U-DBMS: A Database System for Managing Constantly-Evolving Data

Reynold Cheng[†] Sarvjeet Singh[§] Sunil Prabhakar[§]

[†] Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong.
Email: {csckcheng}@comp.polyu.edu.hk

[§]Department of Computer Science, Purdue University, West Lafayette, IN 47907-2066, USA.
Email: {singh35,sunil}@cs.purdue.edu

Abstract

In many systems, sensors are used to acquire information from external environments such as temperature, pressure and locations. Due to continuous changes in these values, and limited resources (e.g., network bandwidth and battery power), it is often infeasible for the database to store the exact values at all times. Queries that uses these old values can produce invalid results. In order to manage the uncertainty between the actual sensor value and the database value, we propose a system called **U-DBMS**. U-DBMS extends the database system with uncertainty management functionalities. In particular, each data value is represented as an interval and a probability distribution function, and it can be processed with *probabilistic query operators* to produce imprecise (but correct) answers. This demonstration presents a PostgreSQL-based system that handles uncertainty and probabilistic queries for constantly-evolving data.

1 Introduction

Due to advances in sensor technologies, systems that acquire information from the physical world have attracted tremendous research interests in recent years. Sensornets [2], for example, acquire temperature, pressure and voltage through a sensor network. Location-based applications allow locations of targets to be constantly monitored and queried [5]. A common problem that is shared by this kind of systems is *data uncertainty*. While limited resources such as network bandwidth and battery power only allow data to be collected in a discrete manner, physical entities like temperature and locations are often constantly changing with time. As a result, the values of the data stored in the system can be inconsistent with the actual ones [5, 1]. This problem can be aggravated if updates are delayed or lost during transmission.

The inconsistency, or *uncertainty*, between the database value and the actual value can lead to erroneous query results [1]. In order to avoid drawing incorrect conclusions due to data uncertainty, the idea of introducing uncertainty information into data has been recently proposed [5, 1]. Instead of storing the data value received, each data item is modeled as a range of possible values, together with a probability density function that describes the distribution of the values within the range. This model captures the uncertainty of applications that deal with constantly-evolving data. For example, the system can negotiate with the sensor a fixed bound d. If the system does not receive any update from the sensor, then the sensor's current value can be assumed to be within the uncertainty interval $[v-d, v+d]$ (where v is the value of the sensor last reported to the server) [5].

To evaluate data uncertainty, *probabilistic queries* have been proposed [5, 1]. Answers to probabilistic queries are *imprecise*. The imprecision of answers is expressed by the probability values augmented to them. Consider a query asking "What is the id of the sensor that yields the highest temperature value?". A probabilistic query may yield $\{(S_1, 0.7), (S_2, 0.2), (S_3, 0.1)\}$ as the answer, where (S_i, p_i) is the probability that sensor S_i has a chance of p_i of giving the highest temperature value. Observe that the probability values accompanied with the probabilistic answers tells us about the validity of the answers.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

Recently systems have been proposed to manage data uncertainty. For example, the Trio system [4] attempts to build a general database that supports data accuracy and lineage, while in [2], a probabilistic model-based data acquisition system has been proposed for sensor networks.

Our prototype, called **U**ncertainty-**D**atabase **M**anagement **S**ystem (or U-DBMS in short), aims at providing uncertainty management for constantly-evolving data. It is built on top of PostgreSQL [3], an object-oriented relational open-source database system. The novel features of U-DBMS are:

1. Meta-queries for specifying and deriving data uncertainty;
2. Extending semantics of a wide class of SQL operators to support probabilistic queries;
3. Measurement of probabilistic answer quality; and
4. A modular design that allows easy addition of new uncertainty types and query operators.

In Section 2 we discuss various issues of probabilistic queries. Section 3 describes the system architecture of U-DBMS. Section 4 presents the materials being demonstrated, and Section 5 concludes the paper.

2 Probabilistic Queries

2.1 Probabilistic Uncertainty Model

To capture data uncertainty, a data scheme known as *probabilistic uncertainty model* was proposed in [1]. This model assumes that each data item can be represented by a range of possible values and their distributions. Formally, let T be a database of size n, and each tuple of T be T_i (where $i = 1, \ldots, n$). Each tuple T_i consists of a *uncertain attribute* a with two elements:

Definition 1 *An* **uncertainty interval** *of a, denoted by $a.U$, is an interval $[a.l, a.r]$ where $a.l, a.r \in \Re$, $a.r \geq a.l$ and $a \in a.U$.*

Definition 2 *An* **uncertainty pdf** *of a, denoted by $a.f(x)$, is a probability distribution function of a, such that $\int_{a.l}^{a.r} a.f(x)dx = 1$ and $a.f(x) = 0$ if $x \notin a.U$.*

As an example, $a.U$ can be a fixed bound d, which is a result of negotiation between the database system and the sensor [5]. An example uncertainty pdf is the Gaussian distribution, which models the measurement inaccuracy of location data [5] and data from sensor network [2]. It can also be a uniform distribution, which represents the worst-case uncertainty within a given uncertainty interval. Next, we investigate queries that operate on this uncertainty model.

