


Fig. 2.4: Query running time comparison of SERP vs. statistical mode only.

2.1.5.2 World Temperature Dataset: Statistical Mode

We turn to the global temperature data set and compute the average temperature anomalies (in °C) for the days of the *year*, across *the globe*, for every fifth year from 1850 to 2005. We use statistical mode. For each year, we compute $\{E, Var, UB, p_1, LB, p_2\}$ of the query without obtaining the full

distribution. We choose $p_1=p_2=0.1$ and compute bounds in two ways: using *Chebyshev's* inequality and using *Azuma-Hoeffding's*. For comparison, we also run a 20-round SERP to compute the full distribution and then get *E*, *Var*, and *UB*/*LB* (using Chebyshev's inequality) out of that. The result is in Figure 2.3. Observe that the bounds from Azuma-Hoeffding's inequality are more conservative (i.e., looser) than those computed using Chebyshev's inequality. This is because Azuma-Hoeffding's inequality is more general in that it does not assume independence of the data, which is useful when in fact the values are correlated. We next notice that the expected value and bounds computed using a full distribution out of a 20-round SERP are really close to the results of the statistical mode. This mutually verifies the accuracy of both the statistical mode and the 20-round SERP. As expected, older data has larger variance and thus wider bounds. We also measured the CPU time, shown in Figure 2.4. Clearly the statistical mode is a lot faster than SERP. And this suffices if statistical information alone is all a user cares about.

2.1.5.3 World Temperature Dataset: Predicate Evaluation Strategies

Finally, we experiment on different predicate evaluation strategies. We issue the query:

SELECT year, AVG(anomaly)
FROM history
WHERE year BETWEEN 1850 AND 2006 AND year MOD 5 = 0
GROUP BY year
HAVING AVG(anomaly) $>_{0.8}$ *0.3 OR AVG(anomaly)* $<_{0.8}$ *-0.3*

Here, "$>_{0.8}$" is a predicate in a generalized form, meaning "with probability at least 0.8, the left side is greater than the right side". Only tuples for which ">" is true with a probability higher than the threshold (0.8) are returned. This semantics has been used in other work (e.g., [CS06]). Thus, the above query selects the years whose average anomaly (averaged on both days in the year and 5° latitude by 5° longitude grids of the globe) is either above 0.3 (with probability at least 0.8), or below -0.3 (with probability at least 0.8). There are three ways to evaluate this query:

1. Compute the full distribution of *AVG(anomaly)* using SERP, and then compute the probability that it is > 0.3. If this probability is at least 0.8, then it satisfies the predicate. The same is true with the <-0.3 part.
2. Run in statistical mode, computing *E* and *Var* of each year. Then apply Chebyshev's inequality to evaluate the predicate.
3. Same as (2), but use Azuma-Hoeffding's inequality.

Method	> 0.3 (0.8 prob.)	< -0.3 (0.8 prob.)
Full Distribution	1990 1995 2000 2005	1850 1855 1860 1870 1875 1885 1890 1895 1905
Chebyshev	1990 1995 2000 2005	1850 1855 1860 1870 1875 1885 1895 1905
Azuma-Hoeffding	1990 1995 2000 2005	1860 1875 1885 1895 1905

Fig. 2.5: Query results with different methods.

Figure 2.5 shows the result. For the "$>_{0.8}$ 0.3" predicate all three methods return the same set of years. For the other predicate, the 2nd method has one fewer (1890) in the output than the 1st, while the 3rd method has three fewer than the 2nd. These years are at the border line of the predicate. This illustrates the fact that the inequalities are theoretical guarantees and thus in general result in conservative decisions. The Azuma-Hoeffding inequality does not assume data independence and can be used in more general cases, resulting in more conservative decisions than Chebyshev's. Moreover, the fact that the discrepancy appears for the "$<_{0.8}$ -0.3" predicate but not the other one is because the data in older years has larger variance. In sum, the result of using Chebyshev's inequality is close to the full distribution, and should be used when possible, since it is much more efficient. Azuma-Hoeffding's inequality should be used when one cannot assume data independence.

2.2 A Special Join Algorithm in the SERP Framework and Experiments

2.2.1 Sampling-Based Join (S-Join)

2.2.1.1 The S-Join Algorithm

One of the drawbacks to query processing through Monte Carlo sampling is that it is a very time-intensive process. If 1000 rounds of sampling are used, then every query must be evaluated 1000 times. Traditionally, the JOIN operation has been one of the larger bottlenecks in query evaluation. Performing 1000 JOINs every time a JOIN query is posed would be a painful process. To alleviate this pain, we have developed a JOIN algorithm that is useful when at least one of the JOIN attributes is uncertain. Our algorithm takes advantage of the structure of the sampling problem to provide a significant speedup over running a standard JOIN algorithm over and over. The algorithm is a modified sort-merge-join and we call it the S-Join algorithm.

Returning to the temperature example, what if we are interested in finding two days that have the same temperature? To realize this query in SQL, a self-join must be performed where the join attribute is the uncertain temperature attribute:

SELECT temp1.day, temp2.day

FROM temperature as temp1, temperature as temp2

WHERE ABS(temp1.value – temp2.value) < ε AND

temp1.day != temp2.day

In this query, value of the temperature table is an uncertain attribute. Assuming that the uncertain attribute is sampled from 1000 times, naively evaluating this relatively simple query will result in performing 1000 join operations on a table that could be very large.

The intuition behind the S-Join algorithm is that given a series of uncertain values, the order of samples drawn from their distributions should be similar to the order of their expected values since the correction parts are typically small compared to the gap between the original parts of any two values. Thus, once we sort the tuples according to their samples in the first round of a Monte Carlo processing, we can draw samples in that same order for all other rounds and the samples themselves should be almost sorted (called pseudo-sorted).

Sorting a pseudo-sorted list is much cheaper than a complete sort. If most of the values to be sorted are already in sorted order, insertion sort is has linear run-time. While an insertion sort takes time N2 to sort a random series of values, if it can be guaranteed that no value is more than c spots away from its correct location, for a constant $c << N$, the time decreases to $c \cdot N$.

The S-Join algorithm is shown in shown in the text box below. It proceeds as follows. Perform an external sort on the input tuples according to their first round's sample values, putting them in that order. Note that in the rest of the algorithm, we perform multiple sampling rounds in parallel in order to share the ordering information from round 1 (the external sort) with as many subsequent rounds as possible. This minimizes the need to re-read the result of the external sort. We assume that further sampling will not change the position of a tuple in this order by more than k pages. Without loss of generality, let us take k = 2. Now allocate k+1=3 pages in memory for each side of the join for some number of rounds R. R is selected so that 2*(k+1)*R pages are able to fit in memory at once. For the remaining R-1 rounds, read in the first three (i.e., k+1) pages of the externally sorted tuples. As each tuple is read in, draw samples from the distribution on its join attribute. When a sampled value is drawn for round r, insert that value into r's pages using an insertion sort, maintaining sorted order in each round.

After this sampling process is complete, we can perform a merge-join on the first page of every round. Because of our assumption that tuples can only deviate by k=2 pages from expected order, the first of these pages will be in final sorted order while the next two will be pseudo-sorted. This is

because only values sampled from tuples in the first three pages can end up in the first page. Tuples whose order (determined in the 1st round) puts them in the fourth page cannot be in the first page when ordered by sample values, so the first page must be in final sorted order after reading and sampling the first three pages. The second and third pages are pseudo-sorted but are not in final sorted order because tuples from the fourth and fifth page, which have not yet been sampled, may need to be interspersed within the second and third page. Now the merge process is performed over the first page in every round. This merge process frees up a page so that a new page can be read in and sampled. Another insertion sort is performed with the samples from this new page, and as a result, another page is now in final sorted order and can be merged in each round. This process continues until sampling and merging is complete.

Step (10) of the algorithm tries to get a new page into the buffer either in sorted order (for the 1st round) or pseudo-sorted order (for other rounds). The same order information comes from the sorted temporary file in step (1). Thus, we can save the I/O cost of reading from the file and once a page is loaded into the buffer by one round, it is shared by all other rounds. Ideally, we want to read the order information from the sorted file only once, for all concurrent rounds. However, as we analyze, in the worst case (although rare) this may not be achievable.

(1) Do sampling and external sort for the first round (of S- JOIN). The result is a temporary file on disk (for each side of the JOIN).
(2) For *k=2*, Allocate 3 pages in buffer for each side of the JOIN for each round (for the number of rounds that can fit in the buffer space).
(3) For round 1, load the sorted values for the 3 pages allocated in the buffer for each side of the JOIN from the sorted file in (1).
(4) For each of the other rounds,
(5) Obtain fresh samples for the 3 pages in buffer (for each side of JOIN) and arrange them in *pseudo-sorted* order determined by (1).
(6) Perform an *insertion sort* on the pseudo-sorted 3 pages (for each side of JOIN). After doing this, according to our assumption, the first of the 3 pages will be *exactly* sorted, and the other two pages will remain pseudo-sorted.
(7) End
(8) For each round, repeat the following until JOIN finishes (all rounds *in parallel*, e.g., in a *lockstep* or in *round-robin* fashion),
(9) Do the "merge" step of the JOIN on *exactly* sorted pages.
(10) When a page is *consumed* (i.e., finishes merging), read the tuple order of a new page either from the temporary file in (1) (if this has not been loaded into a buffer page of another round) or from a buffer page of another round (if it is already loaded there). The new page replaces the consumed one.
(11) If this is not round 1, obtain a set of fresh samples for this new page and do *insertion sort* to adjust the order. This converts another page (the oldest among the three) from *pseudo-sorted* to *sorted*.
(12) End

Throughout the algorithm we let each round only take 3 (i.e., k+1) pages in buffer and allow as many parallel rounds as possible. If, in the rare case, a new sample of a tuple brings it out of order by more than k pages (i.e., our assumption on the pseudo-sorted order does not hold), we can easily find and rectify it by merging the outlier tuple separately. Note that a database server can also pre-sort on the expected values and save the external sort step at runtime.

The S-Join algorithm provides a significant speedup over naively performing all joins independently because only one external sort must be performed. The only subsequent sorts are in-memory insertion sorts on tuples that are already nearly sorted. Additionally, by performing multiple rounds together, unnecessary disk accesses are eliminated. If 200 rounds can be performed at once out of a total of 1000 rounds, the externally sorted tuples need to be read from disk only about 5 times as opposed to 1000 times. S-Join can be utilized whenever one could use a sort-merge-join with an uncertain JOIN attribute. Note that if there is lots of overlap between the distributions in a JOIN column (which is arguably rare since the correction part is typically small), the algorithm would still work but it would not yield as great an improvement since the insertion sorts will take longer.

2.2.2 Result Entropy and Cession of Sampling

One remaining unanswered question is how many rounds of Monte-Carlo sampling should be carried out. By observing the entropy of the uncertain distributions in result tuples, we can get a better idea of when results are stable. Recall that there are two forms of uncertainty that a system seeks to handle. The first is set-membership uncertainty for output tuples in the result. The second is the marginal distribution for a field in a result tuple given that the tuple is in the result.

Theorem 2.3: The entropy of the query result can be computed as $H(T) + H(V|T)$, where $H(T)$ is the entropy of the distribution of the result tuples' membership and $H(V|T)$ is the conditional entropy of the distributions of the fields in the result tuples, given that the result tuples are in the result.

Proof: This follows from the chain rule of entropy [CT91]. □

We next observe some characteristics of the entropy evolution during query execution.

Definition 2.2: A query cut is a set of intermediate tuples that appear at the same time anywhere in the query flow graph (in which nodes are operators such as JOIN and UNION and edges indicate

data flow). The set of all input tuples is a query cut; all returned results is a cut; so is a set of intermediate tuples from which the final results can be completely computed.

Theorem 2.4: As query processing progresses, a series of query cuts are formed from input to output. The entropy of a query cut in the series can only decrease (or stay the same).

Proof: Let X be a random variable representing the query cut at input; let Y and Z be another two query cuts such that Y is closer to X. Then, from data processing inequality [CT91], we have that I(X, Y) ≥ I(X, Z) since random variables X, Y, Z form a Markov chain (assuming database operators are deterministic). Using the property of mutual information that I(X; Y) = H(Y) – H(Y|X), we have H(Y) – H(Y|X) ≥ H(Z) – H(Z|X). Because H(Y|X) = H(Z|X) = 0 (deterministic database operators), we have H(Y) ≥ H(Z), finishing the proof. □

In particular, the result entropy must be no more than the entropy of the query inputs.

We may consider a "reverse query engine" and think of a Monte Carlo algorithm as simply sampling from the result distribution. We use a big number of such samples to estimate the actual distribution. We next establish that regardless of the detailed result distribution, the quality of the result, or the number of sampling rounds needed, is closely related to the entropy of the actual result.

Theorem 2.5: Consider two queries Q_1 and Q_2 with the actual result distributions R_1 and R_2, respectively, where $H(R_1) < H(R_2)$. Suppose an n-round Monte Carlo algorithm obtains an output distribution S_1 for Q_1 and another n-round Monte Carlo algorithm obtains an output distribution S_2 for Q_2. Then we have that the variation distance $VD(S_1, R_1)$ is no more than $VD(S_2, R_2)$.

Proof: Before a formal proof, we first give some intuition. We know that the entropy of a random variable is closely related to its Kolmogorov complexity [CT91]. As sampling progresses in rounds, one monotonically obtains more evidence or information about the result distribution. Thus, a more complex distribution requires more rounds. Equivalently, given the same number of rounds (n), we obtain better result quality for the result distribution that has smaller entropy. We next show the formal proof.

From Lemma 1, we have $VD = \frac{1}{2}\sum_i | p_i' - p_i | = \sum_i o_i$

where p_i and p_i' are the probability of a "bin" in the actual result distribution and the distribution output by an n-round Monte Carlo query evaluation algorithm, respectively; o_i is the overflow difference of p_i' at bin i (it has value 0 if no overflow at that bin). Rewrite the expression and introduce random variables O and P:

$$VD = \sum_i o_i = \sum_i p_i \frac{o_i}{p_i} = E\left[\frac{O}{P}\right] \approx \frac{E(O)}{E(P)} \qquad (1)$$

where the last equality assumes the independence of overflow amount and the bin's probability. Now imagine we do optimal coding (i.e., Huffman codes) and assign binary codes to symbols according to their probabilities such that the expected code length is minimal. One can think of this procedure as in a binary tree, shown in Figure 2.6.

Consider a complete binary tree of depth d. At the leaf level, there are 2^d "small buckets". We uniformly at random throw n balls into the buckets (n corresponds to the n-round Monte Carlo algorithm). In expectation, each bucket receives $n/2^d$ balls. Any number different from the expectation is either an overflow or an underflow. We map a "bin" in the actual result distribution to a node in the tree, whose subtree covers a set of buckets. Thus, when a "bin" overflows, it implies that the set of buckets that it covers receive more balls than the expectation of their total number of balls. The "location" (in particular, depth) of a bin in the tree corresponds to its optimal binary code length (thus, higher probability bin has smaller depth). A bin at depth H has probability $P = 2^{-H}$ and covers 2^{d-H} buckets.

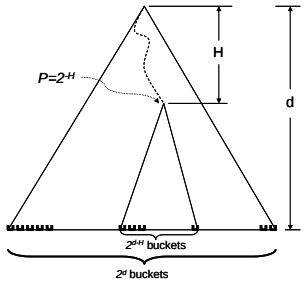


Fig. 2.6: Illustrating the mapping.

Consider two instances of H, h_1 and h_2 with $h_1 < h_2$. They correspond to two bins covering a set s_1 of 2^{d-h_1} buckets and a set s_2 of 2^{d-h_2} buckets. The size of s_1 is $2^{h_2/h_1}$ times of the size of s_2. Treat s_2 as a "unit". Since a "unit" either overflows or underflows or stays even, the expected overflow of s_1, $E(O_1)$ must satisfy

$$E(O_1) \le 2^{h_2/h_1} E(O_2) \qquad (2)$$

where $E(O_2)$ is a unit's average overflow (unless all units in O_1 overflow, an underflow will offset an overflow). Rewriting (2) we have, $\frac{E(O_1)}{2^{-h_1}} \le \frac{E(O_2)}{2^{-h_2}}$. As entropy corresponds to optimal code length, we have $\frac{E(O_1)}{E(P_1)} \le \frac{E(O_2)}{E(P_2)}$. Combining this with equation (1), we have proved the theorem. □

Intuitively, Theorem 2.5 tells us that smaller result entropy requires fewer rounds to converge. Based on Theorem 2.5, we now briefly discuss two optimizations we can have in the stopping of a Monte Carlo query evaluation.

As discussed earlier, a general stopping condition occurs when the result entropy converges. We can determine the convergence by the condition $|H_{i+1} - H_i| \le \delta H_i$ (for some small threshold δ), where H_i and H_{i+1} are the result entropy for round i and i +1 respectively. However, that may cause some overhead due to computing the entropy frequently. Theorem 2.5 provides a mechanism for leveraging results from past queries. If query q_1 has result entropy that converges to h_1 after 200 rounds and we find query q_2's result entropy $h_2 > h_1$, we know q_2 will take at least 200 rounds and can stop testing convergence until then. Thus, one can keep a table of (result entropy, number of rounds to converge) for past queries. During the current query processing, if we compute its result entropy and find it is greater than that of some entry in the history table and the current round number is less than the one in that entry, we can stop computing entropy or testing convergence until we reach that round.

Another possible optimization is to selectively stop computing the value distribution for a subset of the result tuples (that have smaller entropy) first. Between the set-membership uncertainty for tuples in the result and the value uncertainty for a field in a result tuple, the first one is a binary decision and typically takes fewer rounds. For example, after 200 rounds, any output tuple not yet seen has probability of less than 1/200 of being in the result set because each round is symmetric and independent (hence has the same probability) in producing the output tuple. Thus, computing value distributions is typically more costly. After settling the set-membership uncertainty, for value uncertainty (marginal distributions), the database server can compare the value entropy of different result tuples. For those with smaller entropy, in fewer rounds of Monte Carlo query evaluation we can obtain a good quality distribution (convergence). The database server can now "stop" computing these result tuples and only proceed with more rounds for those tuples that have greater entropy. For this methodology to work, the database server needs to keep track of the lineage of result tuples and

selectively choose source tuples to run. Clearly both optimizations save computing time and resources.

2.2.3 Empirical Study

2.2.3.1 Problem and Setup

A common vexing problem in multi-sensor tracking is to devise a mapping between the tracks of one sensor and the tracks of another sensor, assuming both sensors are tracking the same objects [BB01]. Once a mapping has been verified, tracks from different sensors on the same object can be fused together to form a single object track. This problem, known as the track-to-track correlation problem, is well-suited for our study because of the nature of errors in sensor tracks. Track errors are often the result of bias in the sensor that maintains the track. Thus, it can be expected that errors in tracks that originate from the same sensor will be correlated in some fashion. The problem requires testing for equality between uncertain values where the uncertainty may be correlated across multiple values. This problem is analogous to many other problems in multi-sensor fusion where the mapping between sensors must be learned. We generate synthetic datasets for the tracks of different sensors and model correlation of errors for tracks originating from the same sensor with MRF. Each track itself also has a random error independent of other tracks. We implemented the algorithms presented in this paper. The experiments were run on a Debian Linux workstation with an AMD Athlon-64 2 Ghz processor, 512 MB memory and a Samsung HD160JJ disk.

2.2.3.2 Correctness

As an example, Figure 2.7 shows the positions in X and Y of six objects, each being tracked by two sensors. Tracks 1-6 belong to sensor 1 and are illustrated with dots in solid-lined circles. Tracks A-F belong to sensor 2 and are illustrated with dots in dashed-lined circles. The dots in the center depict the reported position of each track, the original value in our model. The circles around those dots are drawn one standard deviation away from the center and serve to demarcate the correction value in our model. The actual position of a track is the original plus the correction. The errors of the tracks from the same sensor are correlated due to the common sensor error. Individual tracks also have a random error.

Let us define T1 as the set of all tracks belonging to sensor 1 and T2 as the set of all tracks belonging to sensor 2. A mapping M is defined as a set of pairs such that each t1 and t2 appear exactly once. The probability of a mapping M being valid is the probability that distance (t1, t2) $< \varepsilon$ for all pairs (t1, t2) in M. Under an (erroneous) independence assumption,

$\Pr(M) = \prod \Pr(t_1 \approx t_2)$.

The query necessary to create this mapping is:

SELECT sensor1.trackID, sensor2.trackID

FROM sensor1, sensor2

WHERE ABS(sensor1.x – sensor2.x) < ε AND

ABS(sensor1.y – sensor2.y) < ε

In this case, the result of note is not any individual result tuple but a joint distribution on many result tuples. In the semantics defined in earlier, each result tuple will be accompanied by a k-bit array where k is the number of rounds of sampling. The value of bit i for tuple t will be 1 if that tuple is a member of the result set in the i'th round of sampling. Thus we can compute the probability of any mapping M by doing a bit-wise AND of all the bit arrays of result tuples in M, counting the number of 1s, and dividing by k.

To the naked eye, it looks obvious that track 1 should be paired with track A, track 2 with track B, and so on. However, our experiments show that if independence between track errors is assumed, the probability of this mapping being valid quickly approaches zero as the number of tracks increases.

We drew 10,000 rounds of samples for each track using the correlated model and the uncorrelated model. The uncorrelated model has the same variance as the marginal distribution of the correlated model. We tallied up the number of rounds that yielded sampled track states such that distance(t1, t2) < ε = 5 for all pairs (t1, t2) in our hypothesis M under each model. We then divided that tally by the number of rounds to yield Pr(M). Figure 2.8 shows the average Pr(M) over 10 distinct trials under both models as the number of tracks increases. Even though the error circles seem to overlap quite a bit, the direction of each error is unknown, and the chance that solid-circled tracks have negative error in X and Y while dashed-circle tracks have positive error in X and Y so that the actual track positions are the same for all pairs is quite small (approximately 0.1 for each track). Thus, if the correlation between errors of a given sensor's tracks is ignored, the probability that all six pairs of tracks are caused by the same six objects is 0.16. In contrast, if most of the track error is attributed to correlated sensor error, the effects of adding more tracks on Pr(M) is mitigated. It is clear from the figure that if correlations in error are ignored, simple queries yield highly erroneous results even when the number of correlated tuples is fairly small.

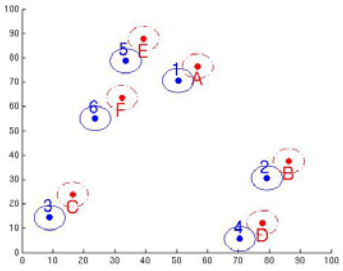


Fig. 2.7: Example tracks from two sensors.

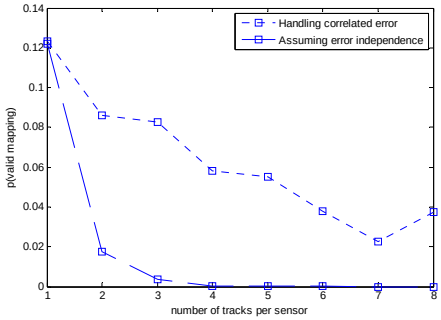


Fig. 2.8: Probability of a valid mapping of tracks under correlated and uncorrelated model.

2.2.3.3 Performance

In order to process queries of the form:

SELECT A.ID, B.ID

FROM A,B

WHERE ABS(A.value – B.value) < ε

as in the previous example, it becomes necessary to process JOINs efficiently over multiple rounds of sampled data. Imagine hundreds of sensors each tracking thousands of objects. In this case, efficiently processing the JOIN is of the utmost importance. We implemented our S-Join algorithm presented and tested it against two standard naïve JOIN algorithms. The first naïve algorithm (1) performs a sort-merge join on each round of samples independently. It reads in all the data, samples from the data, and performs an in-memory quick-sort if space allows or an external merge-sort otherwise. It then does the standard merge on the sorted samples. The second slightly less naïve algorithm (2) performs one external sort on the original values (expected values). It then reads the values, samples, resorts with insertion sort, and merges one round at a time. This algorithm should take the same amount of time to sample and sort as S-Join but must read in the sorted expected values in each of the n rounds. In contrast, if x rounds can fit in memory with each round occupying k+1 pages, S-Join only needs to read the data n/x times. They all performed 1000 rounds of sampling and joined relations of equal size.

Figure 2.9 shows the average runtime of 4 trials for S-Join and the naive merge-join algorithms presented above on relations of various sizes. The results of the experiments show that S-Join has roughly linear performance until the cost of doing one external sort outweighs the cost of doing 1000 rounds of linear traversals. In contrast, the naive sort-merge-join algorithm (1) is slower than S-Join even when the entire sort can be done in memory. The dramatic bump in run-time of (1) occurs

around relation size = 65,000 tuples when the internal sort becomes an external sort. After this point, the cost of repeated sorts becomes overbearing. Algorithm (2) performs better for smaller datasets mainly because it requires less overhead than our algorithm which must keep track of multiple rounds of sampling at once. However, once the entire relation can no longer fit in memory, it must be reread from disk during every round, and the cost of the disk-reads slows it down considerably.

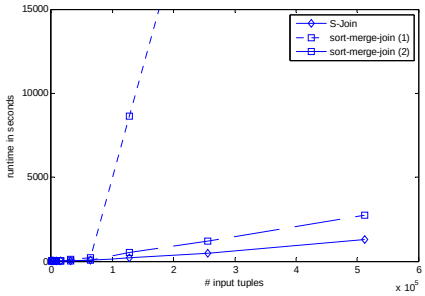


Fig. 2.9: Runtime comparison of S-Join and two other algorithms.

