

ORION: Concepts, Usage, and Installation

February 22, 2006

Contents

1	Summary	4
2	Background	4
2.1	The Data Uncertainty Problem	4
2.2	Probabilistic Uncertainty Model	6
2.3	Classification of Probabilistic Queries	6
2.4	Query Closure and Probability Thresholds	7
3	The ORION System	8
3.1	Supporting Uncertainty Data	9
3.2	Supporting Probabilistic Queries	9
3.3	Global Environment	10
4	The UNCERTAIN Data Type	10
4.1	Data Structure of UNCERTAIN data type	11
4.2	Input and Output Functions for Uncertain Data	12
4.2.1	Output Functions	14
4.3	Inquiry Functions	15
4.3.1	Examples	16
5	Probabilistic Query Operators	18
5.1	Comparison Operators	18
5.1.1	Comparison Functions	19

5.2	Arithmetic Operators	21
5.3	Other Query Operators	21
5.3.1	Examples	22
6	Quality Evaluation Operators	22
6.1	Examples	22
7	Miscellaneous Functions	23
A	Installation	24

1 Summary

The ORION, developed by the database group of Purdue University, is a system that provides querying and data storage utilities for sensor databases. The system focuses on the uncertainty management issues of sensor data values. This document describes the main motivation and concept behind ORION. The system architecture, functionalities as well as their usage will be presented. Installation steps and technical issues interested to developers will also be discussed.

The web-site of ORION (<http://orion.cs.purdue.edu>) contains an overview of the ORION system, downloadable source codes and other useful information. Please feel free to contact the authors in case of any questions.

2 Background

In this section, we will describe briefly the background and concepts behind the ORION system. We present the concept of uncertain data models, probabilistic queries and quality evaluation, which are the essential elements in the ORION system.

2.1 The Data Uncertainty Problem

Due to advances in sensor technologies, systems that acquire information from the physical world have attracted tremendous research interests in recent years. Sensornets [4], for example, acquire temperature, pressure and voltage through a sensor network. Location-based applications allow locations of targets to be constantly monitored and queried [7]. 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 [7, 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 [7, 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) [7].

To evaluate data uncertainty, *probabilistic queries* have been proposed [7, 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.

Recently systems have been proposed to manage data uncertainty. For example, the Trio system [6] attempts to build a general database that supports data accuracy and lineage, while in [4], a probabilistic model-based data acquisition system has been proposed for sensor networks.

Our database system, called ORION (formerly known as U-DBMS in [2]), aims at providing uncertainty management for uncertain sensor values.¹ The ORION is developed on top of PostgreSQL [5], an object-oriented relational open-source database system. The novel features of ORION 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.

Next, we discuss the concept of data uncertainty. We then present various issues of probabilistic queries that operate on this uncertainty model in Section 2.3.

¹According to www.dictionary.com, the Orion is a nebular in the celestial equator near Gemini and Taurus.

2.2 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 $i = 1, \dots, n$). 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 $a.l, a.r \in \mathfrak{R}$, $a.r \geq a.l$ 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 [7]. An example uncertainty pdf is the Gaussian distribution, which models the measurement inaccuracy of location data [7] and data from sensor network [4]. It can also be a uniform distribution, which represents the worst-case uncertainty within a given uncertainty interval. Based on this data uncertainty model, we have developed probabilistic queries that operate on this model, as described next.

2.3 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]. In the classification scheme described next, each query class has a unique definition of answer quality.

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 $l, u \in \mathfrak{R}$ 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 , and $p(x)$ is a pdf of X such that $\int_l^u p(x)dx = 1$.

For more details about the evaluation and quality metrics of each query class, readers are referred to [1].