2.2 Classification of Probabilistic Queries

We describe a classification scheme for probabilistic queries [1]. To justify why classification is needed, notice that queries in the same class have similar evaluation algorithms. Another reason is that probabilistic queries produce probabilistic (or inexact) answers. The vagueness of a probabilistic answer is captured by its *quality metrics*, which are useful to decide whether the answer is too ambiguous and any action needs to be done to reduce data uncertainty. Quality metrics differ according to the query class. To illustrate, denote (T_i, p_i) be the probability p_i that T_i is the answer. In a range query, an answer $(T_1, 0.9)$ is better than $(T_1, 0.1)$ since for the former answer are more confident that it is within a user-specified range, compared with the latter one with only a half chance of satisfying the query. A simple metric such as $\frac{|p_i - 0.5|}{0.5}$ can indicate the quality of the answer. For a maximum query, the answer $\{(T_1, 0.8), (T_2, 0.2)\}$ is better than $\{(T_1, 0.5), (T_2, 0.5)\}$, since from the first answer we are more confident that T_1 gives the maximum value. To capture this, an entropy-based metric is needed [1].

Probabilistic queries can be classified in two ways. First, we can classify them according to the forms of answers required. An **entity-based query** returns a set of objects, whereas a **value-based query** returns a single numeric value. Another criterion is based on whether an aggregate operator is used to produce results. An **aggregate query** is one which involves operators like `MAX`, `AVG` – for these operators, an interplay between objects determines the results. In contrast, for a **non-aggregate query**, the suitability of an object as the result to an answer is independent of other objects. A range query is a typical example. Based on this classification, we obtain four query classes.

(1) Value-based Non-Aggregate Class. An example of this class is to return the uncertain attribute values a which are larger than a constant.

(2) Entity-based Non-Aggregate Class. One example query is the range query: given a closed interval $[l, u]$, a list of tuples (T_i, p_i) are returned, where p_i is the non-zero probability that $T_i.a \in [l, u]$. Another example is a join over two tables R and S. It returns a pair of tuples (R_i, S_j, p_{ij}), where p_{ij} is the probability that the two tuples R_i and S_j join (using comparison operators such as $=, \neq, >, <$).

(3) Entity-based Aggregate Class. An example is the entity-minimum query: a set of tuples (T_i, p_i) are returned, where p_i is the non-zero probability that $T_i.a$ is the minimum among all items in T. Other examples are maximum and nearest-neighbor queries.

(4) Value-based Aggregate Class. Queries for this class include any aggregate operators (e.g., addition, subtraction, multiplication, division, `SUM`, `AVG`) that involves two or more values. This query yields $l, u \in \Re$ and $\{p(x) | x \in [l, u]\}$, where X is a random variable for the sum of values of a for all objects in T,

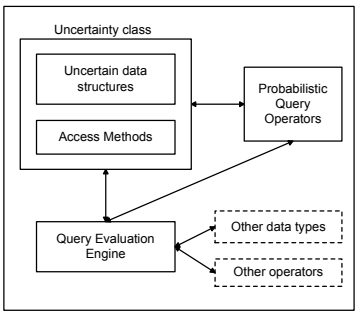

Figure 1: The architecture of U-DBMS

and $p(x)$ is a pdf of X such that $\int_l^u p(x)dx = 1$.

The details of evaluation and quality metrics of each query class can be found in [1]. Next, we discuss how query classes are supported in U-DBMS.

3 System Architecture

We develop our system on PostgreSQL [3] because it is an open-source system. Also, its object-oriented design allows us to extend the functionalities easily without modifying its internal codes. We define new data types and queries through developing external C libraries, and linking them with the PostgreSQL source codes. Another advantage is that the uncertainty functionalities do not interfere with the original database; instead, uncertain and certain data can be "blended" together, and they can be used by database queries at the same time. The high level architecture of U-DBMS is shown in Figure 1.

The interface of the query evaluation engine is modified to interact with both the uncertainty class and query operators. All other existing data types and operators in the PostgreSQL system (dotted-line boxes) remain intact. Next, we discuss how the data types and queries are implemented in U-DBMS.

3.1 Supporting Uncertainty Data

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

In order to represent these data types, we define the *uncertain* class (with keyword **UNCERTAIN**), as shown in Figure 1. It is a variable-length data type, which can store an uncertain value (Gaussian, uniform or histogram). The design of this class is flexible, and other kinds of uncertainty pdf (e.g. Poisson) can be added to it with minimal change.