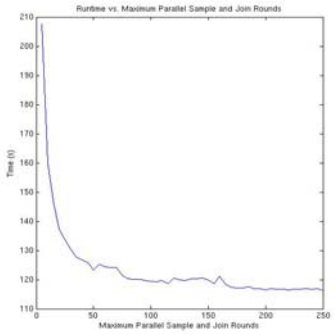


Fig. 2.10: Maximum Parallel Rounds vs. Runtime of S-Join.

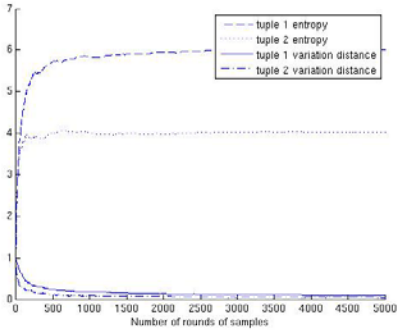


Fig. 2.11: Entropy and variation distance over multiple rounds of samples.

One of the main parameters of the algorithm is the number of sampling rounds to be run at once. If x rounds can fit in memory, our algorithm must read data n/x times, where n is the total number of rounds. Thus, it stands to reason that the more rounds handled in parallel, the faster the algorithm will complete. Figure 2.10 shows how the number of sampling rounds handled concurrently affects the runtime of the algorithm. Ten tests were run for every five maximum parallel round numbers (5, 10, 15…) and the average runtime over the ten trials is plotted. The runtime initially drops precipitously because raising the number of parallel rounds from 10 to 20 lowers the number of database reads from 100 to 50 when sampling 1000 times. However, the difference between 200 and 250 rounds in parallel is only one database read.

2.2.3.4 Stopping Conditions

Theorem 2.5 states that if the entropy of the actual distribution of result tuple t1 is greater than the entropy of the actual distribution of result tuple t2, tuple t1's variation distance will be greater than that of tuple t2 with the same number of rounds. Figure 2.11 illustrates this point using two randomly-selected result tuples of differing entropy from the previous experiment. The true distribution, necessary for calculating variation distance, was computed by sampling 1,000,000 times. For a fixed number of rounds, t1's variation distance is greater than t2's. Notice that the entropy levels off as the number of rounds increases and the smaller the entropy, the faster it levels off (t1 converges at around 1000 rounds while t2 does at around 500 rounds). By observing when the entropy of a particular tuple starts to level off, the system can decide when no more sampling is necessary for that tuple.

Chapter 3

HANDLING CORRELATED UNCERTAIN ATTRIBUTES

In this chapter, we discuss in details about using Markov Random Fields (MRF) combined with *chunking* of multidimensional arrays to model and process correlated array data (Section 3.1). In Section 3.2, we present a novel data structure, called A-trees, that essentially turns a multidimensional array into a tree structure, again to handle correlated data.

3.1 Modeling and Processing Correlated Uncertain Attributes with MRF

3.1.1 Query Semantics

It is not immediately obvious what the semantics of queries in the presence of correlated uncertainty should be. Since an attribute value is in general a distribution, a predicate, such as "temperature > 50", is true with some probability. Hence, whether a tuple appears in the result now becomes an event that happens with some probability. It is tempting to generalize the syntax of a predicate and include a probability threshold, e.g., "temperature $>_{0.8}$ 50", whose semantics dictates that the predicate is true if the temperature is greater than 50 with probability at least 0.8. Under this scheme, deterministic attributes become a special case whose probability is always 0 or 1. Such syntax and semantics have been proposed and used before (e.g., [DG04]). This generalized predicate syntax makes the representation of the result cleaner in that either a result tuple is in the result or it is not: there is no tuple (membership) uncertainty in the result set. However, there is one large problem with this semantics; it is not composable. When two predicates are correlated, the composed predicate (and result) may not be what a user intended. For example, the predicate "temperature $>_{0.8}$ 50 AND temperature $<_{0.8}$ 100" will return "all tuples that have an 80% chance of being greater than 50 and an 80% chance of being less than 100", not "all tuples that have an 80% chance of being between 50 and 100", as is probably the desired meaning. The result of a predicate is not composable from two conjunctive parts if the parts are correlated, as will be the case if they concern the same attribute.

One promising approach has been the "possible world" semantics, where all possible values of a tuple are enumerated and assigned a probability. This approach has been used in most of the previous work

on probabilistic databases (e.g., [RD07]). While the possible world semantics works very well in the discrete tuple uncertainty model, it is not clearly defined for the continuous distribution attribute uncertainty model. We propose to extend and generalize the possible world semantics.

Not only must the semantics incorporate probabilistic membership in the result set, they must also incorporate uncertainty in result values. As an example, consider the query from the introduction of all rows with temperatures greater than 80. Tuple T_1 has a temperature of 79 with a variance of 1, and should thus be in the result set with probability 0.16. However, it does not make sense for a row in the result set to have a mean temperature value of 79. Since the results should only include values greater than 80, all rows in the result set must have a value range above 80. To rectify this problem, we must treat the distribution on the temperature attribute of the tuple in the result set as a conditional probability distribution. The value in the temperature field should be the distribution of the temperature given that it is above 80 degrees, even though its unconditioned mean was 79. In this case, the result set should contain the result row RT_1, derived from T_1, with probability 0.16. The temperature field in RT_1 is the distribution of the temperature of T_1 conditioned on being above 80 (The new distribution has a mean of 80.5).

Thus, two uncertain elements must be well-defined in the query semantics: (1) The probability of membership in the result set and (2) the value distributions of uncertain fields given membership in the result set.

Integral Based Semantics

The intuition behind these semantics is that the probability density function of an uncertain attribute X covers some area A. Given a query q, there is a subregion A^+ of A in which q(X) produces a tuple t in the result set and a complimentary region in which t is not produced. We would like to know the total mass of the probability density function that falls within A^+. To find this mass, we integrate over the region. In our temperature example query, the probably density function of T_1 is a normal distribution with mean 79 and variance 1 where the A^+ region is everything greater than 80. Thus $Pr(RT_1)$ is the integral from 80 to positive infinity of that distribution.

When multiple input tuples are correlated, we must integrate over their joint probability density function. The query q can be thought of as a function that takes in a series of values and returns the

result tuple t when those values satisfy q's predicate. The integral that calculates the probability of the result tuple t, Pr(t), is presented below.

$$\Pr(t) = \int_{\substack{x_1,x_2,\ldots,x_n \\ q(x_1,\ldots,x_n)\to t}} f(X_1, X_2, \ldots, X_n)\,dA$$

In this integral, $f(X_1,X_2,\ldots,X_n)$ is the joint probability density function over correlated attributes X_1 through X_n. When the input tuples are correlated, it will often be the case that result tuples are correlated as well. In this case, the query q maps input values to a set of result tuples t_1 through t_k. The probability of that set of tuples being in the result set is given by:

$$\Pr(t_1, t_2, \ldots, t_k) = \int_{\substack{x_1,x_2,\ldots,x_n \\ q(x_1,\ldots,x_n)\to t_1,\ldots,t_k}} f(X_1, X_2, \ldots, X_n)\,dA$$

Now consider the case of uncertain values in result tuples. If a result tuple t has an uncertain attribute y, then the probability density function over y, f(y), can be calculated using a similar procedure. In this case, the function q returns a value for y given a set of input tuple values. The function f(y) can be calculated as follows:

$$f(y) = \int_{\substack{x_1,x_2,\ldots,x_n \\ q(x_1,\ldots,x_n)\to y}} f(X_1, X_2, \ldots, X_n)\,dA$$

This gives us the marginal distribution of one field (Y) in the result. If a query demands the joint distribution of k (possibly correlated) fields ($Y_1, Y_2,\ldots, Y_k$), the joint distribution is:

$$f(y_1, y_2, \ldots, y_k) = \int_{\substack{x_1,x_2,\ldots,x_n \\ q(x_1,\ldots,x_n)\to y_1,\ldots,y_k}} f(X_1, X_2, \ldots, X_n)\,dA$$

3.1.2 Representation and Query Processing

3.1.2.1 Modeling correlation and processing queries

We propose separating a probabilistic attribute value in a multi-dimensional array into two parts: the original and the correction. The original is the recorded deterministic value, and the correction is the probability distribution of a random variable C, such that original + C is the true value. In the temperature example, the original of tuple T1 is 79 and the correction is a normal distribution with variance of 1. As we observed earlier, the correction part of different cell tuples of an array can be correlated. The database system can store the original and the correction parts separately. For applications that do not care about uncertainty, the database system can simply retrieve the original part of values, and process queries without uncertainty. This uncertainty representation is beneficial for the performance of such applications. We observe that for scientific and intelligence applications of array systems, this seems to be an efficient way of storing the attribute distribution. There is a

natural correlation in the correction part of different cells, which is easier to model when separated out.

Probabilistic graphical models [J98] are a general way to handle correlation. The most often used graphical models are Markov Random Fields (MRF) and Bayesian Networks (BN). We will focus exclusively on representation and query processing using MRFs. It is straightforward to extend the ideas to the BN model. In MRF theory, random variables are represented as nodes in an undirected graph. Edges indicate dependencies between random variables. In our context, nodes in an MRF are the uncertain (correction) part of the cell values of an array, and edges reflect the correlation between the nodes. Nodes in an MRF have the "Markov" property:

$$\Pr(x_i \mid {}_i x) = \Pr(x_i \mid x_{NB(i)})$$

where x_i is a node in the graph, ${}_i x$ is the set of all nodes except x_i, and $x_{NB(i)}$ is the set of all neighbor nodes of x_i. This property illustrates that all nodes are conditionally independent of the rest of the graph given their neighbors. The Hammersley-Clifford theorem [J98] states that this property is equivalent to the Gibbs property, which is

$$\Pr(X) = \prod_{c \in C} f_c(x_c)$$

where C is the set of cliques (i.e., complete subgraphs) of the graph and x_c is the set of nodes in clique c. That is to say, the joint distribution of all nodes in the graph can be expressed as a product of multiple factors, each of which corresponds to a clique and is a function of only random variables (nodes) within that clique.

Since an array is typically huge, modeling it as a single unit may not be efficient. Meanwhile, for I/O efficiency, an array is typically partitioned into chunks to store on the disk [SS94]. This scheme is easily incorporated in the MRF model. We propose to modify the existing array chunking techniques and produce "overlapping chunks" which correspond to cliques in the MRF. Edges (dependencies) that previously crossed the border of two neighboring chunks are now included in both chunks and their MRF models. Thus, when we need to sample cells in the two chunks, the dependencies are preserved. Here is the algorithm which modifies chunking to achieve piecewise modeling and sampling.

```
(1)  Use existing techniques to make the chunking choices (e.g., [SS94]). Thus, a chunk is associated with a subgraph
     of the original dependency graph.
(2)  For each chunk c,
(3)    For each node n in the chunk (the original chunk),
(4)      For each neighbor n' of the node n,
(5)    If n' is not in c, then include n' and any edge between n' and a node in c into c.
(6)      End
(7)    End
(8)  End
```

Obtaining a graphical model is an interesting problem by itself. Many probabilistic databases employ machine learning techniques to learn the model [J98]. In this paper, we ignore this problem and assume the model is created by some third-party application (or by hand) with domain-specific knowledge. Once a model is obtained, it can be inserted into the database by populating the correction attribute of uncertain tuples with references to user-defined functions that describe the model. Assuming a model has been created for the array, query processing is reduced to inference problems in a graphical model. To process an arbitrary query, the general exact inference problem in an MRF is NP-hard. The most often used methods to solve the problem are Markov Chain Monte Carlo (MCMC) methods, such as Gibbs sampling and the Metropolis-Hastings algorithm [J98]. We take Gibbs sampling as an example and illustrate how we perform it with the piecewise MRF models.

Recall that the initial chunking of an array produces non-overlapping chunks. Algorithm CreateChunkModels creates overlapping chunks on top of those. Thus, there is a function C(n) that maps a node n to a chunk ID according to the initial chunking. We assume the chunk ID does not change from the initial chunks to the extended chunks in the CreateChunkModels algorithm. Note that a node n may reside in more than one over-lapping chunk, yet C(n) is unique, which we call the main chunk of n. The text box below shows the Gibbs sampling algorithm using the piecewise chunk models. We can observe the following property with the algorithm CreateChunkModels.

Theorem 3.1. Consider a chunk c after the CreateChunkModels algorithm is run. For every node n such that C(n) = c, all the cliques that include n are in c. No other clique is in c.

Proof. For any clique that is not included in c but covers a node n in c, all its nodes (and edges) not in c will be added to c in line 5 of the CreateChunkModels algorithm because they are all neighbors of n. This follows from the fact that a clique is a complete subgraph. □

(1) A global "visitation order" is assigned to all nodes in an array. This is done only once. It can be an arbitrary order, as long as it is fixed.
(2) Let *N* be the set of nodes that a query *q* needs to access, consistent with the global visitation order.
(3) For each node *n* in *N*,
(4) Use the MRF model of chunk *C(n)* to get a new sample. Specifically, sample from the distribution of *n* conditioned on the current sample values of its neighbors.
(5) Store the new sample value of *n* in *C(n)*.
(6) For each neighbor *n'* of *n*,
(7) If *C(n')* ≠ *C(n)*, also update the sample value of *n* in *C(n')*.
(8) End
(9) End

From Theorem 3.1, we can see that for a node n in a chunk, both its local dependencies and global dependencies are characterized in its main chunk's MRF model. The Gibbs sampling algorithm exactly samples a node from its main chunk. Through sharing cliques (dependency edges), the sampling procedure preserves the dependencies of nodes across two neighboring chunks. Note that the cliques in MRF are typically small, due to the common simplifying assumption that correlations are largely local.

By performing multiple rounds of Gibbs sampling, we evaluate the query on the deterministic input values using the semantics of conventional databases. Later, we discuss how we collect the results from the multiple rounds and assemble the result for the original query.

3.1.2.2 What we return as results

Because result tuples will likely be correlated, the results of query evaluation must reflect this correlation. To achieve this end, each result tuple is augmented with a bit vector which delineates which sampling rounds yielded that tuple. For example, if N = 5 and tuple RT1 was produced in the first and fourth round while tuple RT2 was produced in the first and third round, RT1 will be augmented with the bit string 10010 and RT2 will be augmented with 10100. Now, Pr(RT1) can be obtained by taking the cardinality of the RT1 bit string and dividing by N. In this case Pr(RT1) = 0.4 and Pr(RT1) = 0.4. Pr(RT1, RT2) can be obtained by performing the logical AND of the two bit strings and dividing the cardinality of the result by N. In this case, Pr(RT1, RT1) = 0.2 as opposed to 0.16 which would be the result of multiplying the two marginal probabilities together. Performing query evaluation through Monte Carlo sampling thus provides a powerful framework that can handle and express correlated uncertainty in tuples.

Many conventional APIs such as ODBC and JDBC require that results are retrievable one tuple at a time. Notice that by augmenting each result tuple with these bit strings, we can adhere to this standard and still allow the user to recover the entire joint distribution if necessary. The bit-array is quite compact and bit-wise operations are very fast on today's hardware. In this fashion, the user can generate joint distributions on an arbitrary number of result tuples that are consistent with correlated errors present in the input tuples as described earlier.

As discussed earlier, besides the result tuple membership distribution, we also describe the distribution of uncertain fields in a tuple. This can be accomplished with histograms (e.g., [GZ08]).

3.2 A-trees

3.2.1 A-tree Structure and Basics

In this section, we first describe the A-tree structure and how it encodes the joint distribution of array cells. We then discuss its probabilistic graphical model.

3.2.1.1 Background

A positional tree is a tree in which the children of a node are labeled with distinct positive integers. A k-ary tree is a positional tree [CLRS] in which for every node, all children with labels greater than k are missing. Thus, a binary tree is a k-ary tree with k = 2. Figure 3.1 shows a k-ary tree with k = 4. Some children of a node can be missing, making its degree less than k. Note that an ordered tree in contrast to a positional tree, is one in which the children of each node are simply ordered (but not labeled with unique integers). For example, node N in Figure 3.1 is missing its third child. If instead we move the subtree at its second child to the third position and let it eliminate the second child, then it becomes a different positional tree (still k-ary), but it would be the same ordered tree.

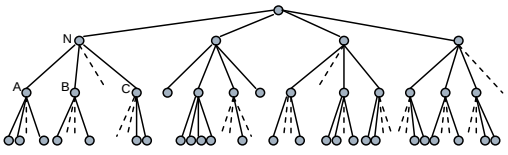


Fig. 3.1: Example of a k-ary tree with k = 4.

3.2.1.1 Basic A-tree Structure

An A-tree is a k-ary tree with the degree k being 2d, where d is the number of dimensions in which the uncertain value is correlated. Note that d is typically small (most often 1, 2, or 3). Thus, it is a binary tree when d = 1 and a 4-ary tree when d = 2, and so on. Figure 2 shows an example partition

for d = 2. Throughout this section, we use d = 2. This can be easily extended to other dimensionalities. We recursively divide an array in half along each of the d dimensions. In Figure 3.2, the first partition (thick dotted lines) divides the array space into four (k = 22) subspaces. The whole array maps to the root of the 4-ary tree in Figure 3.1, and the four subspaces map to its four children in some fixed order (e.g., 1st child is the north-west subspace, 2nd child is the south-west one, etc.). Then recursively, we again partition each of the four subspaces into four, which map to the four children of each node at the level below the root in Figure 3.1. Thus, a recursive partition of the array space corresponds to a top-down traversal of the k-ary A-tree from one level to the next. Eventually, at the leaf level, each leaf corresponds to four neighboring cell values of the array. In Figure 3.2, array cells A, B, C, and D together form a leaf.

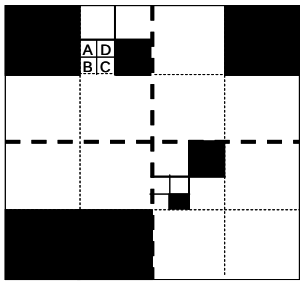


Fig. 3.2: Illustrating recursive partitioning of a two-dimensional array. The joint distribution of the uncertain attribute is encoded in a 4-ary tree.

For now for simplicity of exposition, in the case of d > 1, we assume that each of the dimensions has the same size. We also assume this size is 2n (for some integer n). We extend it to more realistic scenarios later. The black blocks in Figure 3.2 indicate the empty regions (NULL values) of the array that do not have values in the A-tree and, thus, correspond to the "missing" children in a 4-ary A-tree. Thus, arrays of arbitrary sparsity can be accommodated. Here is how a joint distribution is encoded in an A-tree:

- Each leaf in an A-tree stands for the average value (a random variable) of the four neighboring cells it maps to. For example, in Figure 3.2, one of the leaves is X = (A+B+C+D)/4. Each internal node stands for the average of its four children (a random variable). Equivalently, each node stands for the average of all array cells covered by its subtree.

- A leaf stores the joint distribution of the four cells it maps to, relative to (i.e., conditioned on) their average value (which is the random variable that this leaf stands for). There are two ways to specify

this joint distribution "relative to" the average, which we discuss in Section 3.2.3.1. For example, in Figure 3.2, one of the leaves contains the joint distribution of A, B, C, and D, conditioned on their average X.

- Recursively, in a bottom-up manner, an internal node encodes the joint distribution of its four children, relative to their average.

- In addition to this joint distribution, the root node also holds the distribution of the average value of the whole array.

Note that the average of children is weighted. For example, in the A-tree of Figure 3.1, node N contains the joint distribution of its three children (one child node is missing), A, B, and C, relative to their average value (nA·A+nB·B+nC·C) / (nA+nB+nC), where nA is the number of non-empty cells (i.e., not NULL) in the subtree rooted at A; similarly for nB and nC.

We describe the implementation details of the encoding of the distributions at each node in Section 3.2.3. The underlying idea of A-trees is that we model the joint distribution of cells in a manageable way that is relatively compact and automatically structured. The automatic structure is based on the principle of the locality of data correlation: closer cells are more likely correlated. We organize cells into hierarchical clusters according to proximity, each of which contains a small number of random variables so that we can encode their joint distribution compactly. In subsequent sections, we analyze and experimentally verify the A-tree's graphical model, modeling accuracy and efficiency for query processing.

An interesting aspect of the A-tree approach is that if we simply trim the leaves of an A-tree, the remaining A-tree represents the distribution of an array with a coarser grain. This renders a fast approximation of the data and may be meaningful for many applications that demand rapid results (e.g., real-time processing or on-line computation). For the example of image and sound, object or pattern recognition algorithms can work in the coarser level. This can also be beneficial for queries on sensor networks and network routing.

Extensions of the Basic A-tree Structure

We now discuss some extensions of the basic A-tree structure.

3.2.1.2 Extensions of the Basic A-tree Structure

3.2.1.2.1 Arbitrary Dimension Sizes

For ease of exposition, previously we assume that each dimension has size 2n (for some integer n). However, in reality, dimensions may have different sizes and they may not be a power of 2. We can partition the array in a similar fashion. Recall that the recursive partition of an array divides each dimension in half every time. We note two cases:

- We do the same even if a dimension is not a power of 2. When we have to divide a dimension of an odd size 2k + 1. We simply divide it into pieces of size k and k + 1.

- When two dimensions do not have the same size, the "short" dimension must first reach size either 2 or 3 in the recursive partition procedure. At this point, we stop partitioning the short dimension but continue dividing the long dimension in halves, until the long dimension also reaches size 2 or 3. Now we have three combinations of block shape: 2 by 2, 2 by 3, and 3 by 3. As illustrated in Figure 3.3, the first case is the same as the basic A-tree; a final partition for the second case gives us a 1 by 2 and a 2 by 2 block; a final partition for the third case gives us 1 by 2, 1 by 3 and 2 by 2 blocks. Then the final joint distributions are on these blocks.

Note that we are still able to keep track of the dimension ranges that each node of an A-tree covers.

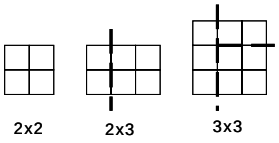


Fig. 3.3: Three combinations of final block shape and their partitions.

3.2.1.2.2 Basic Uncertainty Blocks of Arbitrary Shapes

We define a basic uncertainty block of an array as a box (e.g., a rectangle for two-dimensional arrays) in the array inside which cells have the same distribution. In the basic A-tree, each array cell is a basic uncertainty block. This is the smallest basic block size possible. However, in many applications, this granularity is not necessary and the basic block size can be much larger. Having a larger basic block size makes the representation more succinct and query processing more efficient.

For example, astronomers take photo images of objects in the universe. Due to precision limits, pixels of an image, as cells of a two-dimensional array, exhibit correlated uncertainty in values. A block of

neighboring pixels, due to their proximity, is likely to have the same error distribution. Thus, a basic uncertainty block can be, say, 50 cells by 50 cells in size. Now each basic block is treated as a "single cell" in the A-tree, which only records a single distribution of the "random" part of the pixel values. Each basic block will then store a 50 by 50 block containing the "deterministic" parts of the pixel values. Combining the deterministic and the random parts together gives a true pixel value.

3.2.1.2.3 Initial Partition of an Array

The best initial partition of an array is application specific and a knowledgeable user can define the initial partitions. In the astronomers' photo example, different regions of an image may have different levels of uncertainty. Some parts of the image (e.g., towards the center) are clearer and have less uncertainty, while some parts (e.g., towards the borders) are blurrier and have more uncertainty. Thus, one may want to first partition the array into rectangular regions and assign different basic block sizes for different regions: regions towards the image center have larger basic uncertainty blocks and the distributions there have smaller variances, while regions at the borders need finer basic blocks. This is illustrated in Figure 3.4.

Since correlation among regions may be very weak, an application program can either declare region summaries (i.e., average values) to be independent or let the system manage the joint distribution of the regions as the upper levels of the A-tree. When regions are independent, each of them is a separate A-tree.

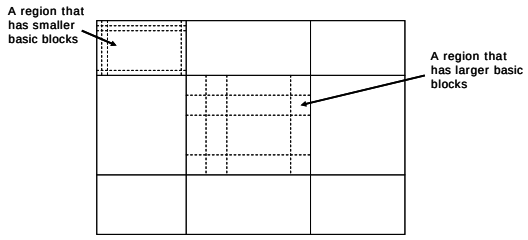


Fig. 3.4: Illustrating the initial partition of an array into nine regions.

3.2.1.2.4 A-tree's Probabilistic Graphical Model

A-tree is a unified structure for both the storage model and the probabilistic graphical model. A probabilistic graphical model (PGM) is a diagrammatic representation of a probability distribution [B06]. It provides a simple way to visualize the structure of a probabilistic model and gives insights into its properties, including conditional independence properties. Complex computations, required to perform inference, can be expressed in terms of graphical manipulations. In a PGM, each node

represents a random variable and edges express probabilistic relationships between these variables. The graph captures the way in which the joint distribution over all of the random variables can be decomposed into a product of factors, each depending only on a subset of the variables.

There are two major classes of PGM's: Bayesian Networks (BN) and Markov Random Fields (MRF). BN's edges are directed, while MRF's edges are undirected. Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to expressing soft constraints between random variables.