2.4 Query Closure and Probability Thresholds

In the previous discussions, a probability value is augmented to each data item in the answers to indicate the degree of answer imprecision (e.g., the join operators in the Entity-based Non-Aggregate Class). However, in the definition of a probabilistic query, the input of the query does not have a probability value is augmented to a data item. In order to ensure that a query answer can be passed as an input to the next query operator (thereby achieving the database closure property), ORION provides operators for separating probability values from the data items in the answer. As an example, ORION provides

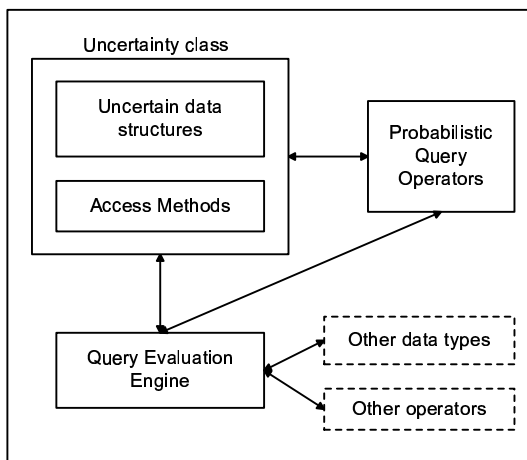


Figure 1: The architecture of ORION

two variants of the equality operator: (1) “ $a = \% b$ ” indicates the probability that the two uncertain data values a and b are equal; (2) “ $a = b$ ” returns a boolean value for whether a is equal to b , with a probability higher than a predefined threshold value. We will describe the detailed semantics of these operators again in subsequent sections. At this point, let us investigate how query classes are supported in ORION.

3 The ORION System

In this section, we describe briefly the architecture of the ORION, which supports various concepts discussed in Section 2.

The ORION is developed on PostgreSQL [5] 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 ORION 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 ORION.

3.1 Supporting Uncertainty Data

We support four types of data uncertainty: (1) Gaussian, (2) uniform, (3) histogram and (4) discrete. While Gaussian and uniform distributions are commonly found in applications, our goal is 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. The “discrete” pdf represents a pdf whose domain is non-continuous. For example, the discrete pdf for a temperature value can be something like: $\{\{10^{\circ}F, 0.4\}, \{11^{\circ}F, 0.5\}, \{12^{\circ}F, 0.1\}\}$.

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 ORION, the user only needs to specify the result is **UNCERTAIN**. The system decides the most appropriate resulting pdf.

3.3 Global Environment

We conclude this section with a brief description of the global environment. Table 1 lists the default global parameter values (their meanings will be explained as appropriate in subsequent sections).

Parameter	Value	Meaning
<code>u_divisions</code>	5	no. of steps used for numerical integration.
<code>resolution</code>	1	resolution of equality operator
<code>p_threshold</code>	0.5	probability threshold value

Table 1: Global Parameter Values.

All of these parameter values can be queried and changed using the `get_u_config` and `set_u_config` functions respectively.

In the rest of this document, we focus on the detailed functionalities provided by ORION. Section 4 describes the structure for each kind of uncertain data model. In Section 5 we present the syntax and semantics of probabilistic query operators. Section 6 then describes operators for inquiring quality of a probabilistic answer.

4 The UNCERTAIN Data Type

In this section, we will discuss how the uncertain data models are supported in ORION. We will describe the associated data structures, input and output functions, as well as functions that allow users to enquire the details of an uncertain value.

4.1 Data Structure of UNCERTAIN data type

As mentioned in Section 3.1, the **UNCERTAIN** data type supports Gaussian, uniform, histogram and discrete pdfs. However, the user only needs to use only the **UNCERTAIN** keyword to declare an uncertain data value. This is implemented by defining the following variable-length data structure, called `Uncertain`:

```
typedef struct {  
    int32 size; /* TOTAL size of this data structure */  
    int type; /* Type of uncertain data - Histogram='h', Gaussian='g', Discrete='d' */  
    char data[1]; /* Pointer to actual data */  
} Uncertain
```

In the above structure, `type` gives us the type of data structure to which data pointer is pointing. the `data` is a pointer that points to the actual uncertain data type, which can be Gaussian, histogram, and discrete. The definition of data array may look a bit strange to experienced C programmers, but this is the standard way of handling variable length data-structures in PostgreSQL. For more information, please refer to PostgreSQL manual.