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

To create a new uncertain data type, we use shared C libraries to specify the internal representation of the data type, along with "helper functions" that operate on the data types. These access functions are specified by the interface that PostgreSQL uses to interact with a data type. The query engine interacts with the uncertain data type through these access functions, as shown in Figure 1. Once these helper functions are properly set, the uncertain data type becomes one of the data types in PostgreSQL.

3.2 Supporting Probabilistic Queries

To support probabilistic queries, we provide PostgreSQL with the semantics of operations like $=, \neq, >, <$ on each uncertainty type, using compiled C functions (Figure 1).We have implemented the four classes of probabilistic queries by defining operations between an uncertain value and a constant, or between two uncertain values.

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

4 Demonstration

4.1 Meta-Queries for Uncertainty

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

```
CREATE table T (
k INTEGER PRIMARY KEY primary key,
a UNCERTAIN);
```

This schema is used for further discussions. An uncertain value is inserted as follows:

```
insert into T values (1,'(g, μ, σ)');
```

Here, g specifies that Gaussian distribution is used and μ and σ are the parameters of the distribution for this item. The bounds of the uncertainty interval are not specified explicitly; an environment variable is used to input the tail-cutoff percentage of the Gaussian distribution in order to convert it to a normalized distribution bounded by the uncertainty interval. The environment variable is changeable by users.

Similarly, an uncertain value modeled by a histogram is represented as '(h, min, max, f, $data_1$, $data_2$, ..., $data_f$)'. Here, min and max are the boundaries of the interval, and $f \geq 2$ is the number of sample points. The value $data_i$ $(i = 1, \ldots, f)$ specifies the probability of the variable at $min + \frac{(i-1)(max-min)}{f-1}$.

If the user does not know the nature of the distribution, an operator called `DERIVE_U` can be used. Given a series of data values, `DERIVE_U` returns the uncertainty interval and pdf for it.

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

```
select Lower_Bound(a) from T;
```

In the demonstration, the uncertain pdfs are displayed as histograms graphically.

4.2 Probabilistic Queries

We will demonstrate queries from each of the four probabilistic queries classes discussed in Section 2.

(1) Value-based Non-Aggregate Class. In this class, we show the following query (Q_1):

```
SELECT a FROM T WHERE a > 5;
```

The uncertainty information of a whose probability of being larger than 5 is non-zero will be displayed.

(2) Entity-based Non-Aggregate Class. An example for this class is a join query (Q_2):

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

The uncertainty attribute a is used to perform an equality join between tables R and S. The pairs that join, $(R.k, S.k)$, as well as their probability of join (obtained by function `p_equ`), are returned.

(3) Entity-based Aggregate Class. Here an example is entity-minimum query (Q_3).

```
SELECT k, p_Emin(a) FROM T;
```

This query returns a list of the primary keys of tuples, together with their (non-zero) probabilities of giving the minimum value (given by `p_Emin`).

(4) Value-based Aggregate Class. An example for this class is a value-minimum query (Q_4):

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

This query returns the minimum of the uncertain attributes a from all the tuples in T. The minimum value returned is an uncertain value itself.

4.3 Probability Threshold and Quality

A user may only want to obtain answers whose probabilities of satisfying a query is larger than a certain threshold. U-DBMS allows the user to specify this "probability threshold". In Q_1, for example, a user may specify a probability threshold of 0.8 by using "$p(a > 5) > 0.8$" in the `WHERE` clause. This returns all values of a, the value of each is larger than 5 with a probability of 0.8 or higher.

As also mentioned in Section 2, each query class has its own metric for measuring the quality of probabilistic query. U-DBMS provides a function called *Quality* to return a quality score. Any probabilistic query can be passed to *Quality*. Then, *Quality* decides the query class and selects the relevant quality metric to compute the quality score.

5 Conclusions

Data uncertainty prevails in systems that monitors constantly-evolving objects. In this paper we propose a database system called U-DBMS to manage data uncertainty. U-DBMS is built upon its PostgreSQL, which allows new data types and query operators to be integrated into it easily. Moreover, the uncertain data can be stored and queried together with the existing database. In the demonstration we will show the managing facilities of U-DBMS for four classes of probabilistic queries.

Acknowledgments

This research is supported by NSF Grants IIS 9985019 and CCR-0010044. We thank the anonymous reviewers for their insightful comments.

References

[1] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proc. of the ACM SIGMOD 2003*.

[2] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *CIDR 2005*.

[3] PostgreSQL Global Development Group PostgreSQL 8.0. http://www.postgresql.org/, 2005.

[4] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR 2005*.

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