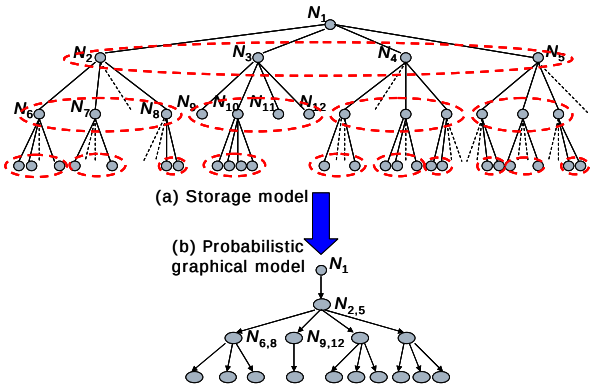


Fig. 3.5: An A-tree encodes a unified storage model (a) and probabilistic graphical model (b). There is a natural conversion from (a) to (b). The root node (N1) stays unchanged. Shrink its children (N2 to N5) into one node, indicated by the dotted ellipse in (a) and N2,5 in (b). The four edges connecting N1 with N2 … N5 in (a) are shrunk into one directed edge in (b). The similar procedure applies to other nodes and we get a Bayesian Network in (b).

The PGM of an A-tree is essentially a Bayesian Network, as illustrated in Figure 3.5. There is a natural mapping from the storage model of an A-tree (Figure 3.5a) to its graphical model (Figure 3.5b). In a nutshell, we need to shrink the multiple children of an internal node (e.g., N2 to N5 in Figure 3.5a) into one composite node in the graphical model (denoted as N2,5); corresponding edges are also merged. This is needed because A-trees encode P(N2,5|N1), but not P(N2|N1), etc. Each edge in Figure 3.5b corresponds to a joint distribution in a node of the A-tree.

An interesting observation here is that originally only the leaf level exists and represents real random variables (each leaf maps to some cells of the array). All nodes (random variables) at upper levels of the Bayesian Network are artifacts of our construction. They are derived random variables. Interestingly, the construction of an A-tree is bottom-up (Section 3.2.3), yet the inference is top-down (Section 3.2.4). Note that a node in the graphical model of an A-tree can be composite, denoting several nodes of the A-tree.

3.2.2 Analysis

We now analyze how accurately an A-tree can model the joint distribution. We do this from three different perspectives: entropy preservation, distribution function preservation, and the expressiveness of neighbor correlation.

3.2.2.1 Entropy Preservation

Entropy preservation is a way to measure how faithfully the actual joint distribution is depicted in a real encoding. To illustrate with a simple example, when the correlation of cells is not modeled at all, but we only encode the marginal distributions for each cell, then the entropy of the whole array is the sum of the entropy of each cell, which can be a lot bigger than the entropy of their actual joint distribution. At the other extreme, when all cells are perfectly correlated, the entropy of their joint distribution is just the entropy of one cell. Closeness in entropy gives strong evidence that the distributions are close.

Theorem 3.2: *Assuming two nodes (with different parents) in the same level of an A-tree are conditionally independent given their parent values, the entropy of the joint distribution given by the encoding of an A-tree is equal to the entropy of the actual joint distribution of the array cells.*

Proof: We present the proof for the two dimensional case. It can be generalized to any dimensionality. Suppose an A-tree has n leaves. Define random variables X1, X2, …, Xn for the values of the leaves. Then the joint distribution on X1 to Xn is the joint distribution on all array cell values. This is shown in the base level of the illustrative A-tree in Figure 3.6. In the second level, denote the summation (i.e., average) of X1 to X4 as X1,4, etc. Thus we have X1,4, X5,8,… at the second level, X1,16, X17,32,… at the third level, and so on. Finally the root is a single variable X1,n. We first observe the Markov property of the levels of an A-tree.

Lemma 1: The Markov property exists among levels of an A-tree. Specifically, suppose the A-tree has h levels. Denote the set of random variables in the first level (X1 to Xn) collectively as L1, the set of random variables in the second level (X1,4 to Xn-3,n) as L2,…,and finally, the root level X1,n as Lh. Then, $Pr[L_i \mid L_{i+1},\ldots,L_h] = Pr[L_i \mid L_{i+1}]$, for $1 \leq i \leq h-1$.

Lemma 1 is true because by definition Li+2 is completely determined given Li+1 (i.e., by taking averages), and Li+3 is completely determined given Li+2, and so on until we reach Lh. □

Now we start from:

$$H(X_1, X_2, ..., X_n) = H(L_1) = H(L_1, L_2, ..., L_h)$$
$$= H(L_h) + H(L_{h-1} \mid L_h) + H(L_{h-2} \mid L_{h-1}, L_h) + ... + H(L_1 \mid L_2, ..., L_h)$$

The second equality follows from the fact that L1 completely determines L2, L3,…, Lh and the third equality is due to the chain rule of entropy [CT91]. Applying Lemma 1 to this, we have

Now we consider one of the terms on the right hand side of equation (1):

$$H(L_i \mid L_{i+1}) = H(V_1, V_2, ..., V_t \mid S_1, S_2, ..., S_t) \qquad (2)$$

where S1, S2,…, St are the random variables for the t nodes at level i + 1, and their corresponding sets of children at level i are V1, V2,…, Vt , respectively (Figure 3.6). We have

$$H(V_1, V_2 \mid S_1, S_2) = H(V_1 \mid S_1, S_2) + H(V_2 \mid S_1, S_2) - I(V_1; V_2 \mid S_1, S_2)$$

where I(V1;V2|S1,S2) is the mutual information of V1 and V2 given S1 and S2 [CT91]. From the assumption of the theorem, we know that V1 and V2 are independent given S1 and S2. Thus, I(V1;V2|S1,S2) = 0. By the same token, we can rewrite equation (2) as

$$H(L_i \mid L_{i+1}) = \sum_{j=1}^{t} H(V_j \mid S_1, S_2, ..., S_t) \qquad (3)$$

Combining (1) and (3), we have

$$H(X_1, ..., X_n) = H(L_h) + \sum_{i=1}^{h-1} \sum_{j=1}^{t_i} H(V_j \mid S_1, S_2, ..., S_{t_i}) \qquad (4)$$

Note that the left hand side of (4) is the entropy of the actual joint distribution of the cells while the right hand side is the entropy of the distribution given by the encoding of the A-tree (the first term is the entropy of the distribution of the average of the whole array stored at the root; each term in the sum is the entropy of a joint distribution encoded at each node of the A-tree and they are independent). Thus the theorem is proven. □

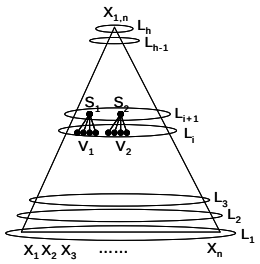


Fig. 3.6: Illustrating the top-down inference of the joint distribution in an A-tree. Each ellipse represents a level of the tree. Each internal node has 4 children. S1 is a node at level i + 1 and V1 is its set of children at level i.

3.2.2.2 Distribution Function Preservation

Similar reasoning applies to the distribution function itself and we further have:

Theorem 3.3: *Assuming two nodes (with different parents) in the same level of an A-tree are conditionally independent given their parent values, the joint distribution resulting from the encoding of an A-tree is the same as the actual joint distribution of the cells in the multidimensional array.* □

We will not show the proof as it is very similar to that of Theorem 3.2. The basic idea is that we use the Markov property in Lemma 1 and the fact that

$$\begin{aligned}\Pr(x_1, x_2, ..., x_n) &= \Pr(l_1) = \Pr(l_1, l_2, ..., l_h) \\ &= \Pr(l_h) \cdot \Pr(l_{h-1} \mid l_h) \cdot \Pr(l_{h-2} \mid l_{h-1}, l_h) ..... \Pr(l_1 \mid l_2, ..., l_h) \\ &= \Pr(l_h) \cdot \Pr(l_{h-1} \mid l_h) \cdot \Pr(l_{h-2} \mid l_{h-1}) ..... \Pr(l_1 \mid l_2)\end{aligned}$$

Intuitively, the assumption of the theorems roughly says that if two nodes are far away (with different parents), then they are less correlated and we treat them as conditionally independent given their local summaries (parents). Clearly, this assumption trades off precision for efficiency. Our analysis and experiments show that the precision loss is insignificant and A-trees model correlations reasonably well.

3.2.2.3 Expressiveness of Neighbor Correlation

An A-tree expresses neighboring correlations in the joint distributions at different levels of the tree. Clearly, the correlation between two cells is easier to encode when this level is lower. In this section, we demonstrate that, from the perspective of any random query, the average level where cell correlation is encoded is low. This indicates that an A-tree is not only efficient for inference (Section

3.2.3), but it also has great power in modeling correlations. We further experimentally verify this in Section 3.2.5.

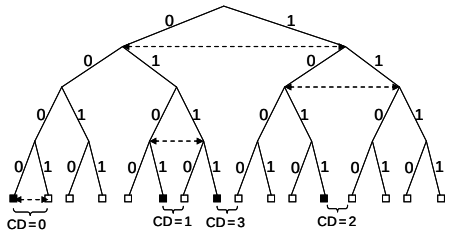


Fig. 3.7: Illustrating cluster distance in a binary A-tree.

Definition 3.1 (*neighboring cells and cluster distance*): Two *neighboring cells* of an array are two cells that are next to each other in one dimension and have the same dimension values in other dimensions. Starting from the cell level (leaves) as level 0, the *cluster distance* (CD) between two neighboring cells is the level in the A-tree at which a joint distribution between their cluster summaries exists. □

Figure 3.7 shows an example of pairs of neighboring cells with CD 0, 1, 2, and 3, respectively. We can see that the CD between two cells is determined by the level below their lowest common ancestors.

When CD is 0, the correlation between two cells is directly modeled; when CD gets bigger, their correlation is embodied in the summaries of bigger clusters they are in. We next quantify the average as well as the maximum CD in the set of cells that an arbitrary query accesses.

Theorem 3.4: *Consider a binary A-tree (d = 1) of height h. Suppose a query references a random part of the array that has q pairs of neighboring cells (either in a contiguous range or scattered in the array). Then the expected average CD is 1+(h+1)/2h and the expected maximum CD of the q pairs is . For d =2 (4-ary A-tree), the expected average CD is the same and the expected maximum CD is where q1 and q2 are the number of neighboring pairs along the two dimensions and q = max(q1, q2).*

Proof: Consider the simpler case of d = 1. We first note the following lemma.

Lemma 3.2.2: *We label every left branch of a binary A-tree with 0 and every right branch with 1. We then label each cell of the array with the concatenation of labels on the path from root to the cell. Then the CD between a cell and its right neighbor is simply the number of trailing 1's in its label.* □

Lemma 2 is a simple property of a binary tree and is illustrated in Figure 3.7. The first cell from the left has label 0000, the second has 0001, and so on, which is essentially a counter. Figure 3.7 shows the cases that CD = 0 to 3. Simply from the labels of the cells marked black we can determine its CD with its right neighbor.

Now consider the expected average CD. The label of a random cell comes from a random walk from the root to a leaf. Thus, Pr[zero trailing 1's] = ½, Pr[one trailing 1's] = ¼, etc. Let random variable A denote the average CD of the random q pairs. Then, from the linearity of expectation and Lemma 2, we have

$$E[A] = \sum_{i=1}^{h-1} i \frac{1}{2^{i+1}}$$

With some algebraic manipulation, which we omit, we get

$$E[A] = 1 + \frac{h+1}{2^h} \qquad (1)$$

We next compute the expected maximum CD. Let random variable X denote the maximum CD of q random pairs. Then we have,

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] = \log q + \sum_{i=1+\log q}^{h-1} q\left(\frac{1}{2}\right)^i = \log q + 1 - \frac{q}{2^{h-1}} \qquad (2)$$

The first equality is due to the fact that X is nonnegative (intuitively, for i from 1 upwards, cumulatively, Pr[X ≥ i] is the probability that we add 1 to the expectation) [MU05]. q(½)i is the probability that any of the q pairs (hence the maximum) has CD at least i. This is effectively 1 for the first logq terms, hence the second equality in the equation above.

Next we consider the case of d = 2. Labeling a 4-ary tree is similar. Each edge is now associated with a 2-bit label, indicating the "left or right" decision for the two dimensions respectively. Thus, four children of a node have labels 00, 01, 10, and 11. To think about it another way, as a random walk is performed from the root to a leaf, we are in fact doing a random walk on two binary trees with the same height, one for each dimension. For a pair of neighboring cells along one dimension of the original 4-ary tree, they are next to each other in the binary tree of that dimension and are on the same

leaf cell in the binary tree of the other dimension. From (1) we know that the expected average CD only depends on the height of the trees, but not q1 or q2. Thus, it is the same as in d = 1.

Let random variable Z denote the maximum CD; let random variables X and Y denote the maximum CD of the q1 pairs along one dimension and that of the q2 pairs along the other dimension, respectively. Thus, Z = max (X, Y). Similar to the reasoning in (2), we have

$$E[Z] = \sum_{i=1}^{\infty} \Pr[Z \geq i] = \log q + \sum_{i=1+\log q}^{h-1} \left[1 - \left[1 - q_1 \left(\frac{1}{2} \right)^i \right] \left[1 - q_2 \left(\frac{1}{2} \right)^i \right] \right]$$
$$= \log q + \sum_{i=1+\log q}^{h-1} \left[(q_1 + q_2) \left(\frac{1}{2} \right)^i - q_1 q_2 \left(\frac{1}{4} \right)^i \right]$$
$$= \log q + (q_1 + q_2) \left[\frac{1}{q} - \left(\frac{1}{2} \right)^{h-1} \right] - \frac{q_1 q_2}{3} \left[\frac{1}{q^2} - \left(\frac{1}{4} \right)^{h-1} \right]$$

This completes the proof of Theorem 3.4. □

In the same vein, we can obtain the CD's for larger d values. Theorem 3.4 indicates that the expected average CD asymptotically approaches 1 from the perspective of any random incoming query. This shows that A-trees can efficiently express correlations of neighbors. We also experimentally verify its accuracy in modeling the underlying true distribution in Section 3.2.5.

3.2.3 Implementation Details

We now look at some details of an A-tree, in particular, the representation of the joint distribution in each node and the layout of an A-tree on disk.

3.2.3.1 Join Distribution at a Node

Previously, we stated that a node encodes the joint distribution of its four children, relative to their average. We now elaborate on this and describe how to encode the joint distribution.

As discussed earlier, each node stands for the average of all cells in its subtree. Since each cell value is a random variable, so is each node value. Thus, we are trying to specify a joint distribution of X1, X2, X3, and X4, relative to a random variable Y (the average of X1 to X4), i.e., the joint distribution of the children (X1 to X4) given their parent's value (Y). But since X4 is completely determined given Y, X1, X2, and X3, we only need to specify the joint distribution of X1, X2, and X3, relative to Y.

The joint distribution relative to Y can be represented either (1) as a joint distribution of multiplicative factors, or (2) as a joint distribution of additive offsets. In the first method, we have Xi = Y (1 + Fi),

for $1 \leq i \leq 3$, where Fi is a multiplicative factor. We then simply encode the distributions of F1, F2, and F3. In the second method, we have $Xi = Y + Oi$, where Oi is an additive offset, and we just encode the distributions of O1, O2, and O3. We use an equiwidth histogram for both methods. Thus, they are similar and we only describe the first method.

Each of the Fi will have a range. There is a parameter k (e.g., $k = 8$) indicating the number of equiwidth intervals for each Fi. To represent the full joint distribution of all combinations of intervals, however, a complete distribution table has k^3 entries (three random variables), which is too costly. Instead, we observe that for any interesting correlation among the children (be it positive or negative), because the entropy is smaller than if they were independent, most of the k^3 entries would have small probabilities, and only a few entries with the highest probabilities are significant. Thus, we only store the n_τ entries with the highest probabilities, where n_τ is a user-specified threshold parameter. Each entry has a bit number for each Fi. It also has an l-bit number to represent the probability. For example, when $l = 4$, a probability number is a multiple of 1/16.

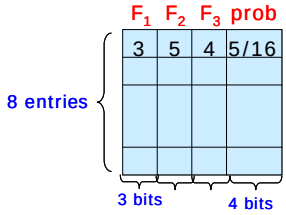


Fig. 3.8: An example of a joint distribution table at a node.

Figure 3.8 shows an example in which there are at most 8 entries ($n_\tau = 8$). Each entry has a 3-bit number (to represent one of the $k = 8$ intervals) for each of F1, F2, and F3. Each entry also has a 4-bit probability number ($l = 4$), thus making the probability value a multiple of 1/16. We can see that the distribution table is quite compact.

Finally, recall that the root also holds the distribution of the average value of the whole array. This can either be a simple one-dimensional histogram or a well-known distribution (e.g., Gaussian).

In general, obtaining a joint distribution is highly application specific. There are statistical methods to do this [B06, J98] and it is outside the scope of this paper. Having said that, we show a simplified example of a specific application on how one might get the distributions in an A-tree. Recall the sensor readings example in Section 1. Suppose the data in the array are temperatures at different

locations in the space. However, the readings in the array are outdated and we have some uncertainty about what the current values are. The basic idea is that we "learn from the history". We look at logs for readings in the past, and figure out what correlation we can assume.

Time	X_1	X_2	X_3	X_4	Y	F_1	F_2	F_3
t*	72	73	71	74	72.5	-6.9	6.9	-20.7
t1	68	69	71	72	70	-28.6	-14.3	14.3
t2	69	71	69	71	70	-14.3	14.3	-14.3
t3	71	72	71	73	71.8	-11.1	2.8	-11.1
t4	77	75	76	78	76.5	6.5	-19.6	-6.5
t5	80	80	78	82	80	0	0	-25
t6	76	77	75	78	76.5	-6.5	6.5	-19.6
t7	72	73	71	74	72.5	-6.9	6.9	-20.7
t8	78	77	75	78	77	13	0	-26
t9	81	83	80	84	82	-12.2	12.2	-24.4

Fig. 3.9: Sensor readings history data.

Time	F_1	F_2	F_3	d
t*	4	6	1	
t1	0	1	7	15
t2	2	7	2	4
t3	3	5	2	3
t4	6	0	3	10
t5	5	4	0	4
t6	4	6	1	0
t7	4	6	1	0
t8	7	4	0	6
t9	3	7	0	3

(a)

F_1	F_2	F_3	prob.
4	6	1	0.25
5	4	0	0.25
7	4	0	0.25
2	7	2	0.25

(b)

Fig. 3.10: Normalized data (a) and the final distribution (b).

We focus on four cells of the array. The highlighted first line in Figure 9 indicates the data in the array. X1 to X4 are the values of four neighboring cells. Y and Fi's are computed as described earlier. The Fi values in Fig. 9 have a scale factor of 10-3. Our log contains readings in the past, at time t1 through t9. Our goal is to learn the correlation (distribution on Fi's) from the past. We first normalize the Fi's into interval numbers (0 to 7), as in Figure 10(a). There are many ways to learn the distribution. For example, one can compute the L1 distance (Manhattan distance) between data entries in the past and the entry in the array (first line in Fig. 10a) and find four entries that have the smallest distance. This is shown in the last column of Fig. 10(a) as those four rows are highlighted. As a simplified illustration, we can use the Fi values in the four rows above them (i.e., the time instances after those entries that are closest to the values in the array) as entries in the joint distribution table and assign probability 0.25 to each (Figure 10b). Likewise, we can do this for nodes in the A-tree at all levels.

3.2.3.2 Layout on Disk

Typically, scientific data (e.g., astronomical images) is rarely updated. The data is mostly read-only. Our goal of managing an A-tree on disk is thus to make it as compact as possible and read-optimized.

We propose to linearize an A-tree in level-order: starting from the root level and descending one level at a time, nodes from left to right at each level are stored on disk in that order. Figure 11 shows an

example in which we store the nodes in the numbered order bypassing the missing children. Note that as with any positional tree, we must record the information about which children are missing: we need that to determine cell locations.

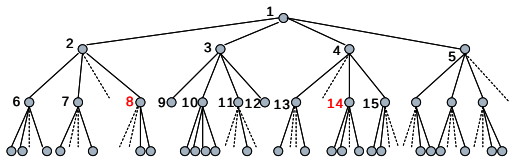


Fig. 3.11: Illustrating a level-order storage of an A-tree on disk.

An advantage of storing nodes in level-order is that we only need to store the pointer to its first child at a node, as opposed to storing one pointer for each child. This is because other children must be stored immediately after the first child, likely in the same page. This makes the structure more compact. For example, in Figure 11, node 3 only needs to store the pointer to its first child, node 9; other children immediately follow node 9.

3.2.4 Query Processing

In this section we discuss techniques of doing query processing on multi-dimensional arrays with uncertain attributes represented as A-trees. We first look at processing general queries and then consider optimizations for COUNT, AVG, and SUM queries.

3.2.4.1 Queries in General

Scientific applications are often computationally intensive and tend to use a different set of operators (e.g., dot products, matrix multiplications). The design of an array database system must take these operators into consideration [SB07, MS02, BD98, CA98]. The complex nature of the query operators complicates the task of probabilistic inference with graphical models. Consequently, often the most viable method of probabilistic inference is through Monte Carlo algorithms [B06, J98]. This requires random sampling from graphical models. We first describe the sampling algorithm from an A-tree given an incoming query. We then demonstrate the efficiency of doing inference using A-trees by comparing with an alternative MRF model.

3.2.4.1.1 Sampling

Sampling from an A-tree is an efficient top-down traversal (logarithmic-length path), shown in Figure 12. It is an application of the ancestral sampling technique [B06] on the Bayesian Network in Figure 5(b). The tree structure allows us to limit the sampling to the path from the root to the target cells Q,

and nothing else. Note that from the recursive partition of the array dimensions during A-tree construction, it is easy to determine the range of dimension values associated with each node. Step (6) in the algorithm uses such information to determine if there is an overlap between the coverage of a node and the set Q.

Let us look at an example. Consider the following astronomy query:

Q1: **SELECT AVG**(brightness) **FROM** Space_image

WHERE DISTANCE(x, y, z, ?, ?, ?) < ?

Q1 asks for the average brightness within a certain distance of some object, whose coordinates are specified in the three parameters marked by "?". Space_image is a three dimensional array. DISTANCE is a built-in function that calculates the distance between two positions. The most effective known method of probabilistic inference for such a query on a graphical model is based on Monte Carlo (MC) algorithms [B06]. Our array system optimizer will compute a minimum bounding box that contains the ball selected by the WHERE clause. The bounding box is a first approximation of the set of cells Q, as input to the sampling algorithm in Figure 12. The sampling algorithm starts from the root and traverses down the tree, targeting only the bounding box Q, which is eventually refined to the actual ball that the WHERE clause selects. Note that our optimizer will obtain all the samples of a cell needed by MC (say, 100 samples) at the same time because they are independent. Thus, we only need to traverse down the tree once, thereby saving I/O costs. This is in contrast to sampling from MRF (next section), in which we cannot use this optimization because sample rounds are correlated and must occur in sequential order.

Input: An A-tree T, a set of cells Q accessed by a query.
Output: A set of samples S, one value for each cell in Q, from the joint distribution of T.

(1) At the root of T, from the *distribution of the average value of the whole array*, get a sample for the root.
(2) Initialize node set N = {root} (one node).
(3) **For** each node $n \in N$,
(4) Sample from the joint distribution at n, get sample values (v_1, v_2, v_3, v_4) for its four children, based on the sample at node n.
(5) **If** n is a leaf of T, then v_i ($1 \le i \le 4$) is for a cell c. If $c \in Q$, then v_i is the final sample for c.
(6) **Else** for each child c_i ($1 \le i \le 4$), if the range of dimension values covered by c_i intersects Q, then add c_i to N.
(7) **End for**

Fig. 3.12: An algorithm to get samples for a set of array cells from an A-tree.

3.2.4.1.2 Comparison with an MRF model

One may wonder what would result if we just model a multi-dimensional array with a simple lattice structure Markov Random Field to capture the neighborhood correlation, as shown in Figure 13(a). However, the problem here is the high computational cost. How big is the MRF model? Ideally, it should span the whole array so that all the local correlations between all pairs of neighboring cells are captured by the model. However, the computation cost of sampling a big MRF is high, as we illustrate next.

The corresponding inference algorithm for an MRF is Markov Chain Monte Carlo (MCMC) [B06]. Gibbs sampling [B06] is often used with MCMC on an MRF. It has to iterate though all the nodes in a model to create one sample, even though the query may only need to access a tiny fraction of the cells of the whole array. Gibbs sampling uses a so-called visitation schedule to update the samples of each node in the graph to produce one sample from their joint distribution. This is because all nodes are either directly or indirectly connected, and thus the sample value of each node in the graph is needed to produce the next round of samples. Therefore, the sampling is rather wasteful for answering a query.

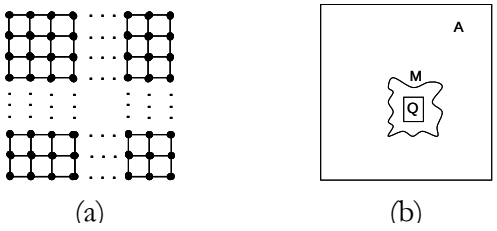


Fig. 3.13: Illustrating MRF construction for a two dimensional array. (a) indicates a simple grid structure. (b) illustrates a box Q actually needed for answering a query inside array A. An MRF over an arbitrary region M that contains Q is used.

Now suppose we do not use an MRF model for the whole array A. Instead, we have an MRF model built over a small region M (of any shape) inside A and M contains Q, the set of cells accessed by the query. This is illustrated in Figure 13(b). We could use the model over M to give an approximate answer to the query. However, the area Q accessed by some incoming query can be arbitrary, and it would be impractical to dynamically build (learn) a model on the fly at execution time or to have a sufficient number of pre-built models.