- **The Gaussian** uncertainty pdf consists of two parameters, mean (μ) and standard deviation (σ) which is represented by the following C construct:

```
typedef struct {  
    float mu; /* Mean */  
    float sigma; /* Standard Deviation */  
} Gaussian;
```

- **The Histogram** uncertainty pdf is defined as follows:

```
typedef struct {  
    float min_value; /* Value at which the sampling begins */  
    float max_value; /* Value at which the sampling ends */  
    float interval; /* size of sampling interval */
```

```

float data[1]; /* The actual histogram data (cdf); exactly
               (max_value-min_value)/interval + 1 values in this array.
               Note that this is a variable length array */
} Histogram;

```

The Histogram is conceptually divided into n segments with equal width (specified by `interval`). Notice that `min_value` and `max_value` are boundaries of the Histogram. The variable-length array `data` stores the i -th cdf value at

```
min_value + i*interval
```

(where $i = 0, \dots, (\text{max_value} - \text{min_value})/\text{interval}$).

Observe that a uniform pdf can be readily represented by this structure, by specifying that `interval` equals to `max_value - min_value`.

- **The Discrete** uncertainty pdf is defined as:

```

typedef struct {
    int32 num_data;
    float4 data[1][2]; /* Data points and their probabilities */
} Discrete;

```

As we can see, `Discrete` is a two dimensional array `data`, which stores the data point and the corresponding pdf for that data point.

Next, let us look at the user interface of ORION for specifying the values for the above data structures.

4.2 Input and Output Functions for Uncertain Data

ORION provides command-line arguments for specifying the parameter values of the `UNCERTAIN` data type. For convenience of explanation, let us assume that a table called `pressure`, consisting of only one attribute v of type `UNCERTAIN`, has been constructed. The format of input parameters for Gaussian, Histogram and Discrete uncertainty pdf are described below:

- **The Gaussian uncertainty pdf**

Format: '(g,mu,sigma)'

- **g**: specifies that the model is Gaussian.
- **mu**: the value of the mean (μ).
- **sigma**: the value of the standard deviation (σ).

Example: insert into pressure values ('g, 0.5, 0.01');

This inserts to the pressure table an uncertain attribute, with a mean of 0.5 and standard deviation of 0.01.

- **The Histogram uncertainty pdf**

Format: '(h, min, max, interval, data_1, data_2, ..., data_n)'

- **h**: specifies that the histogram is used
- **min, max**: the lower and upper bound of the uncertainty interval
- **interval**: width of each of the n segments of the histogram
- **n**: equals to $(\text{max}-\text{min})/\text{interval}$ (Note: if $\text{max}-\text{min}$ is not a multiple of interval , an error flag is raised.)
- **data_i**: the pdf of a segment i (where $i = 0, 1, \dots, n$). We assume that the pdf remains constant within a segment. Also, data_i can be non-normalized – they will be normalized by ORION.

Example: insert into pressure values ('h,1,4,1, 4,8,4');

This inserts to the pressure table a histogram-type uncertain attribute, which has an uncertainty interval [1,4]. Each segment has a width of 1 unit. The probability values at the three segments [1,2], [2,3] and [3,4] are 4, 8 and 4 respectively. After normalization by ORION, the probability values become 0.25, 0.5 and 0.25. Figure 2 shows the plot generated by Orion for the above histogram.