Furthermore, our A-tree sampling algorithm based on ancestral sampling over Bayesian Networks is much more efficient than MCMC sampling, which requires a mixing time before its samples can be used (i.e., the Markov chain needs to get to a stationary distribution first; a.k.a. "burn in period") [B06]. Rigorous justification of inference results would require a theoretical bound on mixing time,

and many interesting practical cases have resisted such theoretical analysis [B06, J98]. A Markov chain may converge very slowly to its stationary distribution, requiring a long mixing time. Later, we further experimentally study the impact of the mixing time of MRFs on result accuracy and speed.

Finally, MCMC sampling requires the samples to be correlated (forming a Markov chain) and in a serial order. As a result, we cannot use the optimization of performing all sampling rounds concurrently to save I/O costs as we did for A-trees. For example, in answering Q1 (Sec 5.1.1), the system needs to follow the site visitation schedule and perform sample rounds one by one (each round obtains one sample for each cell in the bounding box Q).

3.2.4.2 COUNT, AVG, and SUM Queries

For sparse arrays, applications often query the COUNT, AVG, or SUM of "non-empty" cells (i.e., with a value in the A-tree) that fall within in a bounding box (i.e., a range in each dimension). It turns out that we can answer these queries very efficiently using the A-tree data structure.

We add an integer value (cell_count) to each internal node of an A-tree, recording how many non-empty cells there are in the subtree rooted at the node. The cell_count of all nodes can be easily obtained in a bottom-up manner during the construction of the A-tree. Next we introduce a definition.

Definition 3.2 (*minimum cover*): A *minimum cover* of a set of cells of an array is a set of nodes in an A-tree whose subtrees contain exactly the set of cells (no more and no less). Further, there does not exist another set of nodes that has this property but with fewer nodes in it. □

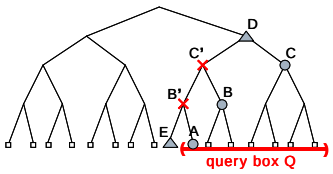


Fig. 3.14: Illustrating minimum cover and minimum cover with subtraction.

For example, in Figure 3.14, the minimum cover of the query bounding box Q (last seven leaves, or cells in the array) has three nodes: A, B, and C. Clearly, once we have the minimum cover of cells in a bounding box, adding up the cell_count in all nodes in the minimum cover gives us the COUNT of non-empty cells. This means that during query processing we can stop early at the minimum cover without going further down the tree. Nonetheless, one might wonder if this is the best we can do. In

Figure 3.14, for example, we could also use nodes D and E (cell_count in D minus that in E), which uses one fewer node. As we increase the tree height, the difference gets bigger. We call such a node set a minimum cover with subtraction. However, the following theorem shows that it does not really reduce the access cost.

Theorem 3.5: *In an A-tree stored on disk in level-order (as described in Section 3.2.2), using a minimum cover with subtraction to answer queries does not save I/O costs compared to using the minimum cover.*

Proof: Consider each node C in a minimum cover. First we claim that if a minimum cover with subtraction does not include C, it must include a node (say, E) in the subtree of at least one of C's siblings (say, C'). This is because at least one of C's siblings covers a cell not in the target set of cells, otherwise C and its siblings all cover cells in the target and their parent node would be in the minimum cover, but not C. The minimum cover with subtraction must include E in order to subtract that cell. For example, in Figure 3.14, for node C in the minimum cover, the minimum cover with subtraction must contain a node (E) in the subtree of node C' (C's sibling). The same is true with node B.

Thus, to access the minimum cover with subtraction, one must access node C' (since it is the only way to reach node E in the top-down access of the minimum cover as discussed earlier). In other words, for each node in the minimum cover, when we use a minimum cover with subtraction instead, we must either access that node, or one of its siblings. Because all siblings are stored contiguously in level-order storage, minimum cover with subtraction does not save costs. □

For an A-tree, we can easily find out, for each node, the range in each dimension of the array that it covers. Thus, the algorithm to compute the minimum cover MC for a set of cells Q is quite simple: Starting from the root, we check if the node covers only cells in Q. If so, we add this node to MC; otherwise we recursively check each of its children that has an overlap with Q.

For a COUNT of non-empty cells, we simply add up the cell_count in the nodes of MC and do not need to do anything extra. For AVG and SUM queries, however, we need to combine with the sampling technique described earlier. In Monte Carlo query processing, the sampling would be done together with our top-down procedure above to get an MC. Then we stop early at nodes in MC without sampling further down the tree. Let the sample value and cell_count at each node in MC be ai and ci, respectively ($1 \leq i \leq t$, where t is the cardinality of MC). Then the SUM and AVG for this

sampling round are and . Thus, for queries over large-scale datasets, many nodes in MC are at high levels and our optimization can significantly improve the performance.

3.2.5 Experiments

We perform a systematic empirical study on the following:

- How well does an A-tree model the underlying joint distribution? How does it compare with MRF?

- How efficient is query processing with A-trees? How does it compare with MRF?

- How much performance improvement do we gain from the optimization on aggregation queries?

- What is the space cost of A-trees?

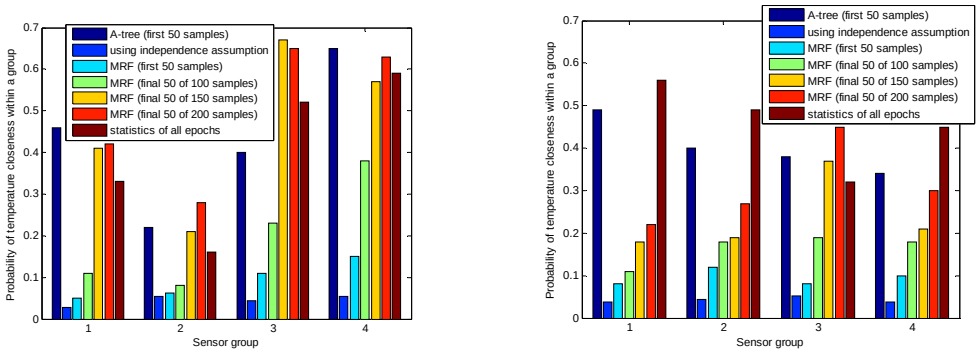
3.2.5.1 Setup

We perform experiments on the following two datasets:

- A real-world dataset: We use the publicly available Intel Lab dataset. It contains traces from a sensor network deployment which measures various physical attributes such as temperature, humidity, voltage of the sensors' batteries, etc. It uses the Berkeley Motes (sensor nodes) at several locations within the Intel Research Lab at Berkeley.

- A synthetic dataset: We also generate a dataset that is similar in nature to the Intel Lab dataset but can be arbitrary in size and sparsity.

We implement the A-tree construction and query processing algorithms presented in the paper. All



(a) **(b)**

Figure 3.15. Comparing the probability that four sensors within each group have close temperature values (within one degree of each other) using the real-world dataset (a) and using the synthetic dataset (b).

the experiments are carried out on a 1.6GHz AMD Turion 64 machine with 1GB physical memory and a TOSHIBA MK8040GSX disk.

3.2.5.2 Accuracy of Modeling the Underlying Joint Distribution

The Intel Lab dataset contains sensor readings that span about 65,535 epochs. We use the temperature readings from that dataset. An epoch is a monotonically increasing sequence number from each sensor. Two readings from the same epoch number were produced from different sensors at the same time. Suppose that resources are limited (e.g., sensor power consumption and communication cost to the server) making it impossible to get readings as frequently as we would like. Thus, temperature readings at missing time instances must be inferred and are uncertain. In the mean time, these uncertain temperatures at various sensors are likely to be correlated. We use A-trees to model the inferred readings at missing time points. This uncertain data forms a three-dimensional array with the third dimension being time. At each missing time instance, we have a grid of temperature values, some of which are missing. Using linear interpolation [DM06] from neighboring cells we can add more temperature values. We build an A-tree for each missing time instance and hence have an array of A-trees.

The joint distribution at each node of an A-tree is learned from a short period of time (100 epochs) before the time instance of the A-tree. The distribution (at the root) of the average value of the whole A-tree is estimated as a normal distribution with the mean being the average at the previous existing time instance and with a standard deviation of 1 degree. In order to test if the A-trees model correlations correctly, we first query the existing dataset and find four groups of sensors that have a

relatively high frequency, during all 65,535 epochs, of temperature readings within a range of one degree. Each group has four sensors. The first group has sensors at locations (60, 2), (60, 3), (61, 2), (61, 3) in the grid and the second group has sensors at (2, 27), (11, 24), (6, 32), (6, 33), etc. The x and y coordinates of sensors are in meters relative to the upper right corner of the lab space. We then arbitrarily pick an A-tree and query the probability that a group of sensors has close temperature readings (within one degree):

```
SELECT close_values (temperature, 1)
FROM lab_array
WHERE (x = ? AND y = ?) OR (x = ? AND y = ?)
        OR (x = ? AND y = ?) OR (x = ? AND y = ?)
```

close_values is a user-defined aggregation function that takes a set of temperature attribute values as the first parameter, and returns 1 if the set of values are all within a distance range of each other as specified in the second parameter (1 degree in the above query). The "?" marks in the query are placeholders for the positions of the four sensors in a group. Thus, using Monte Carlo query processing, we can compute the expected value of the result, which is the estimated probability that the group of four sensors have close values.

Figure 3.15(a) shows the result for the four groups of sensors at epoch 800. We retrieve 50 samples from the A-tree and compute the resulting probability. We execute the query for each sensor group. To compare with the result from an alternative graphical model of a lattice structured MRF, we build an MRF for each of the four sensor groups, as illustrated in Figure 3.13. Using Gibbs sampling and MCMC [B06], we compute the results of the four queries. For comparison, we also use the first 50 samples, as in the A-tree. As discussed earlier, due to the mixing time, the initial samples are not from the stationary distribution and thus are not of good quality. Therefore, we also experiment with 100, 150, and 200 samples respectively, but only use the final 50 samples to compute the result. We omit the initial samples in order to pass the mixing time, and always use 50 samples for fairness of comparisons.

Both A-trees and MRF's model the correlation in the joint distribution. We compare their results with that computed under the independence assumption (the second bar). Under the independence assumption, we assume each sensor reading has a normal distribution with the mean being its value at the previous existing time instance and its standard deviation being one degree, which is the same as

the root distribution of the A-tree. Finally, we also compare these results with the statistics collected over all epochs in the dataset (the last bar), which serve as an indication of the underlying actual joint distribution (for the result of this query).

From Figure 3.15(a), we can clearly see that A-trees model the underlying joint distribution very well in terms of the accuracy of inference results. On the other hand, the approach based on the independence assumption produces a very small probability result because it does not model the correlation among the sensors and thus, the probability that all four independent sensor samples are close to each other is small. The fact that we arrive at the correct results with A-trees verifies the well-structured correlation of the data. For the Markov chain sampling from MRF's, we can see that after about 100 samples (because of mixing time), it converges to a stationary distribution and the result is more accurate. Thus, sampling from these MRF's is not as efficient as sampling from A-trees. We further compare the query execution time in Section 3.2.5.3. Moreover, the results indicate that the modeling accuracy of A-trees is close to that of the MRF's. More importantly, as discussed in earlier (but not shown in this experiment), there is a serious problem with the MRF approach: it is difficult to have a small MRF model pre-built suited for every incoming query.

We next repeat this experiment with the synthetic dataset. Again we use four groups of sensors at different locations. The result is shown in Figure 3.15(b). This dataset again verifies our observations earlier. In fact, the result of using MRF is even worse with the synthetic dataset due to longer mixing time.

3.2.5.3 Execution Time

We now examine the execution time for answering the queries in Figure 3.15a. This verifies the efficiency of answering queries using A-trees compared to MRF's. The result is shown in Figure 3.16. As in Figure 3.15(a), we measure the execution time of answering the query by generating 50 samples from the A-tree. We also measure the execution time by generating 50, 100, 150, and 200 samples from MRF's (but only the last 50 samples are used for computing the result). As observed earlier, due to the mixing time of MCMC, the result of the query is accurate with 150 or 200 samples. Using MRF's is significantly slower than using A-trees to provide a result that has about the same accuracy. The synthetic dataset gives similar results.

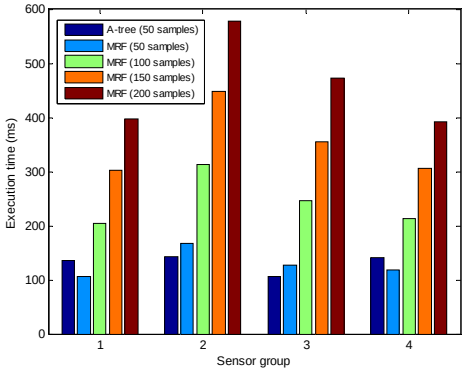


Fig. 3.16: Comparing the execution time of answering queries using A-trees vs. using different numbers of samples of MRF's.

3.2.5.4 Aggregation Queries

In the next experiment, we examine the performance improvement of the optimization using the minimum cover for COUNT, AVG, and SUM queries presented earlier. To arbitrarily control the data size, we use the synthetic dataset whose schema is the same as the Intel Lab dataset. We can programmatically control both the size of the array and the fraction of empty cells in the array. The array size is 32K by 64K (i.e., 2G cells) with half of them empty. We issue an aggregation query of the following form:

```
SELECT AVG(temperature)
FROM synthetic_array
WHERE x BETWEEN ? AND ?
    AND y BETWEEN ? AND ?
```

By controlling the parameters, we run the query over different numbers of non-empty cells. We compare the running times with and without the optimization presented earlier. In both cases, we perform 300 concurrent rounds of Monte Carlo sampling whenever we get to a node of the A-tree. This avoids going back to the node again and saves I/O costs. Figure 3.17 shows the comparison. We use a log scale on the y-axis of Figure 3.17 in order to show both lines clearly. The optimization is about two orders of magnitude faster because it only accesses the A-tree nodes on the path from the root down to the minimum cover, instead of accessing nodes all the way down to the leaves (as is the case without the optimization).

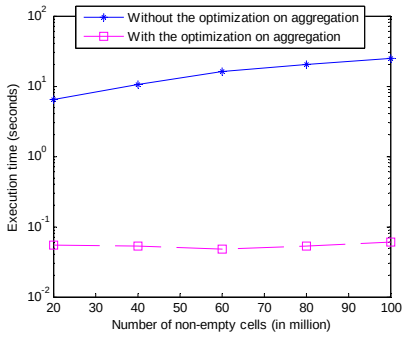


Fig. 3.17: Execution time comparison of an aggregation query with and without the optimization.

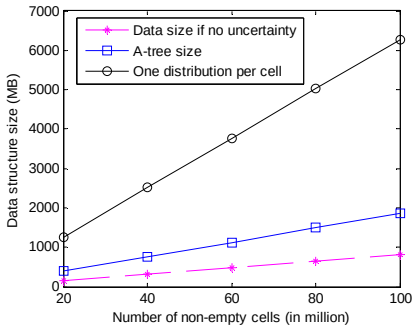


Fig. 3.18: A-tree sizes for different sizes of the underlying array.

3.2.5.5 Space Consumption

Using the generated synthetic dataset, we examine the space costs of A-trees. Figure 3.18 shows the details. The x-axis of Figure 3.18 indicates the number of non-empty cells of two-dimensional arrays with different sizes in which about half of the cells are empty. We compare the sizes of the A-trees with an obvious lower bound in which the data has no uncertainty at all. This lower bound is simply the product of a data value size and the number of non-empty cells. Note that in reality even for data without uncertainty, the storage size should be a little more than this lower bound, since one must store where the non-empty cells are located in the sparse array. Figure 3.18 shows that the A-tree sizes are a little more than twice the lower bound. We also compare with a naive approach in which an array stores one distribution per non-empty cell. This does not model the correlation between cells, and the sizes of the resulting arrays are significantly bigger than A-trees. Note that a lattice-structure MRF model for the whole array, which is too costly for query processing, would have a similar size because we need to store, at each cell, the conditional distribution of the cell on its neighbors for sampling. We also note that the space consumption for A-trees can be further reduced when the basic uncertainty blocks are bigger than single cells, as discussed earlier.

Chapter 4

SEMANTICS AND PROCESSING OF TOP-K QUERIES ON UNCERTAIN DATA

In this chapter, we describe our new semantics and two dynamic programming algorithms to answer top-k queries on uncertain data.

4.1 Problem Formulation

In this section, we present our data model and formal definitions of the Topk score distribution and c-Typical-Topk.

4.1.1 Data Model and Scoring Function

We follow the well-known tuple independent/disjoint data model from the probabilistic database literature [DS07, W05, SI07, HP08]. In this data model, an uncertain database D contains uncertain tables. An uncertain table T has an extra attribute that indicates the membership probability of a tuple in T. If a tuple's membership probability is p ($0 < p \leq 1$), it has probability p of appearing in the table and probability $1 - p$ that it does not appear. Table T also has a set of mutual exclusion rules. Each rule specifies a set of tuples which we call an ME group, only one of which can appear in T. If a tuple has no mutual exclusion constraint, we simply say that it is in its own ME group (of size 1). The sum of the probabilities of all tuples in an ME group should be no more than 1. The ME groups are assumed to be independent of each other.

A scoring function s takes a tuple t and return a real number s(t) as its score. In the previous work, the scoring function s is assumed to be injective (i.e., each tuple maps to exactly one score, and no score is shared by two tuples), meaning that ties are not allowed. In many cases, it is non-trivial to extend the algorithms in the previous work to handle non-injective scoring functions; in fact, the result is undefined when there are ties in tuple scores. In this work, we remove that restriction and allow non-injective scoring functions.

4.1.2 Score Distribution and c-Typical-Topk

As discussed earlier in Chapter 1, the scores of the k-tuple vector returned by U-Topk can be rather atypical, severely restricting the usefulness of the U-Topk result. We therefore propose to compute and provide the distribution of the total scores of top-k tuples. There are two possible usages of such a distribution:

(1) An application can access the distribution at any granularity of precision (e.g., histograms of any bucket width).

(2) An application can receive c typical top-k vectors (n.b., c-Typical-Topk, defined below), where c is a parameter specified by queries.

Intuitively, c-Typical-Topk returns c top-k vectors (for $c \geq 1$) such that the actual top-k result (drawn according to its distribution) is close to at least one of the c vectors. When $c = 1$, the result has a score that is the expected score of top-k vectors; on the other hand, a big c value gives c vectors (and their probabilities) that approach the distribution of all top-k vectors. Put another way, the ith vector has a score that is approximately $i/(c+1)$ through the probability distribution of all possible scores.

Definition 4.1 (c-Typical-Topk scores). *Let the distribution of the total scores of top-k tuples of an uncertain table T be a PMF (Probability Mass Function) D. We call the set of c scores $\{s1, s2, \ldots, sc\}$, where si ($1 \leq i \leq c$) has non-zero probability in D, the* **c-Typical-Topk scores** *if for a score $S \sim D$ (i.e., randomly chosen according to D),*

$$\{s_1, s_2, ..., s_c\} = \arg\min_{\{s_1,...,s_c\}} E[\min_{s_i \in \{s_1,...,s_c\}} |S - s_i|]$$ □

That is to say, over all choices of the c scores, for a random score S chosen according to D, $|S - si|$ is minimal in expectation, where si is the closest score to S among the c scores.

Definition 4.2 (c-Typical-Topk tuples). *We call the set of k-tuple vectors $\{v_1, v_2, \ldots, v_c\}$, where v_i ($1 \leq i \leq c$) is a vector of top-k tuples of T in some possible world, the* **c-Typical-Topk tuples** *if*

$$v_i = \arg\max_{s(v_i)=s_i} \Pr(v_i), 1 \leq i \leq c$$

where $s_1, s_2, \ldots, s_c$ are c-Typical-Topk scores, $s(v_i)$ is the total scores of the tuples in v_i, and $Pr(v_i)$ is the probability that v_i is a top-k tuple vector of T. □

In other words, v_i is the most probable top-k tuple vector that has a total score s_i (if there is more than one such vector, v_i can be any one of them).

For example, we can find that the 3-Typical-Top-2 scores of the table in our earlier example is {118, 183, 235}, with an expected distance 6.6 for a random top-2 vector. The 3-Typical-Top-2 vectors are {(T2, T6), (T7, T6), (T7, T3)}. For comparison, the 1-Typical-Top-2 vector is (T3, T2), which has a slightly smaller probability (0.16) than that of the U-Top-2 vector (T2, T6) with probability (0.2), but has a much more typical score of 170, as opposed to 118 of the U-Top-2.

4.1.3 Non-injective Scoring Function and Ties

Now we consider the case in which the scoring function s is non-injective and there can be ties among the scores of the tuples of an uncertain table. Score ties are common when the score is based on an attribute that does not have many distinct values, e.g., year of publication, number of citations, or even non-numeric attributes [FK04]. It is also called partial ranking in [FK04], where the authors studied combining several ranked lists to produce a single ranking. We call the set of all tuples that have the same score a tie group. When a tuple does not have the same score with any other tuple, it is in a tie group of size one. A tie group in an uncertain table T contains all uncertain tuples that have the same score; a tie group in a possible world contains all tuples that appear in that world and have the same score.

We first discuss what this implies in a single possible world (i.e., without uncertainty). In a possible world w, as usual, a top-k tuple vector still contains a set of k tuples that have the highest scores. When there are ties, it is likely that there are multiple such top-k vectors in w, all ending in some tuples from a tie group. We say that a top-k vector v contains a tie group g if all tuples in g belongs to v. We say that a top-k vector v partially reaches a tie group g if at least one but not all tuples in g belong to v. We say that g contributes m tuples to v if exactly m tuples from g belong to v. We state the following theorem without proof.

Theorem 4.1 *In a possible world w, all top-k vectors must contain the same set of tie groups. If there is more than one top-k vector, they must all partially reach the same tie group g and g contributes the same number of tuples m to all those vectors. In fact, there are such vectors, where |g| is the number of tuples in g.* □

Example 4.1 *We can order the tie groups according to their scores in descending order. Let us say that g1 = {T2, T6}, g2 = {T3, T7, T10}, and g3 = {T5, T9, T12} are the three tie groups in a possible world with the highest*

scores. Among the three groups, g1 has the highest score and g3 has the lowest. Suppose we want to ask for the top-7 tuples. Then there are top-7 tuple vectors {g1, g2, T5, T9}, {g1, g2, T5, T12}, and {g1, g2, T9, T12}, all containing g1 and g2 but partially reaching g3. g3 contributes 2 tuples to each vector. □

It is clear that all top-k tuple vectors of a possible world have the same total score. Thus, in terms of the score distribution, ties would not have any impact: the probability of some score is still the sum of the probabilities of all possible worlds whose top-k vectors have that score. For c-Typical-Topk, among possibly multiple vectors that have some score, we choose one of them with the highest probability to appear in the uncertain table.

4.2 Computing Score Distribution of Top-*k*

A key challenge is to compute the distribution of the total scores of top-k tuple vectors. This is inherently computationally expensive because unlike U-Topk and U-kRanks, this is not really a search problem (e.g., searching for the highest probability vector), as, in this case, one must account for all top-k vectors' scores and probabilities. The goal of such an algorithm is to output the distribution as a set of (score value, probability) pairs.

4.2.1 Two Simple Algorithms

We first present two algorithms which establish a baseline for comparison with the algorithm presented in Section 4.2.2 and 4.2.3. For now, we do not consider non-injective scoring functions and ties in tuples' scores; these will be discussed in Section 4.2.3. Figure 4.1 shows the first algorithm, called StateExpansion.

We first initialize the distribution to be an empty set (step 1). S is a set of states and we initialize it to contain one state – containing the empty tuple vector ε (step 2). We then go through all tuples in descending order by score, expanding each current state in S in two different ways: either include the new tuple or do not. When we reach k tuples at a state, we add it into the distribution to be returned (step 10). When the probability of a state gets too small (below a threshold p_τ as an input parameter), it is dropped. Note that the number of (score, probability) pairs in the output dist could potentially be very large. Thus, in step (10), we use a coalescing strategy to limit the size of the output. We omit the details here, which are described in Section 4.2.3. The StateExpansion algorithm has an exponential cost in the number of tuples considered (subject to the probability threshold).

```
Input: T: an uncertain tuple set in rank order,
       p_τ: a probability threshold – note: a top-k vector with
            probability below p_τ need not be considered.
Output: The score distribution of top-k vectors.
(1)  dist = Φ
(2)  S = {ε}
(3)  for each t from T do
(4)     if S is empty then break end if
(5)     S' = Φ
(6)     for each state s in S do
(7)        Append t to s and get a new state s1.
(8)        Compute s1's score and probability based on s.
(9)        if s1 has k tuples then
(10)          Add its score and probability to dist.
(11)       else if s1's probability is greater than p_τ then
(12)             S' = S' ∪ {s1}.
(13)          end if
(14)       end if
(15)       Append ¬t to s and get a new state s2.
(16)       Compute s2's probability.
(17)       if s2's probability is greater than p_τ then
(18)          S' = S' ∪ {s2}.
(19)       end if
(20)    end for
(21)    S = S'
(22) end for
(23) return dist
```

Fig. 4.1: Algorithm *StateExpansion*.

We next show a more efficient algorithm. In this algorithm, we first determine an upper bound on the number of uncertain tuples that we have to examine when tuples are in rank order by score. A reasonable stopping condition is that we do not need to consider tuples that have probability less than pτ being in top-k.

Theorem 4.2. *Given that we do not need to consider any tuple that has probability less than pτ being in top-k, the stopping condition of the sequential scan of tuples in rank order by score is at a tuple t satisfying*

$$\mu \geq 1 + k + \ln\frac{1}{p_\tau} + \sqrt{\ln^2\frac{1}{p_\tau} + 2k\ln\frac{1}{p_\tau}}$$

(i.e., we do not need to consider any tuple from t onwards), where and T(t) is the set of all tuples ranked higher than t, except those in t's ME group. Furthermore, such a stopping condition also guarantees that no k-tuple vector with probability pτ or more being in top-k is omitted.

Proof. We use an existing result from [HP08]. Theorem 8 of [HP08] says that a slightly different condition $\mu \geq k + \ln\frac{1}{p_\tau} + \sqrt{\ln^2\frac{1}{p_\tau} + 2k\ln\frac{1}{p_\tau}}$ ensures Pr(t is in top-k) < pτ. We note that μ may not be monotonically increasing with more tuples because we have to exclude tuple t's ME group, which can vary from tuple to tuple. However, the sum of the probabilities of t's ME group is no more than 1. Thus, adding 1 to the right hand side of the inequality ensures that once the condition is satisfied at some tuple t, it will always be satisfied for all tuples onwards. We further observe that for any top-k vector v that contains t, because v is top-k implies t is in top-k, we must have Pr(v is a top-k vector) ≤ Pr(t is in top-k) < pτ. Thus, the stopping condition also guarantees that no k-tuple vector with probability pτ or more being in top-k is omitted. □

Theorem 4.2 gives us a stopping condition, which also satisfies the requirement in the StateExpansion algorithm (i.e., no k-tuple vector with probability pτ or more being in top-k is missed). Note that we always stop at the end of a tie group because tuples in a tie group either all satisfy the stopping condition or none does. Let the number of uncertain tuples we need to consider be n. We can simply iterate through all k-combinations of the n tuples using a standard algorithm that generates all k-combinations in lexigraphical order [R95], but exclude those that violate the mutual exclusion rules. For each k-combination, we can compute its total score and probability, and eventually we get the distribution. We call this algorithm k-Combo. Its cost is $O(n^k)$.

4.2.2 The Main Algorithm

We now present our main algorithm, which is based on dynamic programming. Our presentation is done in several steps. In this subsection, we introduce the basic framework of the algorithm. In Section 4.2.3 and 4.2.4, we extend this algorithm to handle mutually exclusive tuples and score ties, respectively.

Consider the table in Figure 4.2. The rows correspond to n (determined by Theorem 4.2) uncertain tuples in rank order by score. The columns are labeled from k to 1. A cell at row Ti column j contains the score distribution of top-j tuples starting from row Ti. Thus, our goal is to get the distribution in the cell at the upper left corner of the table (marked with a "?"), i.e., the score distribution of top-k tuples starting from T1. We first consider the basic case in which tuples are independent (i.e., no mutual exclusion rules) and there are no ties in score.

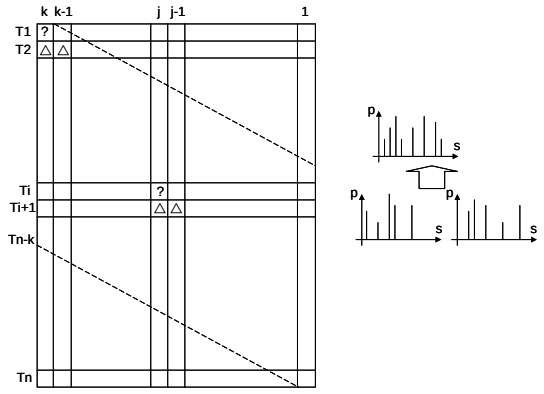


Fig. 4.2: Illustrating the basic dynamic programming algorithm, explained in the text below.

Our goal, the distribution of top-k starting from T1 (upper left corner cell), can be composed using the distributions of two cells below it (marked with triangles in Figure 4.2): the distribution of top-k starting from T2 (when T1 does not exist) and the distribution of top-(k-1) starting from T2 (when T1 exists). In general, the distribution Di,j at row Ti and column j (top-j starting from Ti) is composed from the distribution Di+1,j at row Ti+1 and column j (top-j starting from Ti+1) and the distribution Di+1,j-1 at row Ti+1 and column j-1 (top-(j-1) starting from Ti+1) in the following way:

(1) For each value and probability pair (v, p) in Di+1,j , we transform it to (v, p(1－pi)), where pi is the probability that Ti exists.

(2) For each value and probability pair (v, p) in Di+1,j-1 , we transform it to (v+si , p·pi), where si is Ti's score and pi is the probability that Ti exists.

(3) Merge the value and probability pairs resulting from (1) and (2) by taking their union except for the following: if two pairs have the same value, they become one pair with that value and with the new probability being the sum of the two original ones.

The right hand side of Figure 4.2 shows pictorially the merging process. Since all top-k tuples (there are k of them) must be among the n tuples T1 to Tn, we only need to fill in the distributions in the table of Figure 4.2 between the two dotted lines. For example, we do not need to get the distribution of top-(k-1) starting from T1; nor do we need top-2 starting from Tn , etc.

The recursive process described above fills in the table in a bottom-up manner. For the boundary conditions of the recursion, we add an auxiliary column 0 at the right border of the table. The

distribution at a cell of column 0 has only one (value, probability) pair: (0, 1), i.e., score 0 with probability 1. For the a boundary cell (at row Tn-i+1 and column i, for i = 1, …, k) immediately above the bottom dotted line, its distribution also has only one (value, probability) pair:

In the algorithm we also keep track of one tuple vector for each (v, p) pair, which is needed for obtaining c-Typical-Topk. The vector is one (among possibly many) that has score v and has the highest probability of being the top vector. The recorded tuple vector is initially empty at column 0 and contains only Tn for the cell at row Tn and column 1. Thereafter, step (1) of the distribution merging process does not change the tuple vector while step (2) prepends Ti to the vector. In step (3), when two pairs have the same value and get combined, we keep the vector that has the higher probability.

4.2.2.1 The Need for Approximation

Thus far, it appears that the cost of this algorithm is O(kn). However, there is one potential problem. For a cell at row Ti and column j (i.e., the distribution of the total scores of top-j starting from row Ti), there are possible combinations that make up the top-j scores ($1 \leq i \leq n$, $1 \leq j \leq k$). In the worst case, each combination has a distinct total score, resulting in a distribution that has the same number of discrete values (vertical lines in the PMF) in the cell. Thus, the number of vertical lines of a distribution is upper bounded by , which is O(nk). Recall that the distribution merging process described above goes through each vertical line (v, p), increasing the worst case complexity of the main algorithm to O(nk). Note that in most applications, in reality, scores are not too far apart, and total scores of different combinations are often very close or even the same. Even if they were all distinct, it would often be unnecessary to keep all O(nk) lines in the PMF. It is more desirable to have a slight sacrifice in the accuracy of the distribution in exchange for a gain in efficiency. Imagine that the range of total scores of top-k is [smin, smax]. The range can be easily determined: smax is the total score of T1 to Tk and smin is the total score of Tn-k+1 to Tn since they are sorted. Note that the span smax – smin is relatively insensitive to the problem size n. We divide the span into a constant number c' of same-size intervals (e.g., c' = 200). Each interval size is δ = (smax – smin) / c'. Suppose for the application we can coalesce vertical lines that are no more than δ away from each other in the distribution (i.e., differ by no more than δ in total scores). Then the cost to describe the output distribution is a constant.

We call the distribution at row T1 and column k (i.e., upper left corner) the final distribution and those at other cells intermediate distributions. We can have a "line coalescing" strategy as follows. At

any intermediate or final distribution, whenever the algorithm results in more than c' vertical lines, (1) pick two lines that are closest to each other and coalesce them into one: the score value is their average and the probability is their sum; (2) repeat the first step until we have c' vertical lines. As for the recorded top vector, when we coalesce two lines, we keep the tuple vector that has the higher probability.

We first observe that in the bottom-up process of computing the dynamic programming table of Figure 4.2, two lines (v1, p1) and (v2, p2) in an intermediate distribution are always going to change in a synchronized way: either they both stay at the same scores (step 1 of the distribution merging process) or the two lines get "shifted" with the same offset by adding the same score (step 2 of the merging process). In both cases their probabilities are scaled by the same factor. Thus, coalescing two lines in an intermediate distribution effectively is equivalent to coalescing them in the final distribution since they would have the same distance in scores, had we not coalesced them in any of the intermediate distributions.

Secondly, it is not hard to see that the span of any intermediate distribution is no more than that of the final distribution (smax – smin). This is because intermediate distributions either only consider top-j ($j < k$) or they use a subset of the n tuples. Thus, if an intermediate distribution has more than c' lines, by picking the two lines with minimum distance, we must be coalescing two lines that are no more than δ apart.

Now given that we have a constant cost of distribution merging, our basic algorithm so far has O(kn) time complexity. In the next two subsections, we extend our basic algorithm to more complex and realistic scenarios in which there are mutual exclusion rules and possible score ties among tuples.

Note that we do this line coalescing similarly for the StateExpansion and k-Combo algorithms in Section 4.2.1 as well. For example, in step (10) of StateExpansion, we make sure dist has no more than a constant number of score/probability pairs. This, however, does not change the complexity of those two algorithms.

4.2.3 Handling Mutually Exclusive Rules

The problem gets more complicated when there is correlation among the tuples. We now describe how to handle mutually exclusive tuples. The original algorithm would not work in the presence of

mutually exclusive tuples because the final distribution would be wrong if more than one tuple in an ME group simultaneously contributes to a top-k score.

4.2.3.1 Two False Starts

In the bottom-up dynamic programming algorithm, one might first be tempted to do the bookkeeping of which ME groups have contributed a tuple to a score (and with what probability). In this case, we do not add additional tuples from those ME groups into the intermediate distributions. Unfortunately, this is combinatorial and is too costly.

Another approach compresses all tuples in a mutually exclusive set into one tuple. We use the terminology in [HP08] and call it a rule tuple. A rule tuple has a composite score and a probability of the sum of the original tuples. At a row of a rule tuple, step (1) of the distribution merging process stays the same and step (2) changes to adding each score/probability of the original tuples of the rule separately. For example, if a rule tuple has three original tuples, we do step (2) three times. However, the problem with this approach is that we have nowhere to place the rule tuple in the dynamic programming table since it has a composite score. Wherever we place it, we are unable to compute the probability of a top-k score correctly because we have lost the information of exactly which original tuples appear (or do not appear) in a strict score order.

4.2.3.2 A Good Start

Although the second strategy above fails, it provides the following inspiration: suppose we require that the last tuple (i.e., the k-th) of the top-k has to be Tn, then the tuples in the dynamic programming table can be in any arbitrary order (i.e., they do not have to be ordered by scores as stated earlier). This is because for any tuple i with a score higher than the last tuple of the top-k, if i is in the top-k, we simply multiply the current probability by its probability pi ; if i is not in top-k, we multiply by (1 − pi). The earlier order requirement simply prevents us from multiplying the (1 − pi) for any tuple i with a score smaller than the last one in top-k. But if the last one in top-k is Tn , we know for sure all other tuples have a higher score. Now without the order constraint, we can then modify the original tuples in the following way:

(1) Remove all other tuples (if any) that are in the same ME group as Tn from the table.

(2) Compress all other ME groups into rule tuples and leave them in any order. Remember the constituent original tuples' scores and probabilities for a rule tuple. A rule tuple also has a probability that is the sum of those of the constituent tuples.

The next trick ensures that the dynamic programming algorithm only considers the top-k vectors that end with Tn. Recall that we added an auxiliary column 0 at the right border of the dynamic programming table of Figure 4.2. Each cell in column 0 holds a distribution (0, 1) – score 0 with probability 1. We call a cell in column 0 an exit point because it indicates that we do not need to select any more tuples as top-k from that tuple and below. In order to only incorporate top-k vectors that end with Tn, all we need to do is simply "block" those exit points by letting them have a distribution of (0, 0) instead – score 0 with probability 0. It can be easily verified that such a distribution cannot be propagated by the distribution merging process. With that change, the dynamic programming algorithm can proceed as before.

The change on the distribution merging process to the main algorithm is the same as that described in the second attempt in Section 4.2.3.1.

What we have achieved so far is only the distribution of total scores of top-k vectors that end with Tn. To get the distribution for all top-k vectors, an easy extension is simply to repeat this for each tuple from Tk to Tn (i.e., truncate the dynamic programming table at each of those tuples and treat them as the last tuple of the top-k, respectively) and then we merge all the final distributions together. For a truncated table, an ME group may be truncated as well. That is, if the table is truncated at Ti ($k \leq i \leq n$), an ME group now only contains tuples in the remaining table (i.e., from T1 to Ti). The compression step now applies to the reduced ME groups.

4.2.3.3 Refinement

It turns out that we can do better than the simple extension above. We call a tuple a lead tuple if it is the first one (i.e., with the highest score) in an ME group. If an ME group has only one tuple (i.e., not mutually exclusive with any other tuple), that tuple is a lead tuple. In a score-sorted sequence T1 to Tn, a maximal contiguous subsequence of lead tuples Ti, Ti+1, …, Tj is called a lead tuple region. For a subsequence to be maximal, it must be satisfied that (1) either $i = 1$ or Ti-1 is not a lead tuple; and (2) either $j = n$ or Tj+1 is not a lead tuple.

We can see that we do not need to do the dynamic programming procedure for each tuple. Instead, we only need to do it once for every lead tuple region and once for every non-lead tuple. This is because when the dynamic programming table ends with a lead tuple region, tuples in it behave exactly as independent tuples and they will not interfere with any other tuples above. Thus, for a lead tuple region, we can simply do one dynamic programming to get the score distribution of top-k vectors that end with any tuple in that lead tuple region. We achieve this by setting the boundary conditions properly. For the distributions in the cells of the auxiliary column 0, we set it to be (0, 1) at the rows of a lead tuple region in question and set it to be (0, 0) for other rows. Recall that (0, 0) is to block an exit point and (0, 1) is to enable it. Everything else, including the rule tuple compression, stays the same. This is illustrated in Figure 4.3.

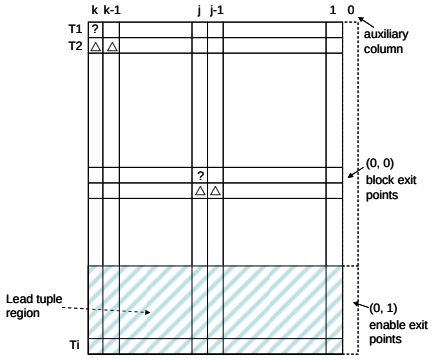


Fig. 4.3: One dynamic programming for a lead tuple region.

With this improvement we can see that the time complexity of our algorithm that handles mutually exclusive tuples is O(kmn), where m is the number of tuples (among T1 to Tn) that are mutually exclusive with other tuples. In many applications, mutually exclusive tuples are only a small proportion of the total. The computational cost is proportional to this fraction.

4.2.4 Handling Ties

In many real applications, the scoring function s is non-injective which leads to ties among the tuple scores [FK04]. We discussed the semantics of top-k vectors and score distributions at the end of Section 4.1. We now extend the dynamic programming algorithm that we have developed so far to take care of the case of score ties. We shall prove that the following simple extension of the algorithm satisfies our requirements:

Recall that before, the sort order was on scores. Now, sort tuples in descending order by (score, probability). When two tuples have the same score, they are in descending order of probability; when they have the same probability as well, break ties arbitrarily.

Aside from this adjustment, the algorithm works the same as before. The next theorem shows that this modification is correct.

Theorem 4.3. *With the above extension to the dynamic programming algorithm, we achieve our two goals: (1) we obtain the correct final score distribution of top-k and (2) among vectors that have the same score, the one that is captured at the end of the algorithm is the one with the highest probability.*

For the proof of Theorem 4.3, we first need the following definition and lemma.

Definition 4.3 (**Configuration of top-k**). *A* configuration of top-k *is a set of* $(k - g)$ *uncertain tuples plus g tuples from a tie group in non-increasing score order, with the ending tie group having the lowest score (the* $k - g$ *tuples are not in that tie group).*

Note that a configuration has a fixed total score and two configurations may have the same total score. The probability of a configuration is the probability that such a configuration is the top-k tuple vector.

Lemma 4.1. *Let A be the set of* $(k - g)$ *uncertain tuples and T be the ending tie group of a configuration. Let B be the set of tuples that have higher scores than those in T but are not in the configuration. The probability of the configuration is the probability that (1) tuples in A appear, and (2) those in B do not, and (3) at least g tuples from T appear.*

Proof (Lemma 4.1). Clearly, (1) and (2) must be true for the configuration to be top-k. Except for the case that fewer than g tuples from T appear, this configuration will be top-k. Thus, we have (3). □

Proof (Theorem 4.3). A top-k score distribution is made up of different configurations. Therefore, to prove goal (1) of Theorem 4.3, we only need to show that our algorithm computes the probability correctly for each configuration.

For the ending tie group T of a configuration, our algorithm puts the tuples in probability descending order. In fact, we can see that for any arbitrary order, as long as it is fixed, the dynamic programming will compute the probability of the configuration correctly. Let the ending tie group T have t tuples in total: T1, T2, …, Tt in some fixed order. The event (3) in Lemma 1 (i.e., at least g tuples from T appear) can be decomposed into sub-events as follows. Imagine a t-bit binary string. We choose g bits and set them to 1; the other bits are all 0. Clearly there are such strings. We use each of them to construct a sub-event: we truncate the string at the last 1 bit; then starting from the 1st bit until the last bit (which is 1), if the i'th bit is 1 (or 0), we add "Ti appears" (or "Ti does not appear", respectively) into the sub-event. It is easy to see that the dynamic programming procedure computes the probability of each such sub-event and adds them up to be the probability of the event (3) in Lemma 1. Thus, the algorithm computes the probability of the configuration correctly and we finish the proof of goal (1) of Theorem 4.3.

Example 4.2. *Consider the scenario that the first seven uncertain tuples are:*
(T1, 10, 0.5),
(T2, 8, 0.3), (T3, 8, 0.2), (T4, 8, 0.1),
(T5, 7, 0.5), (T6, 7, 0.4), (T7, 7, 0.2).
That is, T1 has score 10 and probability 0.5, and so on. Consider a top-5 configuration c that includes T1, T2, T4, and two tuples from the tie group g = {T5, T6, T7}. Then
$Pr(c) = Pr(T1)Pr(T2)(1-Pr(T3))Pr(T4)Pr(\geq 2 \text{ tuples in } g \text{ appear})$
We can compute that $Pr(\geq 2 \text{ tuples in } g \text{ appear}) = 0.5 \cdot 0.4 \cdot 0.2 + 0.5 \cdot 0.4 \cdot (1-0.2) + 0.5 \cdot (1-0.4) \cdot 0.2 + (1-0.5) \cdot 0.4 \cdot 0.2 = 0.3$. *On the other hand, our dynamic programming algorithm will calculate the probability of this part of c to be:* $0.5 \cdot 0.4 + 0.5 \cdot (1-0.4) \cdot 0.2 + (1-0.5) \cdot 0.4 \cdot 0.2 = 0.3$ *as well. Thus, our algorithm computes the probability of the configuration c correctly.*

We next show that our algorithm achieves goal (2), i.e., the vector recorded is the one with the highest probability. Note that the algorithm may not compute the probability correctly for all vectors in a top-k configuration, but it does compute it correctly for the one with the highest probability, due to the fact that we order the probability in non-increasing order in the ending tie group. In Example 4.2, our algorithm computes the probability of the vector that ends with T5 and T6 correctly: $0.5 \cdot 0.4 = 0.2$ (for the part in tie group g). On the other hand, for the vector ending with T5 and T7, the algorithm computes $0.5 \cdot (1-0.4) \cdot 0.2 = 0.06$, but the actual probability should be $0.5 \cdot 0.2 = 0.1$. This is fine because we only need to return the vector that has the maximum probability.

Note that the extension of our algorithm to handle mutually exclusive tuples as discussed in Section 4.2.3 would not affect the results of our proof above. This is because for a given configuration of top-k, after removing tuples in set T that are mutually exclusive with any tuple in set A (sets T and A as defined in Lemma 1), our proof holds in the same way. This concludes the proof of Theorem 4.3. □

It is not hard to see that the same method can be applied to the algorithm StateExpansion in Section 4.2.1 as well to handle score ties: we just need to sort the tuples in (score, probability) descending order.

4.3 Computing *c*-Typical-Top-*k*

Given a distribution of the total scores of top-k vectors as computed in Section 4.2, we now study how to compute c-Typical-Topk vectors. We first formalize the problem. Let the score distribution be {(s1, p1), (s2, p2), …, (sn, pn)} and each score si ($1 \leq i \leq n$) is associated with a top-k tuple vector vi . The vector vi is the one with the highest probability of being top-k, among those having the same total score. Our goal is to choose from the n vectors and output c of them such that their scores satisfy the optimality requirement in Definition 4.1. We call si a typical score if its vector is chosen by the algorithm.

Using ideas similar to [HT91], we can derive an efficient O(cn) time dynamic programming algorithm to solve this combinatorial optimization problem. We use a two function recursive approach. Let $F^a(j)$ be the optimal objective value of the subproblem reduced to the set {sj, …, sn}, for j = 1, …, n, where a is the maximum number of typical scores and let $G^a(j)$ be the respective value for the same subproblem, provided that sj is a typical score. We have, for j = 1, …, n, and $a \leq c$,

$$F^a(j) = \min_{j \leq k \leq n} \left[\sum_{b=j}^{k} p_b (s_k - s_b) + G^a(k) \right] \qquad (1)$$

$$G^a(j) = \min_{j < k \leq n+1} \left[\sum_{b=j}^{k-1} p_b (s_b - s_j) + F^{a-1}(k) \right] \qquad (2)$$

In equation (1), k iterates over the possible first typical score's positions, and in (2), k is the first position that is closest to the second typical score (i.e., sj to sk-1 are closest to the first typical score, sj). The solution for our original problem is thus given by $F^c(1)$. The boundary conditions are

$$G^1(j) = \sum_{b=j}^{n} p_b (s_b - s_j),\ j = 1, \ldots, n, \quad F^a(n+1) = 0,\ a \geq 1 \qquad (3)$$

We define, for j = 1, …, n,

$$P(j) = \sum_{b=1}^{j} p_b, \quad PS(j) = \sum_{b=1}^{j} p_b s_b \qquad (4)$$

Then we can rewrite (1) and (2) as

$$F^a(j) = \min_{j \le k \le n} \left[(P(k) - P(j-1)) s_k - PS(k) + PS(j-1) + G^a(k) \right] \quad (5)$$

$$G^a(j) = \min_{j < k \le n+1} \left[PS(k-1) - PS(j-1) - (P(k-1) - P(j-1)) s_j + F^{a-1}(k) \right] \quad (6)$$

With some preprocessing of (4) that takes O(n) time, we can first get all P(j) and PS(j) values. Then the dynamic programming algorithm based on (5) and (6) will just take O(cn) time.

We show the algorithm in Figure 4.4. We first pre-compute all the P(j) and PS(j) values (lines 1 to 5). Lines 6 to 13 set the boundary conditions according to Equation (3). Lines 14 to 35 iteratively apply Equation (5) and (6) in turn to fill in the two dynamic programming tables (i.e., all F and G values). Note that f and g values (lines 22 and 32) keep track of the k values that minimize the r.h.s. of Equations (5) and (6). This is needed to trace back and output the c typical top-k tuple vectors (lines 36 to 41).

Input: A top-k score distribution (s_i, p_i, v_i), $1 \le i \le n$, where s_i is a score, p_i is its probability, and v_i is a top-k tuple vector that has score s_i and has the highest probability; an integer c

Output: c tuple vectors that are c-Typical-Top$k$.

(1) $P[0] = PS[0] = 0$
(2) **for** $j = 1$ to n **do**
(3) $P[j] = P[j\text{-}1] + p_j$
(4) $PS[j] = PS[j\text{-}1] + p_j * s_j$
(5) **endfor**
(6) **for** $j = 1$ to n **do**
(7) $G[1][j] = 0$
(8) **for** $b = j$ to n **do**
(9) $G[1][j] = G[1][j] + p_b * (s_b - s_j)$
(10) **endfor endfor**
(11) **for** $a = 1$ to c **do**
(12) $F[a][n\text{+}1] = 0$
(13) **endfor**
(14) $a = 1$
(15) **for** $j = 1$ to n **do**
(16) $F[a][j] =$ MAX_DOUBLE
(17) $f[a][j] = 0$
(18) **for** $k = j$ to n **do**
(19) $tmp = (P[k] - P[j\text{-}1]) * s_k - PS[k] + PS[j\text{-}1] + G[a][k]$
(20) **if** $tmp < F[a][j]$ **then**
(21) $F[a][j] = tmp$
(22) $f[a][j] = k$
(23) **endfor endfor endif**
(24) **for** $a = 2$ to c **do**
(25) **for** $j = 1$ to n **do**
(26) $G[a][j] =$ MAX_DOUBLE
(27) $g[a][j] = 0$
(28) **for** $k = j\text{+}1$ to $n\text{+}1$ **do**
(29) $tmp = PS[k\text{-}1] - PS[j\text{-}1] - (P[k\text{-}1] - P[j\text{-}1]) * s_j + F[a\text{-}1][k]$
(30) **if** $tmp < G[a][j]$ **then**
(31) $G[a][j] = tmp$
(32) $g[a][j] = k$
(33) **endfor endfor endif**
(34) Do the for loop between line (15) and (23)
(35) **endfor**
(36) $k = 1$
(37) **for** $a = c$ down to 1 **do**
(38) $i = f[a][k]$
(39) **output** v_i
(40) $k = g[a][i]$
(41) **endfor**

Fig. 4.4: The algorithm to select c-Typical-Topk.