- **The Discrete uncertainty pdf**

Format: '(d, n, point_1, prob_1, point_2, prob_2, ..., point_n, prob_n)

- **d**: specifies that the discrete uncertainty pdf is used.
- **n**: number of data points
- **prob_i**: the probability value at point_i

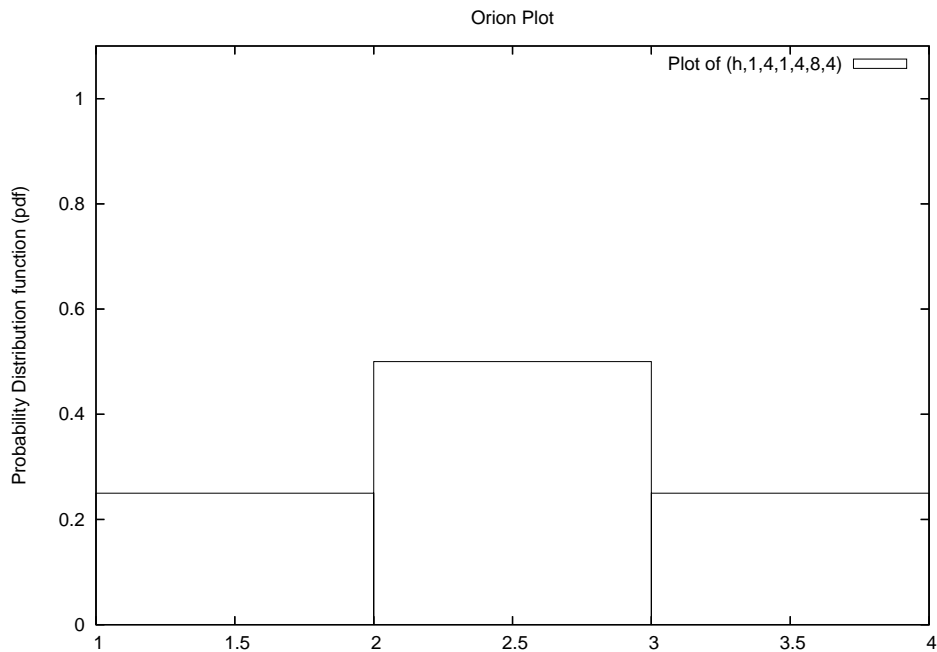


Figure 2: Orion’s plot of '(h,1,4,1,4,8,4)'

Example: insert into pressure values ('(d, 2, 0.1, 0.6, 0.2, 0.4)');

This inserts to the “pressure” table a discrete uncertain attribute. The probability at points 0.1 and 0.2 are 0.6 and 0.4 respectively.

4.2.1 Output Functions

For each of the above data types, the **output function** simply retrieves the values stored in the data structure and composes a null-terminated string, conforming to the format specified above. For example, the following query

```
select v from pressure
```

displays the following string:

```
(d, 2, 0.1, 0.6, 0.2, 0.4)
```

Notice that we have decided not to further format this “raw” display format because the output string may be the input to another query operator. Thus it is important that the output string conforms

to exactly the standard defined for an uncertain data attribute.

4.3 Inquiry Functions

The following functions are provided in ORION to allow users to inquire the parameter values of the uncertain attribute concerned.

- `u_type(uncertain)`: returns the type of the uncertain variable (i.e., Histogram, Gaussian, or Discrete)
- `u_expected(uncertain)`: returns the expected value (i.e., mean) of the uncertain variable.
- `u_2moment(uncertain)`: returns i.e., second moment (mean-square) of the uncertain variable.
- `u_variance(uncertain)`: returns the variance of the uncertain variable.
- `u_cdf(uncertain, real x)`: returns the cdf of the uncertain variable at point x .
- `u_pdf(uncertain, real x)`: returns the pdf of the uncertain variable at point x ; returns pdf for $[x, x + \delta]$ if pdf is not defined at point x .
- `u_prob(uncertain, interval_min, interval_max)`: returns the probability that the uncertain variable lies in the interval `[interval_min, interval_max]`.
- `u_upper(uncertain)`: returns the upper limit of the uncertain variable.
- `u_lower(uncertain)`: returns the lower limit of the uncertain variable.
- `u_interval(uncertain)`: returns the sampling interval of the uncertain variable.

The following functions are specific to Gaussian pdfs. They both provide a mechanism to convert a Gaussian distribution to a Histogram distribution, which can be useful in case query operators that are only defined for Histogram pdfs.