4.4 Empirical Study

In this section, we conducted a systematic empirical study addressing the following questions:

• What does the score distribution of top-k tuple vectors look like for real-world data? Furthermore, where does the U-Topk vector stand in the distribution, and where do c-typical vectors stand in the distribution?

• What is the performance of our main algorithm that computes the score distribution? How does it compare with StateExpansion and k-Combo? What are the scan depth (i.e., the number of tuples n that need to be read by our algorithms) values for various k values as determined by Theorem 4.2? How does the proportion of mutually exclusive tuples affect performance? By trading off accuracy for performance, how does the line coalescing strategy presented in Section 4.2.2 improve performance?

• What is the impact on score distribution and typicality of U-Topk as we alter the following system parameters: (1) the correlation between scores and confidence, (2) the score range (variance), (3) the score range within ME groups and the size of ME groups?

4.4.1 Setup and Datasets

We performed the study using the following two datasets:

• A real-world dataset collected by the CarTel project team [HB06]. It consists of measurement of actual traffic delays on roads in the greater Boston area performed by the CarTel vehicular testbed [LB08], a set of 28 taxis equipped with various sensors and a wireless network.

• A synthetic dataset generated using the R-statistical package [RPRJ]. With the synthetic dataset we can control the various parameters of the data and study their impact on results.

We implemented all the algorithms presented in this paper and the U-Topk algorithm presented in [SI07] to study the results. All the experiments were conducted on a 1.6GHz AMD Turion 64 machine with 1GB physical memory and a TOSHIBA MK8040GSX disk.

4.4.2 Results on the Real-world Dataset

In the first experiment, we examine the score distribution of top-k tuple vectors as computed by the main algorithm presented in the paper using the CarTel data. We execute the following query over some random areas taken from the whole dataset:

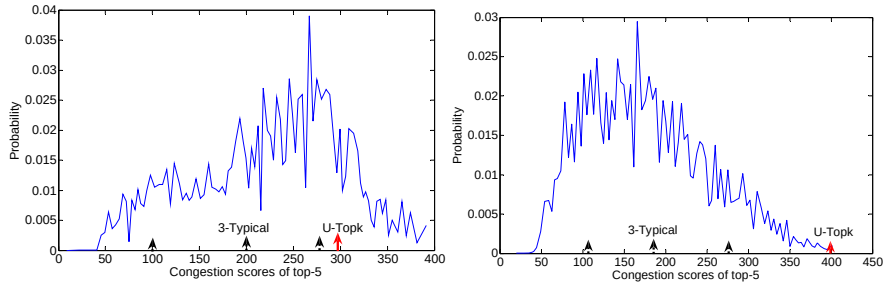
```
SELECT segment_id,
        speed_limit / (length / delay) AS congestion_score
FROM area
```

ORDER BY congestion_score DESC

LIMIT k

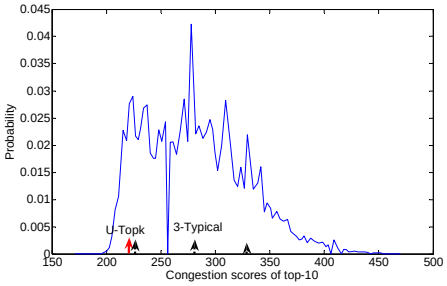
Each tuple of the relation area is a measurement record of the actual travel delay of a road segment. In this query, we define where the denominator is the actual travel speed and the numerator is the speed limit of the road segment. Thus, the congestion score is an indication of the travel speed degradation at a road segment (up to a constant factor: in the dataset, the speed_limit is in km/hour while the length is in meters and delay is in seconds). A higher congestion score implies a more congested road segment. The query selects the top-k most congested road segments in an area (say, a city). City planners might want to first locate the k most congested roads and their total (or equivalently, average) scores to give them an idea of how serious the situation is. For example, when the total scores exceed some threshold, the city planners will spend some funding to fix the traffic problem on the most congested road segments (e.g., by adjusting traffic light cycles, adding parallel roads or widening existing ones). Each road segment contains one or more measurement record. In general, each record is considered uncertain and the delay of a road segment is probabilistic [LB08]. If a road segment contains multiple measurements, we bin the samples and collect the statistics of the frequencies of the bins and obtain a discrete distribution, in which each bin is assigned a value that is the average of the samples within the bin. Bins in a distribution are mutually exclusive so that at most one of them may be selected in a possible world. Thus, a top-k tuple vector always contains distinct road segments.

Figure 4.5 shows the distributions of the total congestion scores of top-k roads at three random areas from the dataset. We use our main algorithm presented in Section 4.2.2 to 4.2.4 to compute the score distributions and the algorithm in Section 4.3 to compute c-Typical-Topk. We also examine where the resulting vector from the U-Topk algorithm [SI07] stands in the distribution. We show the U-Topk result as a solid (red) arrow and the three dotted arrows are 3-Typical-Topk results. The height of an arrow roughly indicates the probability of the corresponding k-tuple vector. We can see that in all three subplots, the score of the U-Topk result is rather atypical. In Figure 4.5 (a) and (b) it is higher than the three typical scores while in Figure 4.5 (c) it is lower. Although being the highest probability vector, the U-Topk result still has a very small probability, and it may only be slightly bigger than many other k-tuple vectors. By the definition of c-Typical-Topk, the actual top-k vector (drawn according to its distribution) is more likely to have a score that is close to one of the c typical vectors. Informed by the score distribution and typical vectors, the city planners will have a much more accurate picture of how serious the top-k most congested road segments are.



(a)

(b)



(c)

Fig. 4.5: Congestion score distribution of top-k tuple vectors in three random areas and the results of U-Topk and 3-Typicals.

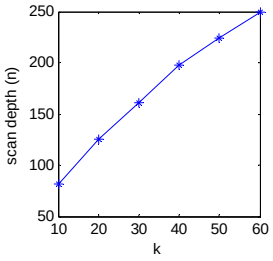


Fig. 4.6: k vs. scan depth (n).

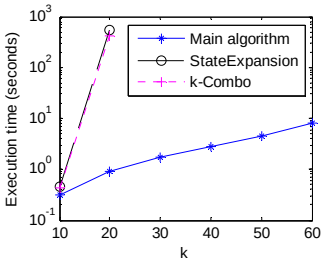


Fig. 4.7: k vs. execution time.

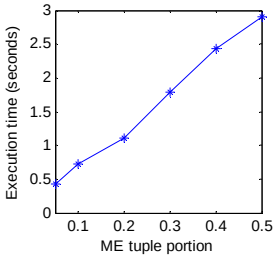


Fig. 4.8: ME portion vs. time.

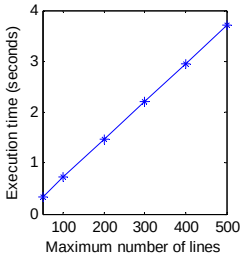


Fig. 4.9: # of lines vs. time.

4.4.3 Performance on the Real-world Dataset

In the second experiment, we examine the performance of our algorithms. We run the same query as shown in Section 4.4.2, but try different system parameters. Since the performance of both our main algorithm and k-Combo relies on the scan depth n as determined by Theorem 4.2, it is interesting to study what are the actual values of n for various k's with the real-world dataset. We set p_τ to be 0.001. Figure 4.6 shows the result that n grows roughly linearly with k as is expected from the theorem.

We next compare the performance of our main algorithm that computes the score distribution with the two simple algorithms presented in Section 4.2.1, namely StateExpansion and k-Combo. For all three algorithms, we limit the number of lines in the output distribution to be more than 100. We try

different k values in the query and compare the execution times of the three algorithms, as shown in Figure 4.7. We can see that both State-Expansion and k-Combo have an exponential growth on the running time as k increases, with k-Combo being slightly better. On the other hand, our main algorithm which uses dynamic programming techniques is significantly more efficient.

Next we examine the performance of our main algorithm as we vary the portion of mutually exclusive tuples by first selecting a subset of road segment records and run our query against it. The result is shown in Figure 4.8. As expected, the computation cost increases as we increase the portion of tuples that are mutually exclusive with other tuples, as discussed earlier.

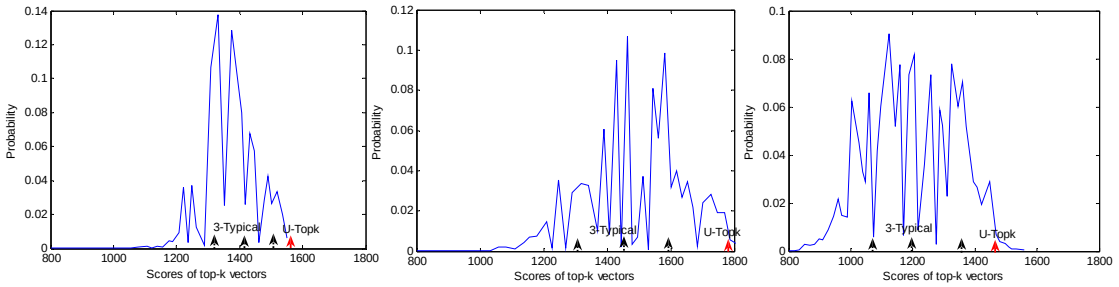
Finally, recall that in Section 4.2.2 we devised a line coalescing strategy in order to trade off accuracy for performance. The parameter here is the maximum number of lines allowed in the distributions. We vary this parameter from 50 up to 500 and the result is shown in Figure 4.9. We can see that the runtime varies linearly as the number of lines grows. The reason is that as the dynamic programming algorithm progresses bottom-up, very soon line coalescing takes effect, and the amount of computation thereafter is proportional to the number of lines in the distributions.

4.4.4 Results on the Synthetic Dataset

In this section, we use synthetic datasets because they give us control over various characteristics of the data. We further examine the impact of different kinds of data on score distribution and on how typical U-Topk results are. We first study different correlations between score and probability of tuples. We generate scores and probabilities as bivariate normal distributions with different correlation coefficients for the cases of independence ($\varrho = 0$), positive correlation (we use $\varrho = 0.8$), and negative correlation (we use $\varrho = -0.8$). We show the top-10 results for these three cases in Figure 13 (a), (b), and (c) respectively. We can see that compared to the independence case (Figure a), a positive correlation between scores and probabilities shifts the score distribution of top-k vectors to the right (Figure b) while a negative correlation shifts it to the left (Figure c). This is because if leading tuples (with higher scores) are more likely to exist, they are also more likely to be in top-k, thus making the total scores of top-k tuples higher. Moreover, we also observe here that in all three cases, the U-Topk result is atypical.

We next study how the results change when we alter the range (i.e., variance) of scores in the table. In the previous experiment in Figure 4.10, we use a bivariate normal distribution with the standard deviation of the scores being 60. With other parameters being the same as in Figure 4.10a (i.e., $\varrho = 0$),

we only increase the standard deviation of the scores σ to be 100. The result is shown in Figure 4.11. It is clear that the distribution of the total scores of top-k vectors now covers a wider range, with the span of the significant portion of the distribution increased from around 350 (Figure 4.10a) to around 1000, making the distance between U-Topk score and typical scores farther apart.



(a) (b) (c)

Figure 4.10. Score distribution of top-10, U-Topk, and 3-Typical for different score & probability correlations: ϱ=0 (a), 0.8 (b), -0.8(c).

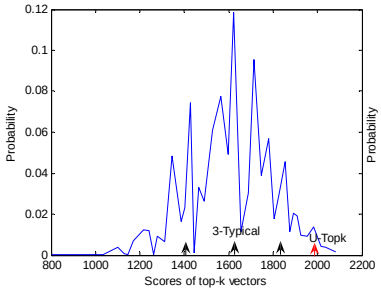


Figure 4.11. ϱ=0, but increase σ to 100.

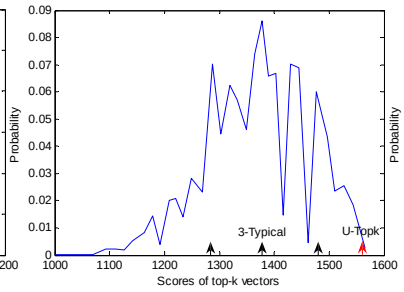


Figure 4.12. Increasing gaps between ME tuples.

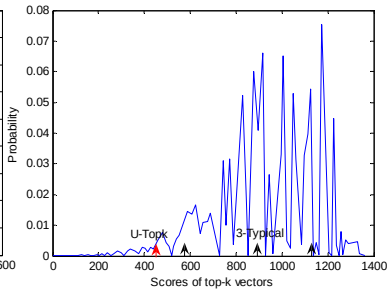


Figure 4.13. Increasing sizes of ME groups

Finally, we examine the impact on the results as we vary the mutual exclusion (ME) group settings. With everything else being the same as in Figure 4.10a (ϱ=0, σ=60), we only change the score gaps between two ME tuples. Without changing any scores in the table, we only change the assignment of the tuples to ME groups: we change the distance between two neighboring tuples in an ME group from d1 tuples to d2 tuples, where d1 is a random number from 1 to 8 and d2 is a random number from 1 to 40. The result is shown in Figure 4.12. We observe that there is no noticeable change from Figure 4.10a. However, when we increase the size of ME groups from s1 to s2 where s1 is a random number of either 2 or 3, and s2 is a random number from 2 to 10, there are some obvious changes in the results, as shown in Figure 4.13. First of all, we observe that the score distribution of top-k vectors covers a much wider range but with smaller values. The bulk of the distribution is at [200, 1350] compared to the original range of [1150, 1550] (Figure 4.10a), almost three times in width. The

reason is that because we can only take at most one tuple from each ME group to include in top-k, a larger ME group implies that we end up scanning more tuples and lower scored tuples have a higher chance to be in top-k, which in effect increases the variance of the scores of tuples that contribute to the distribution. Secondly we observe that because each ME group now contains a lot more tuples with small probabilities (they must add up to no more than 1), we essentially have an exponential growth in possible top-k vectors, all have small probabilities. This makes U-Topk (which seeks the highest probability) more unstable or atypical. Figure 4.13 shows that in this case the U-Topk result shifts to the lower end of the score distribution.

Chapter 5

PREDICTIVE QUERIES: QUERYING UNCERTAIN DATA IN THE FUTURE

In this chapter, we describe a skip-list approach to organize and possibly pre-build models for answering predictive queries that ask for uncertain data in the future.

5.1 Elements of Our Approach

5.1.1 I/O conscious skip-lists

We adopt the skip list data structure in our context, and make it I/O conscious. As stated earlier, time series databases can be too large to fit in memory. For example, 20 years of per second stock quotes have about 630M data points and reach gigabytes. Thus, for scalability, we need to consider the efficiency of query processing when storing a skip list on disk.

The original skip list structure requires a large number of pointers, which is detrimental for I/O performance. In model building for prediction queries, we use a contiguous sequence of data at some level of the skip list. A search operation, as described, also accesses a contiguous sequence of data at each level. Thus, we replicate key values at each level and store them compactly and contiguously in disk pages, instead of using pointers (one for each level) on only one copy of keys as in the original skip list. Time series data associated with the keys are stored together on pages. For example, in our stock example, time is the key and the (time, stock price) pair is stored in the skip list. Clearly, for the search to proceed, we need to store, for each key value, a pointer to its copy in the level below. Figure 5.1 illustrates this.

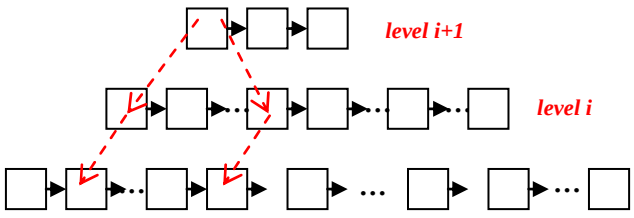


Fig. 5.1: Illustrating the I/O conscious skip list structure.

We can handle overflow and underflow of pages when there are updates using "open" and "closed" pages, in the same manner as in [AA05, AK04]. We omit the details due to space limitations. The basic idea is to maintain an invariant that requires every page to be filled within a percentage range. Time series data updates are mostly "appends" [MW98], which makes merging and splitting of pages rare. For append only data sets, we simply keep adding pages at each level, and possibly removing pages at the other end of a level of skip list when the oldest data is no longer relevant.

Note that unlike a B+ tree, whose fan-out is fixed by the database page size, the parameter p of a skip list is flexible, which we need for different sample data point densities. Furthermore, key values at each level of a skip list are chained together, unlike a B+ tree. We use these features of a skip list to efficiently retrieve samples with some needed probability from a level of the skip list to build forecasting models.

5.1.2 Prediction models

As we mentioned, searching and interpolation with a skip list are straightforward. For searching, a skip-list only helps predicates over the history based on its sort key (e.g., time=10). If the desired data points are missing, we have models for interpolation. Searching is the basic functionality provided by a skip list; interpolation occurs only at the base level of the skip list and is a well-studied problem. We refer readers to [N91] for some of this work in databases. Therefore, from now on, we only discuss prediction using the skip list approach.

As we discussed, for a given prediction interval, we pick a level of the skip list to build a model. We shall present the method of how to pick a level and how many data points in that level to use later. Given that, since we always use data up to the most recent for answering prediction queries, we use a suffix of some level in building a model. Thus, a given level of a skip list can have 0 or more associated models, each of which is built with a different suffix sequence.

5.1.3 Determining a Proper History Length

In this section, we first study the issue of how to determine a proper history length h(f) to use for a given forecast interval f. The basic idea is that we use a small number of most recent data points as the target training set, and "go back in time", starting from the earliest point in the training set, for an interval f (denoting that point in time as T-f). We then determine a proper history length h' going further back (i.e., from T-f to T-f-h') from which we can predict the target training set data well. We

determine h' using statistical tests of hypotheses. Figure 5.2 illustrates this. The algorithm is shown in the text box below.

For multiple regressions, F = si-12 / si2 has an F distribution with ni-1-ki-1-1 numerator degrees of freedom and ni-ki-1 denominator degrees of freedom [MS94]. Thus, each iteration of the loop conducts a statistical test of hypotheses with H0 being "use hi-1" and Ha being "use hi". If Ha is true, then F is big. The rejection region is F > Fα. The stopping condition (line 8) is to stop the loop at a point in the final downward slope of the F distribution. Intuitively, the algorithm iteratively increases the history length and runs statistical tests of hypotheses, until it determines that any further increase in history length "is not worth it".

An implicit assumption here, of course, is that for a given forecast interval f, if we "went back" in time for a period of f, and could use some duration of history data points relative to that time to "predict" the "present" time data points (thus the forecast interval is also f), then we can use this data to predict accurately the "real interval f into the future" (illustrated in Figure 5.2).

Input: A forecast interval f of a query.
Output: A proper history length $h(f)$ to use for answering the query.

(1) Set the most recent c_t data points as the *target training set* T, whose values we use other data points to "forecast" (to be able to compare the "forecast" values with the actual ones).

(2) Let the smallest time value in T be T_0. Let $T_f = T_0 - f$.

(3) Set $h_0 = c_0 f$. Use standard techniques [14] to build an optimal multiple regression model using data points in $[T_f - h_0, T_f]$ and compute its mean square error $s_0^2 = SSE_0/(n_0-k_0-1)$, where SSE_0 is the sum of squared error, n_0 is the number of data points used, and k_0 is the number of parameters in the model. Let $i = 0$.

(4) Do

(5) $i = i + 1$; $h_i = c_1 h_{i-1}$.

(6) Use standard techniques to build an optimal multiple regression model using data points in $[T_f - h_i, T_f]$ and compute its mean square error $s_i^2 = SSE_i / (n_i-k_i-1)$.

(7) $F = s_{i-1}^2 / s_i^2$.

(8) While $F > F_\alpha$.

(9) Output $h_{i-1} + f + |T|$, where $|T|$ is the time length of T.

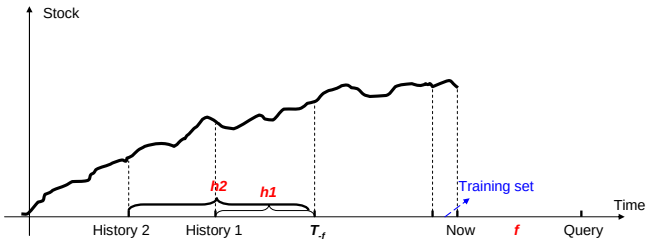


Fig. 5.2: Illustrating the determination of a proper history length.

5.1.4 Determining the Number of Data Points

Note that in the above algorithm, we use all available data points within a trial history length to build a multiple regression model. We have shown that this is often too expensive for building and maintaining models, and the excessive granularity is actually unnecessary and wasted. Thus, a natural approach is to sample and use a subset of all the available data points. Studies in the statistics and forecasting literature are concerned with the minimum number of data point requirement for forecasting (e.g., see [HK07]), which is just a lower bound and using it may still give bad prediction results.

Therefore, the basic problem is: given f and h(f) (determined by the above algorithm), how do we determine the number of random points to use within h(f)? The idea is similar to the previously presented algorithm to determine the proper history length; thus we omit the details. Roughly, we iteratively increase the number of random points used in h(f) for building a trial model, and again we use statistical testing of hypotheses to determine a good choice of the number, within a reasonable computational cost constraint.

5.2 Selection of Model Set to Build and Maintain

5.2.1 Basic Working Model

We organize the time series in question into a skip list. The skip list has a parameter p, which is the probability that an element in a lower level is also present in the next higher level. We choose a set of models to pre-build at various levels of the skip list (i.e., Pre-built Models, or PM's). Query processing picks one or more closest PM's to use, or could even build a model on the fly. The interesting aspects between PM's and skip list levels are:

- A PM uses a suffix sequence of the data points of some level.
- A level can have 0 or more PM's.

We also maintain the set of PM's we have chosen to pre-build when new data comes in or when updates happen. More specifically, a model is rebuilt whenever both θ (a threshold parameter) new data points have entered the model and the model is used by some query. Thus, it is a lazy maintenance strategy. There is a constraint on the total model rebuilding cost as described below. A model update involves using the same number of the most recent data points at the level of that model in the skip-list to rebuild the regression model. In addition, after a sufficient number of new data points enter the model, we choose the history length and the number of data points again.

5.2.2 Quantifying Model Maintenance Cost

We next quantify the maintenance cost of a set of models. We assume a set of models in a skip list that we have chosen to build and maintain. New tuples arrive at some rate.

Theorem 5.1. *We organize time series data into a skip list with parameters p, θ and the lazy maintenance strategy as described earlier. New tuples come in at a rate of r (tuples/sec), and we consider the expected incoming rate for upper levels of the skip list. Let the set of models be M. For a model* $m \in M$, *let l(m) be the skip list level at which the model is located and q(m) be the reference rate (times/sec) of the model by queries. Let* C_R *denote the canonical rebuilding cost of a model. Then with the tuple incoming rate, the maintenance cost rate of M is* $C_R \cdot \sum_{m \in M} \frac{1}{\max\left(\frac{1}{q(m)}, \frac{\theta}{r \cdot p^{l(m)}}\right)}$.

Proof: For a model *m* located at level *l(m)* of the skip list, the arrival rate of new tuples for that level is $r \cdot p^{l(m)}$. The *lazy maintenance* strategy implies that a model is rebuilt either when every θ new data points come in, or when the model is used by some query, whichever happens later. Thus, a model is rebuilt every $\max\left(\frac{1}{q(m)}, \frac{\theta}{r \cdot p^{l(m)}}\right)$ seconds. Then it is clear that the overall maintenance cost rate is $C_R \cdot \sum_{m \in M} \frac{1}{\max\left(\frac{1}{q(m)}, \frac{\theta}{r \cdot p^{l(m)}}\right)}$. □

Note that the optimal history length and the number of data points to use for a given prediction interval length may change as time progresses. We consider this as part of the model rebuilding (i.e., an ingredient of CR in Theorem 5.1). A system can choose these parameters again after a certain number of new data points enter the model.

5.2.3 Choosing a Set of Models to Pre-build

We are only concerned with the set of forecast intervals of a query workload. Thus, we model the query workload as a discrete PMF w on forecast intervals $f_i = \frac{iF}{n} (1 \le i \le n)$, with their associated probabilities $p_i \ (1 \le i \le n)$, respectively, where F is the maximal forecast interval.

The optimization problem is that given a query workload, subject to a constraint on maximal maintenance cost, we want to find a set of intervals for which we build models so that the *expected model distance* for a random query in the workload is minimized. Note that different models use different levels of the skip list and can have different maintenance cost (Theorem 5.1). This problem is similar in spirit to the *knapsack problem* (but with the extra complication that the *value* of an item is correlated with what items are being selected). Thus, an efficient optimal algorithm is unknown. Because randomized algorithms are known for their simplicity and efficiency [MU05], we devise such an algorithm, to provide a practical solution and to make theoretical analysis easier. In fact, because of its efficiency, one can repeat the algorithm several times to choose the result with the smallest expected model distance. Here is the algorithm.

Input: a query workload w as a discrete PMF; a constraint on maximal model maintenance cost rate C_M.
Output: A set of forecast intervals for which we build models.

(1) Let $M = \Phi$.
(2) Repeat
(3) Obtain a random sample of forecast interval f from query workload PMF w, using a standard method to sample from a discrete distribution.
(4) $M = M \cup \{f\}$.
(5) From f, determine the proper history length h and the number of data points n to use within the history length using algorithms in Section 4. From h and n, we get the density of the data points. Thus, a model will be built using the skip list level that has the closest density.
(6) Incrementally compute the maintenance cost rate C of the set M using Theorem 1.
(7) Until $C > C_M$ or M contains all intervals.
(8) If $(C > C_M)$ then $M = M - \{f\}$.
(9) Output M.

The algorithm repeatedly samples a new forecast interval f from the workload PMF w using established weighted sampling methods from a discrete PMF. It continues this process until the maintenance cost rate of the models exceeds the constraint.

Analogous to the database design problem for materialized views, this kind of pre-built structure often requires knowledge of the statistics of future requests. The statistics are collected through profiling at the database server, etc. Although PM's can be robust against certain changes of the workload, a rebuild is unavoidable when dramatic changes occur. As an input of ChoosePMSet, distribution w can reflect how much knowledge of the workload is assumed. Less knowledge implies a "flatter" distribution while more knowledge renders a more specific distribution.

5.2.4 Analysis of *ChoosePMSet* algorithm

We next analyze "how well" the workload PMF *w* is satisfied after running the algorithm *ChoosePMSet* to produce a set of models to build and maintain within the cost budget. To be precise, we need the following definition.

Definition 5.1. *Let the output M of ChoosePMSet have m forecast interval points out of a total of n points* $f_i = \frac{iF}{n} (1 \le i \le n)$ *where i is called the* index *of a point. Then for an arbitrary query point* $f_i = \frac{iF}{n} (1 \le i \le n)$ *define its* model distance *as the index distance between* f_i *and the closest point in M.* □

For example, for query point f_{95}, if the closest point in *M* is f_{99}, then the model distance of f_{95} is $99 - 95 = 4$.

Theorem 5.2. *Let m and n be as described in Definition 5.1. Then the expected model distance of a query point in workload w is* $\sum_{i=1}^{n} \sum_{d=1}^{n-1} p_i (1 - \sum_{j=i-d}^{i+d} p_j)^m$

Proof. For a query point with *index i*, define random variable D_i as its *model distance*. Then the probability that none of the *m* independent samples falls in a radius *d* of the query point *i* is,

$$\Pr[D_i \ge d] = \left(1 - \sum_{j=i-d}^{i+d} p_j\right)^m \qquad (1)$$

As D_i is a discrete random variable with *non-negative* values, we have (intuitively, for d from 1 upwards, cumulatively, $\Pr[D_i \ge d]$ is the probability that we add 1 to the expectation [MU05]),

$$E(D_i) = \sum_{d=1}^{\infty} \Pr[D_i \ge d] = \sum_{d=1}^{n-1} \Pr[D_i \ge d] \qquad (2)$$

From (1) and (2), we have $E(D_i) = \sum_{d=1}^{n-1} \left(1 - \sum_{j=i-d}^{i+d} p_j\right)^m$

Define random variable *D* as a random query point (in *w*)'s *model distance*. Thus,

$$E(D) = E(\sum_{i=1}^{n} p_i D_i) = \sum_{i=1}^{n} p_i E(D_i) = \sum_{i=1}^{n} \sum_{d=1}^{n-1} p_i (1 - \sum_{j=i-d}^{i+d} p_j)^m$$

where the second equality follows from the linearity of expectation. □

As we shall show in the experiments, we can write a simple program to compute the expected model distance for a specific instance of the problem.

5.3 Query Processing

In this section, we discuss query processing techniques with a PM set. In general, for a query on future time series data, we pick the closet pre-built model to use. This is clearly straightforward for point queries. We discuss interesting query types, namely, range query, aggregations, and joins.

5.3.1 Range Queries and Aggregations

We discuss aggregation queries (in particular, SUM/AVG and MIN/MAX) with a range predicate, as that would include the treatment of both range queries and aggregations.

5.3.1.1 SUM/AVG with a Range Predicate

Let us start with an example query:

Q1: SELECT AVG(price) FROM ibm_ticks WHERE time ≥ now + 10 days AND time ≤ now + 30 days

A trivial way to evaluate such a query is to "materialize" all future data points in the range of the predicate, and then compute the aggregate in the brute-force way. However, it turns out that there are much more efficient ways. For that, we first demonstrate an axiom called the *monotonicity assumption*.

Monotonicity Assumption. *When the forecast interval f increases, we can assume that the optimal history length h(f) also increases or stays the same, and the data point density of the model used either decreases or stays the same.* □

Intuitively, the monotonicity assumption makes sense because to predict a longer interval, one wants to use a longer history length, with a sparser granularity of the data points. Since the data point density drops when the skip-list level increases, we have the following corollary.

Corollary 5.1. *For a forecast interval f, let m(f) denote the pre-built model we use to answer f, and accordingly, l(m(f)) denotes the skip-list level of the model. Then, when f increases, l(m(f)) either also increases or stays the same.* □

As the prediction interval increases, the level (in a skip list) of the model used must either go up or stay the same (in which case the number of data points used does not drop). Thus, there is a total order of all the PM's, consistent with the order of query intervals.

From the corollary, we can see that a range query is answered by *a set of contiguous models* (in terms of their skip-list levels), each answering a sub-range of the predicate. We shall verify the validity of the monotonicity assumption empirically.

Theorem 5.3. The *result of a basic SUM query with a range predicate for a future time interval* $[t_0, t_k]$ *as in Q1 can be computed as*

$$\sum_{t=t_0}^{t_1} f_1(t) + \sum_{t=t_1+1}^{t_2} f_2(t) + \ldots + \sum_{t=t_{k-1}+1}^{t_k} f_k(t)$$

where $f_i(t) = \sum_{j=0}^{d_i} (c_{ij} \cdot t^j),\ 1 \le i \le k$ *are a set of contiguous polynomial regression models in the skip list.* □

As the sum of powers of integers is a well-studied problem in mathematics [BB43], we can compute the *SUM/AVG* with time complexity *O(kd)*, where *k* is the number of models spanned by the range predicate, and *d* is the maximal degree of any of those models. Since typically both *k* and *d* are small constants, we achieve constant time complexity. This is in contrast to the naive method of materializing every future data points, which requires a linear processing cost.

Example 5.1. *Suppose a range predicate like the one in Q1 spans three models and the sum can be represented by the following:*

$$\sum_{t=10}^{15} (3t^2 - 7t + 10) + \sum_{t=16}^{22} (-0.1t^3 + 11t^2 - t + 9) + \sum_{t=23}^{30} (8t^2 - 15t + 2)$$

It is known that

$$s_1(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \qquad s_2(n) = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6},$$
$$s_3(n) = \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Thus, the sum can be rewritten as

$$3(s_2(15) - s_2(9)) - 7(s_1(15) - s_1(9)) + 60 - 0.1(s_3(22) - s_3(15))$$
$$+11(s_2(22) - s_2(15)) - (s_1(22) - s_1(15)) + 63 + 8(s_2(30) - s_2(22))$$
$$-15(s_1(30) - s_1(22)) + 16$$

and we obtain the result for sum. □

5.3.1.2 MIN/MAX with a Range Predicate

We now look at the MIN/MAX aggregations in a range of a future time interval. Consider this example query:

Q2: SELECT MAX(price), MIN(price) FROM ibm_ticks WHERE time ≥ now + 10 days AND time ≤ now + 30 days

To answer a MAX aggregation over a future time range, consider the simple case that the time range is covered by only one model. Let *f* be the polynomial function of the multi-regression model. For a continuous function, to get the maximum [S01], we want to find a time value *t*, such that

$$\frac{df}{dt} = 0 \qquad (1)$$

$$\frac{d^2 f}{dt^2} < 0 \qquad (2)$$

Most functional relationships in nature seem to be smooth (except for random errors) – that is, they are not subject to irregular reversals in direction. So the degree of the polynomial is generally low [MS94], most often 1 to 3, rarely greater than 3. In fact, a high degree often indicates over-fitting and is not a good model. Skip-lists reduce the data points and avoid over-fitting. Thus, in practice, computing roots for (1) and (2) is easy and there are not many solutions.

However, we actually have a set of *discrete* time values and a peak value we find from solving (1) and (2) may not fall in the set. In that case, we call the two closest time values in the set *discrete peaks*. For example, suppose a model spans the range *[10, 20]*, but one of the solutions from (1) and (2) is $t = 16.3$, then $t = 16$ and $t = 17$ are the "discrete peaks". We also note that the time range of the query can span multiple models. The following theorem determines the result for a MAX or MIN query.

Theorem 5.4. *Let $M = \{f_0(\cdot), f_1(\cdot),\ldots, f_{k-1}(\cdot)\}$ be the set of k contiguous regression models spanned by the range predicate of a MAX query. Let the range of the predicate be $[t_0+1, t_k]$ and each model $f_i(\cdot)$ covers the sub-range of $[t_i+1, t_{i+1}]$. We call t_0+1, t_1, t_1+1, t_2, $t_2+1,\ldots, t_k$ the* borders *of M. Then for the MAX query, we only need to examine the discrete peaks (if any) of each of the k models and the borders of M. A MIN query can be answered analogously by changing the inequality in (2) to ">".*

Proof. Suppose that the MAX value were not a discrete peak or a border of *M*. Let the time of the MAX value be *t* and let it be in model *f*. It must be true that both $t - 1$ and $t + 1$ are also in *f*, since *t* is not a border. Because *f* is a continuous function and *t* is not a discrete peak, it must be true that either $f(t - 1) \geq f(t)$ or $f(t + 1) \geq f(t)$. Thus, we could use either $t - 1$ or $t + 1$ as the MAX. The same argument repeats until we reach either a border or a discrete peak. □

5.3.2 Join Queries

We now look at JOIN queries with JOIN predicates on values in a future time range. Consider this query:

Q3: SELECT ibm.day, ibm.stock, sun.day, sun.stock FROM ibm, sun WHERE ibm.day BETWEEN (now, now+30days) AND sun.day BETWEEN (now, now+30days) AND ibm.stock > sun.stock

A naive way to answer a JOIN query of a future time range is to generate all future data points in the range for both relations, and then determine a JOIN strategy using a classical optimizer. However, a much more efficient way is to do a "model JOIN".

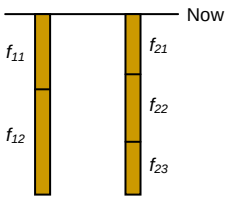


Fig. 5.3: Illustrating the "model JOIN".

As shown in Figure 5.3, for each model of one relation in the query range (f11 and f12 of ibm), we solve an inequality or equality (depending on the JOIN predicate). In this example, we solve f11(t) > v, i.e., say, 3t2 – 6t + 5 – v > 0. Likewise, we solve f12(t) > v, etc. Thus, for each value in the query range of the second relation, we use the solution of the inequalities/equalities (i.e., f11(t) > v and f12(t) > v, etc.) to get the matching tuples in the first relation. Clearly, this is just a linear cost overall, and is much more efficient than materializing the data points.

5.4 Empirical Study

5.4.1 Setup and Datasets

We implemented the skip list approach, the algorithms and query processing techniques presented in this paper. The experiments were conducted on a 1.6GHz AMD Turion 64 machine with 1GB physical memory and a TOSHIBA MK8040GSX disk. The implementation is in Java. We performed the experiments on two sets of stock price data (from Commodity Systems, Inc.).

- IBM's stock price history data from January 3rd, 1966 to October 10th, 2007. This per second tick dataset is over 1 GB.
- McDonald's stock price history data from January 2nd, 1970 to October 10th, 2007. This dataset also has almost 1 GB of tick data.

The data is already normalized through the adjusted price. In order to verify the accuracy of predictions of different length of future time intervals, we go back one year in history and pretend it is now October 10th, 2006. We use data up to this date to build models and predict stock ticks at different "future" time intervals relative to October 10th 2006, for up to one year. Then we can use the actual stock prices from October 10th 2006 to the same day in 2007 to verify the accuracy of predictions using various methods.

5.4.2 Effectiveness of the Skip-list Approach

In the first experiment, we examine the effectiveness of the skip list approach for answering prediction queries.

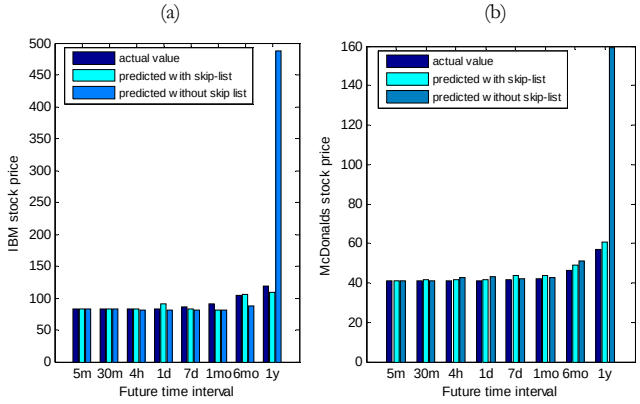


Fig. 5.4: Comparison of actual value and prediction results with and without the skip list approach for different prediction intervals.

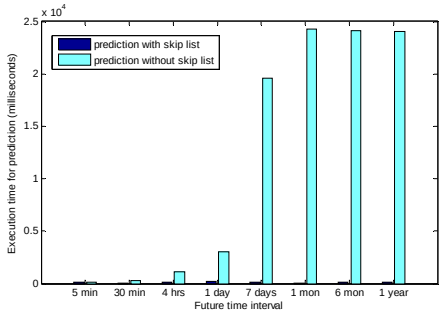


Fig. 5.5: Comparison of query execution time with and without the skip list approach.

We issue queries of the form "SELECT stock_price FROM IBM_ticks WHERE time = NOW + ?". Figure 5.4 shows the prediction accuracy with and without the skip list approach for seven prediction intervals. The figure shows the results for both datasets. Without using the skip list (the third bar), we directly build models on the original dataset and apply a limit (500,000) on the maximum number of data points that can be used, which we will explain next. Figure 5.5 compares the query execution time with and without the skip list approach.

From Figure 5.4, we can see that as prediction interval increases, the quality of prediction without the skip list approach drops rapidly. The reason is as follows. We showed that the "proper" history length increases with the prediction interval. Because a skip-list supports efficient retrieval of samples at different granularities, model building and, thus, model maintenance as well, only reads the necessary data as opposed to reading all the raw data within the same history length. Furthermore, accessing a level in the skip-list is a sequential scan requiring no additional seeks. Figure 5.5 shows that even for a short prediction interval of 4 hours, the query execution time without the skip list is already a few times longer since we are building models on the fly. We apply a limit (500,000) on the maximum number of data points used because (1) when beyond this limit, the model building takes so much memory and CPU that it runs too slowly on our test machine; and (2) at this limit it is already more than 300 times slower than using a skip list. Figure 5.5 indicates that when the prediction interval is one month or longer we already reach this limit. Figure 5.4 also shows when the query interval is one year, the history length available (subject to the limit on maximum number of data points) without using skip lists is too short to make a meaningful prediction.

5.4.3 Effectiveness of the PM's

In the second experiment, we examine the effectiveness of answering prediction queries using a set of PM's chosen by the ChoosePMSet algorithm subject to different levels of maintenance cost constraints. Typically, we assume that through profiling at the database server, for example, we can collect some statistics on the query workload, a PMF over a set of query intervals. In the experiment, we pick 27 intervals in the one-year window, ranging from 5 minutes to 1 year. We test with an arbitrary PMF, shown in Figure 5.6. Again we look at accuracy and speed.

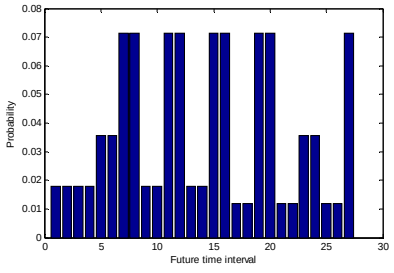


Fig. 5.6: Probability mass function (PMF) of future time intervals as the workload.

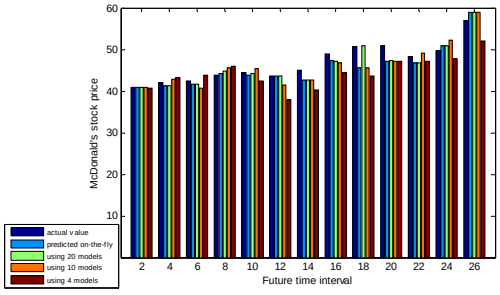


Fig. 5.7: Prediction accuracy using different number of PM's.

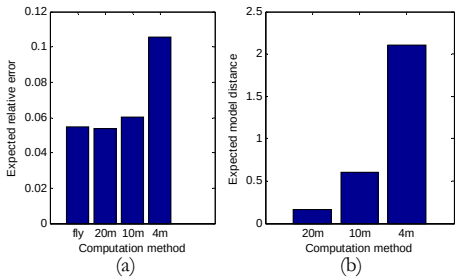


Fig. 5.8: Expected prediction error (a) and expected model distance metric computed using Theorem 2 (b) of using different number of PM's.

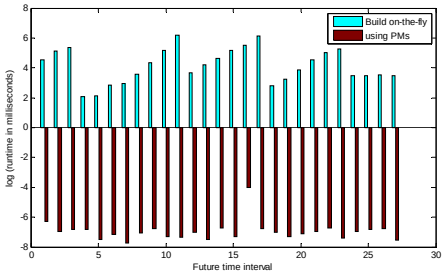


Fig. 5.9: Query execution time comparison between building models on-the-fly and using PM's.

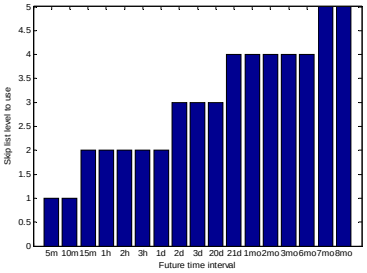


Fig. 5.10: Monotonicity of skip list level used for different query intervals.

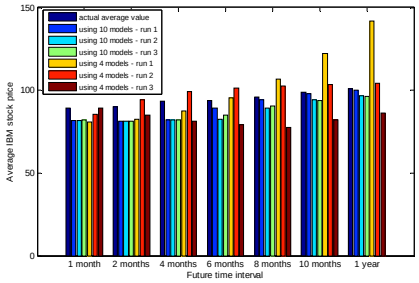


Fig. 5.11: Comparison of the prediction accuracy of average stock prices in different future time intervals under different number of PM's.

Figure 5.7, 5.8 and 5.9 show the results for the McDonald's dataset (Due to space constraints, we omit the figures for the IBM dataset in which we see similar trends). Each group of Figure 5.7 has five bars. They are the actual value, predicted on-the-fly, using the first 20 models selected by the ChoosePMSet algorithm, using the first 10 models, and using the first 4 models, respectively. For clarity, we only show the even number intervals (the other half shows similar information). Using 20 models is about as good as building models on the fly. Using 10 models is in fact also very close to this "best-we-can-do" result, and using 4 models is sometimes quite inaccurate compared to others.

Figure 5.8(a) summarizes the expected relative error of Figure 5.7 (but all 27 intervals) according to the workload PMF. The first bar is for building models on-the-fly to answer a query, and the other three bars are for answering with 20 PM's, 10 PM's and 4 PM's, respectively. We can see that the error of using 20 PM's is about the same as building models on the fly. Using 10 PM's is nearly as good, but using 4 PM's has significantly more error. Figure 5.8(b) simply plots the result from our theoretical analysis in Theorem 5.2 of the expected model distance of an incoming query. The model distance with 10 PM's is close to 20 PM's, while using 4 PM's has significantly bigger model distance. This is consistent with the result of Figure 5.8(a) on prediction errors.

Figure 5.9 compares the execution time of answering queries (for different intervals) by building models on the fly versus using PM's. Since the running time of using PM's, regardless of how many of them, is about the same, we only show one of them. Here we observe that the query processing time using PM's is negligible compared to building models on-the-fly. For both bars to be visible, we use log base 2 for the y axis. From the running time of building models on-the-fly, we can also observe that there are five groups (intervals 1 to 3, 4 to 11, 12 to 17, 18 to 23, and 24 to 27), within each of which the on-the-fly running time monotonically goes up (or stays about the same). Each of the five groups corresponds to the usage of a different level of the skip list, and within each level, as the query interval goes up, the history length, hence the number of data points used also goes up, which causes model building time to go up. This partially verifies the monotonicity assumption. We further verify the skip list level part of the monotonicity assumption later.

5.4.4 Monotonicity Assumption and Query Processing

In the third experiment, we verify the monotonicity assumption we made for query processing, and examine the prediction accuracy, as well as the stability (variance) of result, of an aggregation query using 10 PM's and 4 PM's.

Figure 5.10 shows that using our statistical testing of hypotheses algorithms for determining a proper history length and number of data points to use, we determine a skip list level to use which is monotonically increasing as interval goes up. We can narrow down the exact transition intervals that make a jump of the skip list level.

We next look at the processing result of an average query "SELECT AVG(stock_price) FROM IBM_ticks WHERE time BETWEEN NOW AND NOW + ?" using 10 and 4 PM's. Figure 5.11 shows the result for the IBM dataset. Due to space constraints, we omit the figure for the McDonald's dataset, as it leads us to the same conclusions. We have three runs using 10 PM's and three runs using 4 PM's. In each run, we start from ChoosePMSet, which is a randomized algorithm. Thus the result of query processing is also a random variable. The first bar in each group of Figure 5.11 shows the actual average value, and the next three bars are the results of three runs of using 10 PM's, and the last three bars are those of using 4PM's. We can see that using 10 PM's predicts the aggregation result pretty well, and the results of the three runs are close to each other, which indicates that the query processing result from 10 models is quite stable. On the other hand, using 4PM's, the result is about the same as 10 PM's in expectation. The 4 PM case, though, has a much larger variance, and hence the prediction result can be far off.

Chapter 6

RELATED WORK

6.1 Management of Uncertain Data

There is a broad range of related work on probabilistic databases, but as far as we know, none is built on top of a DBMS specialized for scientific/intelligence applications, or on a multi-dimensional array system. Perhaps the closest are those addressing imprecise and uncertain data in sensor networks [CK03]. In their work, the authors model a value distribution as a continuous PDF. This approach incurs a high cost and complexity when one has to deal with a huge amount of data (typical in scientific applications). Frequently one must resort to approximation and this has not been discussed before. Other work that studies value uncertainty and uses discrete PDF (like us) includes [BG92] and [BD05]. They both use discrete PDF in the same form as our heuristic algorithms. [BG92] only studies the result of conventional database operators, but not arbitrary mathematical operators in scientific applications. And it does not discuss the cost with a large amount of data. [BD05] specifically studies representing and querying ambiguous data in the OLAP setting where the focus is aggregation.

Dalvi and Suciu [DS04] studied a different problem of querying probabilistic databases: uncertain matches and ranking results. [RD07, TW04] are additional work on result ranking. Other work on the tuple uncertainty model includes [SD07]. In the latter, the authors also integrate uncertainty with data lineage. In multidimensional arrays of scientific databases, as in sensor networks, due to the different problem we are solving, we focus on value uncertainty, rather than set and tuple uncertainty.

There is also some existing work on statistical estimation of aggregation queries using random samples, such as in statistical databases [D80] and online aggregation [HH96]. However, there are fundamental differences. They only handle "certain" data while this paper deals with operations on "values" each of which is a probabilistic distribution. SERP does sampling but queries are not limited to aggregations and our statistical mode does not do sampling.

There has been recent interest in applying Monte Carlo algorithms for managing uncertain data [RD07] and for information recovery [XY08]. Re et al [RD07] use a Monte Carlo algorithm to obtain

top-k probability result tuples for certain types of queries in the discrete tuple uncertainty model. Jampani et al [JX08] propose a flexible framework to allow an uncertainty model to be dynamically parameterized and to represent uncertainty via VG functions, allowing arbitrary correlation. Our contributions differ significantly from all the above. We use general probabilistic graphical models inside array databases and combine the modeling, sampling, and query processing with the chunking mechanism of arrays for efficient I/O. Furthermore, we devise the S-Join algorithm to make the expensive JOIN operation in this context much more efficient. Finally, using information theory and statistics, we perform a study on optimizations in determining the cessation of sampling.

6.2 Top-*k* Queries on Uncertain Data

Re et al. [15] studied top-k queries on uncertain data where the ranking is based on the probability that a result tuple appears in the result. The semantics of top-k queries on uncertain data with arbitrary ranking functions was first studied by Soliman et al. [18]. The authors in [18] gave two kinds of semantics (U-Topk and U-kRanks) and devised optimal algorithms in terms of the number of accessed tuples and search states. Yi et al. [21] improved the time and space efficiency of the algorithms that compute U-Topk and U-kRanks results. Hua et al. [9] proposed a new semantics called probabilistic threshold top-k (PT-k). More recently, Jin et al. [13] studied top-k queries in the uncertain data stream setting.

As discussed in Chapter 5, we can classify the proposed semantics into two categories, both of which are useful for their own application scenarios. In this thesis, we extend the work in the first category and propose new semantics which shifts the emphasis more toward ranking scores. As we have discussed, our new semantics is useful for many applications that are not sufficiently addressed before.

Zhang and Chomicki [22] proposed the Global-Topk semantics which falls into the second category. Interestingly, in the future work section of [22], two of the open problems that the authors listed are: (1) integrating the strength of preference expressed by score into the semantics framework (i.e., existing semantics are not as sensitive to score as to probability) and (2) considering non-injective scoring functions (ties). Our work happens to address both of these open problems.

6.3 Modeling Correlation in Databases

Probabilistic graphical models have been widely used in statistical machine learning. Recently, this technique has been used in the database community to model the inherent correlation in data; both

from its sources and during query processing (e.g., JOIN). For example, Sen and Deshpande [SD07] use a graphical model to describe the dependencies among tuples or attributes and cast the query processing problem as an inference problem in a constructed graphical model. Wang et al. [WM08] propose a declarative first-order (or, relational) extension of BN models to capture correlations at various levels of granularity and have a clean separation of probabilistic models and relational data. Gupta and Sarawagi [GS06] studied methods of curating imprecise databases using MRF-based information extraction.