- `u_g2h(uncertain, integer)`: converts an uncertain value with Gaussian pdf to the Histogram pdf, where `integer` is the number of segments for the Histogram pdf so formed.
- `u_g2h(uncertain)`: same as before, except that the global parameter `u_divisions` is used as the default number of segments for the Histogram pdf generated.

4.3.1 Examples

In this section we provide the

- `u_expected(uncertain)`:

```
# select u_expected('(h,1,5,1,1,2,3,100)::uncertain);  
u_expected  
-----  
4.40566  
(1 row)
```
- `u_pdf(uncertain, real x)`:

```
# select u_pdf('(h,1,3,1,1,2)::uncertain, 2);  
u_pdf  
-----  
0.666667  
(1 row)
```
- `u_prob(uncertain, interval_min, interval_max)`:

```
# select u_prob('(g,0,1)::uncertain,-2,2);  
u_prob  
-----  
0.959452  
(1 row)
```
- `u_upper (uncertain)`:

```
# create table temp (a uncertain);  
CREATE TABLE  
# insert into temp values ('(h,1,2,0.5, 3,4)');  
INSERT 256727 1  
# insert into temp values ('(g,3,1)');  
INSERT 256728 1
```

```
# select u_lower(a) from temp;
u_lower
-----
1
-Infinity
(2 rows)
```

- `u_g2h(uncertain, integer)` and `u_g2h(uncertain)`:

In these examples, we are using the temp table that we created in the above example.

```
# select u_g2h(a,10) from temp;
u_g2h
-----
(h, 1.00, 2.00, 0.50, 0.86, 1.14)
(h, 0.00, 6.00, 0.60, 0.01, 0.04, 0.13, 0.27, 0.38, 0.38, 0.27, 0.13, 0.04, 0.01)
(2 rows)
```

```
# select get_u_config('u_divisions');
get_u_config
-----
5
(1 row)
```

The following query uses the default value from the global environment.

```
# select u_g2h(a) from temp;
u_g2h
-----
(h, 1.00, 2.00, 0.50, 0.86, 1.14)
(h, 0.00, 6.00, 1.20, 0.02, 0.19, 0.40, 0.19, 0.02)
(2 rows)
```

5 Probabilistic Query Operators

In this section we present the querying facilities of ORION. These query operators implement the representative queries of the query classification scheme in Section 2.3. New query operators will be added in the future, and this section will be updated accordingly. We will discuss fundamental comparison operators in Section 5.1. Then, arithmetic operators (Section 5.2) and other common query operators (Section 5.3) will be presented.

5.1 Comparison Operators

In ORION, binary comparison operators (such as equality and greater than) are provided for comparing values. They can be used for database join as well. A full discussion of the definitions and evaluation of these operators are described in [3]. For user's convenience, default parameter values (e.g., the value of probability threshold) are used for evaluating these operators. Let a be a value of `UNCERTAIN` type, and v be a real-valued constant. Two variants of operators are available, which return a boolean value and probability, respectively:

- **Operators that return a boolean value (i.e., true or false) for the comparison.** The following operators can be used in a SQL statement. Notice that the operators `>=` and `<=` are not defined since the probability that the two values are exactly equal is zero.

- $a = v, v = a$
- $a = v, v = a$ (`=` can be replaced by `<>`)
- $a < v, v < a$
- $a > v, v > a$
- $a == a$ (`==` can only be used for uncertainty pdfs with the same kind of distribution)

Example: Assume the default probability threshold value is 0.7, then the following statement:

```
SELECT '(h,1,4,1,1,2,1)>::uncertain = 2.5;
```

will return `FALSE`, since the uncertain value has a probability of 0.5 being equal to 2.5.

- **Operators that return the actual probabilities for the comparison to be true.** The following operators are defined for use in a SQL statement.

- $a = \%v, v = \%a$
- $a = \%v, v = \%a$
- $a < \%v, v < \%a$
- $a > \%v, v > \%a$

Example: The following statement:

```
SELECT '(h,1,4,1,1,2,1)>::uncertain =% 2.5;
```

will return 0.5 – the probability that the specified uncertain value equals to 2.5.