Our contributions differ significantly from all the above. We use general probabilistic graphical models inside array databases and combine the modeling, sampling, and query processing with the chunking mechanism of arrays for efficient I/O. Furthermore, using information theory and statistics, we perform a study on optimizations in determining the cessation of sampling.

Finally, our work on *A-trees* utilizes the special characteristics of multi-dimensional array data in scientific applications. As we demonstrate in this work, by taking advantage of the predictable and structured correlations that is often present in multidimensional data, we can provide a more efficient way of representing uncertainty in large-scale array data and of answering queries over this data.

6.4 Usage of Models in Query Processing and Prediction Queries

In the context of online and streaming applications, there has been previous work (e.g., [BS03] and [PV04]) that addresses a similar problem to ours, namely, query processing when there is a large amount of historical data. Bulut and Singh (in [BS03]) develop a technique using Discrete Wavelet Transform that summarizes a dynamic stream incrementally at multiple resolutions. Palpanas et al. (in [PV04]) introduce the notion of general amnesic functions which describe the precision loss for queries on different periods in the past.

The work in [BS03, PV04] concerns online streaming in which large amounts of historical data must be discarded, while our work is aimed at stored data. Often, fine-granularity historical data is needed for queries. Also, in the case of stock ticks or medical databases, there is often a regulatory requirement to store all the data. These days, large amounts of data are being generated by measurement infrastructures that continuously monitor a variety of things like military object positions or environmental properties. In these examples, the data volume is huge. Searching, for existing values, interpolating missing values, and predicting future values are all important. The skip

lists in our solution can be used for searching and interpolation in addition to prediction, making them more general than [BS03, PV04].

Furthermore, [BS03, PV04] addresses general queries on the past (point, range and "inner product" queries) while our work aims specifically at forecasting queries of various types: point, range, aggregations, and join. For forecasting queries, our skip list approach is simpler and more efficient in that (1) the database engine does not need to pay any computation overhead associated with maintaining and transforming data summaries; (2) the approach in [BS03, PV04] has to discard some recent data points to build a model that uses data points (almost) equidistant in time in order to ensure that the least square error metric for optimization is fair for all time periods in the chosen history length.

In fact, forecasting using data of higher sample frequency is a known problem in the literature [AB99]. In particular, the study in [AB99] shows that the improvement of forecasting results using higher sampling frequency can be quite dramatic. The skip list approach provides a platform to explore data of different densities.

The skip list data structure was invented in 1990 by Pugh [P90]. Its elegance and simplicity have drawn a lot of attention. Munro et al. [MP92] proposed a deterministic version to guarantee logarithmic costs. Aspnes and Shah [AS02] proposed skip graphs, which are a distributed structure based on skip lists, and provide the functionality of a balanced tree in a distributed system for fault tolerance. Abraham et al. proposed an improved version, so-called "skip B-trees", that combines the advantages of skip graphs with features of B-trees. There is also a project called "skipDB" which is a database implemented with a skip list instead of a B-tree. It is claimed to be transactional, portable, fast and small.

Time series is one of the primary special data types required within scientific databases [WG93]. There has been a lot of work, especially in data mining, on similarity and pattern matching in time series. To list but a few, work along these lines includes [FR94, PY06]. Time series forecasting has been a major focus for research in other fields. In particular, valuable tools for forecasting and time series processing appear in statistics and signal processing. [GH05] is a recent and comprehensive review of this research over the past 25 years.

In the context of databases, Yi et al. [YS00] developed a fast method to analyze co-evolving time sequences jointly to allow estimation or forecasting of missing/future values, quantitative data mining, and outlier detection. Tulone and Madden [TM06] presented a method for approximating the values of sensors in a wireless sensor network based on time series forecasting. Also in the context of sensor networks, Deshpande and Madden [DM06] developed view abstraction for the underlying interpolation and prediction models to support declarative queries. More recently, Duan and Babu [DB07] developed algorithms that can compose prediction operators into a good plan for a given query and dataset.

Our work differs from earlier work in important ways. We focus on the data management aspects, specifically, the scalability issue for predictive query processing when the time series data set is large. This is crucial for query performance as well as prediction accuracy since typically model building is expensive. We target the issue of choosing the right subset of data to answer prediction queries on a given future interval. We also discuss interesting query processing strategies for handling complex query types, whereas in [DB07], for example, only point queries are supported, but not other query types such as range query, aggregation, and join. Last but not least, our skip list approach also simultaneously provides search and interpolation capabilities.

Chapter 7

CONCLUSIONS

Querying processing on uncertain data is a young field. Yes it is a very active research topic because of the large number of applications in recent database systems. We have done some interesting work and developed a series of techniques that are instrumental for a workable architecture for query processing on uncertain data.

We propose a discrete treatment of probabilistic data in a database system for scientific and intelligence applications. In order to measure the result quality of an algorithm, we present a novel way to adopt a standard distribution distance metric into our context. We present SERP for computing the result distribution and prove an upper bound on the variation distance between its result distribution and the ideal one. We also propose a fast "statistical" mode of reporting results, which is sufficient and much more efficient for many applications and queries. Using statistical mode in the query evaluator also enables efficient evaluation of predicates.

A much needed requirement in managing uncertain data is continuous probability distributions that model correlated attributes. In this thesis work, we advance the understanding of this area. We define query semantics as an extension of the well-known possible world semantics. We incorporate piecewise probabilistic graphical model building with array chunking. We admit the difficulty of performing inference on such a model, and adopt Markov Chain Monte Carlo algorithms for query processing. Under this framework, we develop an efficient JOIN algorithm, and study entropy evolution of the result set and its relationship with result quality.

Uncertainty in multidimensional array database systems must be carefully handled before such systems can efficiently and correctly handle scientific data. Correlations are common in such data and they are usually structured along dimensions. Based on this observation, we develop a novel data structure, called A-tree, which is a unified model for storage and modeling of such data. We demonstrate that compared to alternative approaches, A-tree can not only perform inference much more efficiently, but it also models the underlying joint distribution accurately. A systematic empirical study is conducted on both real and synthetic datasets.

For the new and important topic of top-*k* queries on uncertain data, we observe the need to shift the emphasis a little more on ranking scores, as opposed to the probabilities for many applications. We propose to provide the score distribution of top-k vectors and c-Typical-Topk answers to applications and devise efficient algorithms to cope with the computational challenges. We also extend the work to score ties. Experimental results verify our motivation and our approaches.

Finally, we have also done some work for the newly arising prediction queries that ask for uncertain future data. We address the scalability issue on processing prediction queries on large time series data sets, which are often seen in financial and scientific databases. We propose statistical tests of hypotheses to determine a proper subset of data points to use for a given query interval. We adopt the skip list data structure, make it I/O conscious, and use it as samples for our query purpose, in addition to the search capability that a skip list already provides. We further present an algorithm *ChoosePMSet* to choose a set of models to pre-build (PM), subject to some maintenance cost constraint. We discuss query processing strategies using the PM's.

BIBLIOGRAPHY

[GZ07a] T. Ge and S. Zdonik. Fast, Secure Encryption for Indexing in a Column-Oriented DBMS. In *ICDE*, 2007.

[GZ07b] T. Ge and S. Zdonik. Answering Aggregation Queries in a Secure System Model. In *VLDB*, 2007.

[GZ08a] T. Ge and S. Zdonik. Handling Uncertain Data in Array Database Systems. In *ICDE*, 2008.

[GZ08b] T. Ge and S. Zdonik. A Skip-list Approach for Efficiently Processing Forecasting Queries. In *VLDB*, 2008.

[GZ09a] T. Ge and S. Zdonik. Light-weight, Runtime Verification of Query Sources. In *ICDE*, 2009.

[GZ09b] T. Ge, S. Zdonik, and S. Madden. Top-k Queries on Uncertain Data: On Score Distribution and Typical Answers. In *SIGMOD* 2009.

[GG09c] T. Ge, D. Grabiner, and S. Zdonik. A Treatment of Correlated Attribute Uncertainty in Array Database Systems. In submission.

[GZ09d] T. Ge and S. Zdonik. *A-tree*: A Structure for Storage and Modeling of Uncertain Multidimensional Arrays. In submission.

[SB07] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? – Part 2: benchmarking results. In CIDR, January, 2007.

[BG92] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. In TKDE, 1992.

[BS06] O. Benjelloun, A. Das Sarma, A. Halevy, J. Widom. ULDBs: Databases with Uncertainty and Lineage. In VLDB, 2006.

[BK06] P. Brohan, J. Kennedy, I. Haris, S.F.B. Tett and P.D. Jones. Uncertainty estimates in regional and global observed temperature changes: a new dataset from 1850. In Geophysical Research, 2006.

[BD05] D. Burdick, P. Deshpande, T. Jayram, R. Ramakrishnan, S. Vaithyanathan. OLAP over uncertain and imprecise data. VLDB'05.

[CK03] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In SIGMOD, 2003.

[CS06] R. Cheng, S. Singh, and S. Prabhakar. Efficient join processing over uncertain data. In CIKM, 2006.

[DS04] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In VLDB, 2004.

[D80] D. Denning. Secure statistical databases with random sample queries. In TODS, Volume 5, Issue 3, September 1980.

[DG04] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, W. Hong. Model-driven data acquisition in sensor networks. In VLDB, 2004.

[D86] L. Devroye. Non-Uniform Random Variate Generation. Chapter 2. New York: Springer-Verlag, 1986.

[FR97] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. ACM Transactions on Information Systems, 1997.

[H70] J. Halton. A retrospective and prospective survey of the Monte Carlo method. In SIAM Review, Vol. 12, Jan. 1970.

[HH96] J. Hellerstein, P.Haas, H. Wang. Online aggregation. SIGMOD'96.

[OR86] F. Olken and D. Rotem. Simple random sampling from relational databases. In VLDB, Kyoto, August, 1986.

[RD07] C. Re, N. Dalvi and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In ICDE, 2007.

[SD07] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In ICDE, 2007.

[S01] J. Stewart. Calculus: concepts and contexts (2nd ed.). Thomson Learning, Inc. 2001.

[SB07] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? – Part 2: benchmarking results. In CIDR, January, 2007.

[TW04] M. Theobald, G. Weikum, R. Schenkel. Top-k Query Evaluation with Probabilistic Guarantees. In VLDB, 2004.

[PA08] http://www.navcen.uscg.gov/mwv/vts/PAWSS.htm.

[T08] http://www.cru.uea.ac.uk/cru/data/temperature/.

[GPS08] http://en.wikipedia.org/wiki/Global_Positioning_System.

[AK91] Abiteboul, S., Kanellakis, P., Grahne, G.: On the Representation and Querying of Sets of Possible Worlds. In TCS, 78(1), 1991.

[AJ08] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. In ICDE, 2008.

[BB01] Y. Bar-Shalom, D. William Dale Blair. Multitarget-Multi-sensor Tracking: Applications and Advances, Vol. III, Artech House, Boston, London, 2001.

[BK06] Brohan, P., J. Kennedy, I. Haris, S.F.B. Tett, P.D. Jones. Uncertainty estimates in regional and global observed temperature changes: a new dataset from 1850. In Journal of Geophysical Research, 2006.

[CP87] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In VLDB, 1987.

[CS06] R. Cheng, S. Singh, and S. Prabhakar. Efficient join processing over uncertain data. In CIKM, 2006.

[DS05] N. Dalvi and D. Suciu. Query answering using statistics and probabilistic views. In VLDB, 2005.

[DS96] D. Dey and S. Sarkar. A probabilistic relational model and algebra. In ACM Trans. on Database Systems, 1996

[G06] L. Getoor. An Introduction to Probabilistic Graphical Models for Relational Data. IEEE Data Eng. Bull. 29(1): 32-39, 2006.

[G89] G. Grahne. Horn tables - an efficient tool for handling incomplete information in databases. In PODS, 1989.

[HP08] M. Hua, J. Pei, W. Zhang, X. Lin. Ranking Queries on Uncertain Data: A Probabilistic Threshold Approach. In SIGMOD, 2008.

[IB08] I. Ilyas, G. Beskales, and M. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. In ACM Computing Surveys, 2008.

[JX08] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data. In SIGMOD, 2008.

[J98] M. Jordan. Learning in Graphical Models. The MIT press. Nov., 1998.

[LL97] Lakshmanan, L.V.S., Leone, N., Ross, R., Subrahmanian, V.: ProbView: A Flexible Probabilistic Database System. In ACM TODS 22(3), 1997.

[MS02] A. Marathe and K. Salem. Query Processing Techniques for Arrays. In VLDB Journal 11: 68-91, 2002.

[PI97] V. Poosala and Y. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. VLDB, 1997.

[RD07] C. Re, N. Dalvi and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In ICDE, 2007.

[SS94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In Proceedings of the 10th International Conference on Data Engineering (ICDE), 1994.

[SM08] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville and R. Cheng. Database Support for Probabilistic Attributes and Tuples. In ICDE, 2008.

[XY08] J. Xie, J. Yang, Y. Chen, H. Wang, and P. Yu. A Sampling-Based Approach to Information Recovery. In ICDE, 2008.

[YL08] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient Processing of Top-k Queries on Uncertain Databases. In ICDE, 2008.

[AA05] Abraham, I., Aspnes, J., and Yuan, J. Skip B-trees. In Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005).

[AB99] T. Andersen, T. Bollerslev, and S. Lange. Forecasting financial market volatility: Sample frequency vis-à-vis forecast horizon. In Journal of Empirical Finance, Dec 1999, pages 457-477.

[AK04] James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load Balancing and Locality in Range-Queriable Data Structures. In Twenty-Third ACM Symposium on Principles of Distributed Computing, pages 115–124, July 2004.

[AS02] James Aspnes and Gauri Shah. Skip Graphs. In Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 384–393, January 2002.

[B43] Boyer, C. B. Pascal's Formula for the Sums of Powers of the Integers. In Scripta Math. 9, 237-244, 1943.

[BD02] Brockwell, P., and Davis, R. Introduction to Time Series and Forecasting. 2nd Edition. Springer Texts in Statistics. 2002.

[BS03] Bulut, A. and Singh, A.K. SWAT: Hierarchical Stream Summarization in Large Networks. In ICDE, 2003.

[DH05] J. G. De Gooijer and R. J. Hyndman. 25 Years of IIF Time Series Forecasting: A Selective Review. June 2005. Tinbergen Institute Discussion Papers No. TI 05-068/4.

[DM06] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In SIGMOD 2006.

[DB07] Duan S., and Babu, S. Processing Forecasting Queries. In VLDB, 2007.

[E06] Eubank, R.L. A Kalman Filter Primer. Chapman & Hall/CRC, 2006.

[FR94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In SIGMOD, 1994.

[GK95] D. Goldin and P. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In CP'95, Sep 1995.

[HK07] Hyndman, R.J., Kostenko, A.V. Minimum Sample Size Requirements for Seasonal Forecasting Models. In Foresight, Issue 6, Spring 2007.

[MW98] Makridakis, S., Wheelwright S., and Hyndman, R. Forecasting Methods and Applications. Third Edition. John Wiley & Sons, Inc. 1998.

[MS94] Mendenhall, W., and Sincich, T. Statistics for Engineering and the Sciences. Fourth Edition. Prentice-Hall, Inc. 1994.

[MC84] Mentzer, J.T., and J.E. Cox Jr. Familiarity, Application and Performance of Sales Forecasting Techniques. In Journal of Forecasting, 3, 1984, 27-36.

[MP92] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic Skip Lists. In Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92). Orlando, Florida, United States. pp. 367-375.

[N91] Leonore Neugebauer. Optimization and Evaluation of Database Queries Including Embedded Interpolation Procedures. In SIGMOD, 1991.

[PV04] Palpanas, T., Vlachos, M., Keogh, E., Gunopulos, D., and Truppel, W. Online amnesic approximation of streaming time series. In ICDE, 2004.

[PS06] Papadimitriou, S., Sun, J., and Yu, P. Local Correlation Tracking in Time Series. In ICDM, 2006.

[PY06] Papadimitriou, S., and Yu, P. Optimal Multi-scale Patterns in Time Series Streams. In SIGMOD, 2006.

[P90] Pugh, W. Skip lists: a probabilistic alternative to balanced trees. In Communications of the ACM, June 1990, 33(6) 668-676.

[S80] Schultz, H. J. The Sums of the k'th Powers of the First n Integers. In Amer. Math. Monthly 87, 478-481, 1980.

[S01] J. Stewart. Calculus: Concepts and Contexts (2nd ed.). Thomson Learning, Inc. 2001.

[TM06] D. Tulone and S. Madden. PAQ: Time series forecasting for approximate query answering in sensor networks. In EWSN, 2006.

[WS01] Whitney, A., and Shasha, D. Lots o' Ticks: Real-time High Performance Time Series Queries on Billions of Trades and Quotes. In SIGMOD, 2001.

[WD96] Winklhofer, H., A. Diamantopoulos, and S.F. Witt. Forecasting Practice: A Review of the Empirical Literature and an Agenda for Future Research. In International Journal of Forecasting, 12, June 1996, 193-221.

[WG93] Wolniewicz, R., and Graefe, G. Algebraic Optimization of Computations over Scientific Databases. In VLDB, 1993.

[YS00] B. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online Data Mining for Co-evolving Time Sequences. In ICDE, 2000.

[ZS03] Zhu, Y., and Shasha, D. Query by Humming: a Time Series Database Approach. In SIGMOD, 2003.

[S08] http://dekorte.com/projects/opensource/SkipDB/.

[EG08] J. Eriksson, L. Girod, B. Hull, R. Newton, S. Madden, and H. Balakrishnan. The Pothole Patrol: Using a Mobile Sensor Network for Road Surface Monitoring. In MobiSys, 2008.

[HB06] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In SenSys, 2006.

[BW01] Y. Bar-Shalom, D. William Dale Blair. Multitarget-Multi-sensor Tracking: Applications and Advances, Vol. III, Artech House, Boston, London, 2001.

[BD06] Benjelloun, O., Das Sarma, A., Halevy, A. and Widom, J. ULDBs: Databases with Uncertainty and Lineage. In VLDB, 2006.

[BK06] Brohan, P., J. Kennedy, I. Haris, S.F.B. Tett, P.D. Jones. Uncertainty estimates in regional and global observed temperature changes: a new dataset from 1850. In Journal of Geophysical Research, 2006.

[CP87] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In VLDB, 1987.

[CK03] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In SIGMOD, 2003.

[CS06] R. Cheng, S. Singh, and S. Prabhakar. Efficient join processing over uncertain data. In CIKM, 2006.

[CT91] T. M. Cover and J. A. Thomas. Elements of Information Theory. A Wiley-Interscience Publication, 1991.

[DS04] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In VLDB, 2004.

[DS05] N. Dalvi and D. Suciu. Query answering using statistics and probabilistic views. In VLDB, 2005.

[DS96] D. Dey and S. Sarkar. A probabilistic relational model and algebra. In ACM Trans. on Database Systems, 1996

[FR97] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. ACM Transactions on Information Systems, 1997.

[G06] L. Getoor. An Introduction to Probabilistic Graphical Models for Relational Data. IEEE Data Eng. Bull. 29(1): 32-39, 2006.

[G07] G. Grahne. Horn tables - an efficient tool for handling incomplete information in databases. In PODS, 1989.

[GS06] R. Gupta and S. Sarawagi. Curating probabilistic databases from information extraction models. In VLDB, 2006.

[HP08] M. Hua, J. Pei, W. Zhang, X. Lin. Ranking Queries on Uncertain Data: A Probabilistic Threshold Approach. In SIGMOD, 2008.

[IB08] I. Ilyas, G. Beskales, and M. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. In ACM Computing Surveys, 2008.

[JX08] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data. In SIGMOD, 2008.

[J98] M. Jordan. Learning in Graphical Models. The MIT press. Nov., 1998.

[LL97] Lakshmanan, L.V.S., Leone, N., Ross, R., Subrahmanian, V.: ProbView: A Flexible Probabilistic Database System. In ACM TODS 22(3), 1997.

[MS02] A. Marathe and K. Salem. Query Processing Techniques for Arrays. In VLDB Journal 11: 68-91, 2002.

[MU05] M. Mitzenmacher and E. Upfal. Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, 2005.

[PI97] V. Poosala and Y. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. VLDB, 1997.

[RD07] C. Re, N. Dalvi and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In ICDE, 2007.

[SS94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In Proceedings of the 10th International Conference on Data Engineering (ICDE), 1994.

[SD07] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In ICDE, 2007.

[SM08] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville and R. Cheng. Database Support for Probabilistic Attributes and Tuples. In ICDE, 2008.

[B06] C. Bishop. Pattern Recognition and Machine Learning. Springer, 2006.

[CP87] R. Cavallo, M. Pittarelli. The theory of probabilistic databases. In VLDB, 1987.

[CG01] K. Chakrabarti, M. Garofalakis, R. Rastogi, K. Shim. Approximate query processing using wavelets. In VLDB Journal, 2001.

[CA98] C. Chang, A. Acharya, A. Sussman, J. Saltz. T2: a customizable parallel database for multi-dimensional data. In SIGMOD, 1998.

[CM97] Chang C, Moon B, Acharya A, Shock C, Sussman A, Saltz JH. Titan: a high-performance remote sensing database. In ICDE, 1997.

[CK03] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In SIGMOD, 2003.

[CC05] A. Choi, H. Chan, A. Darwiche. On Bayesian Network Approximation by Edge Deletion. In UAI, 2005.

[CLRS] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms (2nd edition). MIT Press and McGraw-Hill.

[CT91] T. M. Cover and J. A. Thomas. Elements of Information Theory. A Wiley-Interscience Publication, 1991.

[DS04] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In VLDB, 2004.

[DK06] T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In Interface 2006.

[D96] R. Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In UAI, 1996.

[DS06] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In SIGMOD, 2006.

[DS96] D. Dey and S. Sarkar. A probabilistic relational model and algebra. In ACM Trans. on Database Systems, 1996.

[FG99] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning Probabilistic Relational Models. In IJCAI, 1999.

[FR97] N. Fuhr, T. Rolleke. A probabilistic relational algebra for the inte-gration of information retrieval and database systems. TOIS, 1997.

[JX08] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, P. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data.SIGMOD‘08.

[J98] M. Jordan. Learning in Graphical Models. The MIT press. 1998.

[LL97] Lakshmanan, L., Leone, N., Ross, R., Subrahmanian, V.. ProbView: A Flexible Probabilistic Database System. In TODS 22(3), 1997.

[MS02] A. Marathe and K. Salem. Query Processing Techniques for Arrays. In VLDB Journal 11: 68-91, 2002.

[MU05] M. Mitzenmacher, E. Upfal. Probability & Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge U. Press, 2005.

[O93] F. Olken. Random Sampling from Databases. PhD Thesis, UC Berkeley, 1993.

[RD07] C. Re, N. Dalvi and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In ICDE, 2007.

[SD07] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In ICDE, 2007.

[SM08] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, R. Cheng. Database Support for Probabilistic Attributes and Tuples. In ICDE, 2008.

[WM08] D. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models. In VLDB, 2008.

[AJ08] L. Antova, T. Jansen, C. Koch, D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. In ICDE, 2008.

[B57] R. Bellman. Dynamic Programming. Princeton Univ. Press, 1957.

[CK03] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In SIGMOD, 2003.

[CT91] T. Cover and J. Thomas. Elements of Information Theory. A Wiley-Interscience Publication, 1991.

[DS04] N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In VLDB, 2004.

[DS07] N. Dalvi and D. Suciu. Management of Probabilistic Data: Foundations and Challenges. In PODS, 2007.

[FK04] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, and E. Vee. Comparing and Aggregating Rankings with Ties. In PODS, 2004.

[HT91] R. Hassin and A. Tamir. Improved complexity bounds for location problems on the real line. In Operations Research Letters, 1991.

[HP08] M. Hua, J. Pei, W. Zhang, X. Lin. Ranking Queries on Uncertain Data: A Probabilistic Threshold Approach. In SIGMOD, 2008.

[HB06] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In SenSys, 2006.

[IB08] I. Ilyas, G. Beskales, and M. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. In ACM Computing Surveys, Vol. 40, 2008.

[JX08] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data. In SIGMOD, 2008.

[JY08] C. Jin, K. Yi, L. Chen, J. Yu, X. Lin. Sliding-Window Top-k Queries on Uncertain Streams. In VLDB, 2008.

[LB08] S. Lim, H. Balakrishnan, D. Gifford, S. Madden, D. Rus. Stochastic Motion Planning and Applications to Traffic. In WAFR, 2008.

[RD07] C. Re, N. Dalvi and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In ICDE, 2007.

[R95] K. Rosen. Discrete Mathematics and Its Applications. 1995.

[SD07] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In ICDE, 2007.

[SI07]M. Soliman, I. Ilyas, and K. Chang. Top-k Query Processing in Uncertain Databases. In ICDE, 2007.

[TB04] N. Tatbul, M. Buller, R. Hoyt, S. Mullen, S. Zdonik. Confidence-based Data Management for Personal Area Sensor Networks. In DMSN, 2004.

[W05] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In CIDR, 2005.

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

[XY08]J. Xie, J. Yang, Y. Chen, H. Wang, and P. Yu. A Sampling-Based Approach to Information Recovery. In ICDE, 2008.

[YL08] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient Processing of Top-k Queries on Uncertain Databases. In ICDE, 2008.

[SI07] M. Soliman, I. Ilyas, and K. Chang. Top-k Query Processing in Uncertain Databases. In ICDE, 2007.

[ZC08] X. Zhang and J. Chomicki. On the Semantics and Evaluation of Top-k Queries in Probabilistic Databases. In *DBRank*'08 Workshop.