5.1.1 Comparison Functions

For the operators that we have just introduced, they are actually operators which are “overloaded” with comparison functions defined for uncertain data. Here a complete list of all the variants of comparison functions are described, which provide the user the maximum flexibility in choosing the most suitable ones.

- **Equality (=):**

- `u_eq(uncertain, real, resolution)`: returns the probability that the given uncertain value is equal to the real value (with a given resolution, or accuracy, as defined in [3]).
- `u_eq(uncertain, real), u_eq(real, uncertain)`: returns the probability using a global value of resolution.
- `u_eq_const_bool(uncertain, real), u_eq_const_bool(real, uncertain)`: returns a boolean value (true/false) using global values for resolution and probability threshold parameters.
- `u_eq(uncertain, uncertain, resolution, divisions)`: returns a boolean value (true/false) for comparing the equality of two uncertain variables using a given resolution and division value.
- `u_eq(uncertain, uncertain, resolution), u_eq(uncertain, uncertain, divisions), u_eq(uncertain, uncertain), u_eq_bool(uncertain, uncertain)`: similar as above; the omitted parameters (`resolution` and `divisions`) are read from the global environments.
- `u_identical(uncertain, uncertain)`: returns true if the two uncertain values are identical (i.e. the type of the distribution and all parameters of the distribution are the same for both uncertain variables). This function overloads the symbol `==`.

- **Inequality (!=):**

- `u_neq(uncertain, real, resolution)`: returns the probability that the uncertain variable is not equal to the real value, with a given resolution.
- `u_neq(uncertain, real), u_neq(real, uncertain)`: returns the probability using the global value of resolution.
- `u_neq_const_bool(uncertain, real), u_neq_const_bool(real, uncertain)`: returns the boolean value (true/false) using global values for resolution and probability threshold
- `u_neq(uncertain, uncertain, resolution, divisions)` returns a boolean value (true/false) for comparing the inequality of two uncertain variables using a given resolution and division value.
- `u_neq(uncertain, uncertain, resolution), u_neq(uncertain, uncertain, divisions), u_neq(uncertain, uncertain), u_neq_bool(uncertain, uncertain)`: similar as above; the omitted parameters (`resolution` and `divisions`) are read from the global environments.

- **Smaller Than (<):**

- `u_less(uncertain, real)`: returns the probability that the uncertain variable is less than the real value.
- `u_less(real, uncertain)`: returns the probability that the real value is less than the uncertain variable.
- `u_less_const_bool(uncertain, real), u_less_const_bool(real, uncertain)`: similar to the above two functions, except that the boolean value of the comparison is returned (i.e., a true/false value based on whether the comparison evaluates a probability higher than the probability threshold value).

- **Greater Than (>):**

- `u_greater(uncertain, real)`: returns the probability that the uncertain variable is greater than the real value.
- `u_greater(real, uncertain)`: returns the probability that the real value is greater than the uncertain variable.
- `u_greater_const_bool(uncertain, real), u_greater_const_bool(real, uncertain)`: similar to the above two functions, except that the boolean value of the comparison is returned

(i.e., a true/false value based on whether the comparison evaluates a probability higher than the probability threshold value).

5.2 Arithmetic Operators

The following lists the binary arithmetic operators provided:

- `u_add(uncertain, uncertain, integer)`: Adds two uncertain values. The third parameter represents number of divisions used for numeric integration.
- `u_add(uncertain, uncertain)` and the overloaded operator '+': Uses the global parameter 'u_divisions' for numeric integration.

5.3 Other Query Operators

In this section we list query operators that are representatives for each query class of the query classification scheme.

- `u_set(uncertain)`: Aggregate function which returns a set of uncertain values. For example, if we have a table containing an uncertain attribute *a*, then the query:

```
SELECT u_set(a) FROM tablename
```

will return a set that contains all the values of attribute *a* present in the table. This function is useful for creating sets and passing them to functions like `u_range` and `u_Emin`.
- `u_range(lower, upper, uncertain_set, threshold)` and `u_range_all(lower, upper, uncertain_set)`: These functions perform the range queries. `u_range` function returns all the uncertain values in the `uncertain_set` that are between `lower` and `upper` with a probability greater than `threshold`. The last parameter for `u_range` is optional. If it is not provided, the system will use the global probability threshold (`p_threshold`). `u_range_all` is similar to `u_range` with the threshold set to 0. (The `uncertain_set` can be constructed using the `u_set` function. Note that these queries can be written without constructing any sets by using the `u_prob` function)
- `u_Vmin(uncertain)`: This is the aggregate function for the value minimum query . For example, if we have a table containing an uncertain attribute *a*, then the query:

```
SELECT u_Vmin(a) FROM tablename
```

will return an uncertain value which is the minimum value of attribute a . Orion also provides non-aggregate form of value minimum query using sets: `u_Vmin(uncertain_set)`.

- `u_ENNQ(uncertain_set, x, divisions)`: Entity Nearest Neighbor Query, which return a set of objects being the nearest neighbor of x along with the probabilities. The last parameter is optional. If it is not provided, the system will use the global value for divisions (`u_divisions`).
- `u_Emin(uncertain_set)`: This is the Entity based minimum query. It returns the values from `uncertain_set` along with the probability of that value being the minimum.

Please refer to [1] for more information about the semantics of `u_Vmin`, `u_Emin` and `u_ENNQ`.

5.3.1 Examples

Coming soon...

6 Quality Evaluation Operators

As mentioned in Section 2.3, quality metrics are defined in [1] to indicate the degree of imprecision of a probabilistic answer due to data uncertainty. We now describe operators in ORION that provide such functionalities. These operators compute a value between $[0, 1]$, called the *quality score*, which gives a higher value if the query answer is less ambiguous.

- `u_qual(uncertain)`: Returns the “quality” of an uncertain value. If the uncertain value has a larger variance, the quality is lower. It can be used to measure the quality of a value-based query.
- `u_qual_NA`: Returns the quality of entity based non-aggregate query results.
- `u_qual_A`: Returns the quality of entity based aggregate query results.

6.1 Examples

Coming soon...

7 Miscellaneous Functions

This section describes other supporting functions provided in ORION.

The following two functions modify and retrieve global parameters (e.g., resolution, and probability threshold), defined in Table 1.

- `set_u_config(property, value)`: `property` is the name of a global parameter, for instance, "resolution".
- `get_u_config(property)`: retrieve the value of a given `property`, i.e., the name of a global parameter.

References

- [1] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.
- [2] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *Very Large Data Bases (VLDB)*, 2005.
- [3] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient join processing over uncertain data. Technical Report CSD TR# 05-004, Dept. of CS, Purdue University, 2005.
- [4] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *CIDR, 2nd Biennial Conf. on Innovative Data Systems Research*, 2005.
- [5] PostgreSQL Global Development Group PostgreSQL 8.0, 2005.
- [6] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR, 2nd Biennial Conf. on Innovative Data Systems Research*, 2005.
- [7] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3), 1999.

A Installation

The Orion system is packaged with the GNU tools: autoconf, automake and libtool. Please refer to `INSTALL` and `README` file in the Orion distribution for more information about installing Orion.

1. Install PostgreSQL and make sure you can connect to the server without any problems.
2. Run the orion instalation from the machine on which postgres server is running.
3. Make sure `psql` command works without any arguments (You may have to set `PGDATABASE` environment variable. For more information refer to PostgreSQL documentation)
4. Set `CPATH/C_INCLUDE_PATH` environment variables to make sure `gcc` can find your PostgreSQL include files (The files inside `postgres/include/server`).
5. Follow the instructions provided in `INSTALL` file to complete the installation.

Please contact sarvjeet@purdue.edu if you are having any problems during installation.