5-18-2007

Evaluation of Expressions with Uncertainty in Databases

Moginraj Mohandas
University of New Orleans

Evaluation of Expressions with Uncertainty in Databases

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Masters of Science
in
The Department of Computer Science

by

Moginraj Mohandas

B.Tech, Lal Bahadur Shastri College of Engineering, India, 2003

May, 2007

Acknowledgement

I wish to express my gratitude to a number of people who became involved with this thesis, one way or another.

I am deeply indebted to my supervisor, Dr. Nauman Chaudhry whose help, stimulating discussions; suggestions and encouragement helped me in all the time of research and for writing of this thesis. In addition, I would like to thank Dr. Mahdi Abdelguerfi and Dr. Golden Richard III for being on my thesis defense committee.

Finally, I would like to thank my family and friends for the continuous support and help they provided to me. I would like to dedicate this thesis to my parents.

Table of Contents

List of Tables v
List of Figures…………………………………………………………………………vi
Abstract……………………………………………………………………………..vii
1 Introduction…………………………………………………………………..1
2 Expression Syntax and Terminology……………………………………………...5
2.1 Syntax of Crisp Expressions……..……………………………….....5
2.2 Syntax for Modeling Uncertainty in Expressions and Data Items……………………..9
3 Related Work……………………………………………………………………14
3.1 Previous Work on Expressions: The Oracle Method…………….14
3.1.1 How Oracle stores Expressions………………….14
3.1.2 Evaluation of Expressions(EVALUATE operator)…………………………………………15
3.1.3 Expression Data Type and Handling: Oracle Approach………………………………………...16
3.2 Uncertainty Handling by different Systems……………………..17
3.2.1 Automated Ranking……………………………...17
3.2.2 Uncertainty in Publish/Subscribe Systems………21
3.2.3 Trio(ULDB)……………………………………..23
4 Semantics of Matching Uncertain Expressions and Uncertain Data……………………...…………………………………………….....25
4.1 Evaluating Crisp Expressions against Crisp Data Items…………25
4.1.1 Semantics of Matching at the Predicate Level…………………………………….…………25
4.1.2 Formal Model for Evaluating Crisp Expressions against Crisp Data Items…………………………27
4.1.2.1 Variable to describe Evaluation cases……………………………...27
4.1.2.2 Categorizing Cases of Crisp Expression-Data Item Evaluation..28
4.2 Semantics of Matching Uncertain Predicates Against Uncertain Data Terms………………………………………………………29
4.3 Evaluation Semantics for Expressions and Data Items with Approximate Matching and Uncertainty……………….……..38
4.3.1 Crisp Data Items, Crisp Expressions…….……...38
4.3.2 Uncertain Data Items, Uncertain Expressions…...42
5 Supporting Uncertain Expressions in a DBMS…………………………………46
5.1 Expression Data Type……………………………………………46
5.2 Details of Implementation………………………………………46
6 Conclusion and Future Work…………………………………………………....51
References…………………………………………………………………………..53
Appendix.…………………………………………………………………………...55
Vita…………………………………………………………………………………58

List of Tables

Table 2.1 Example of an Expression Table……………………………………………….7, 8

Table 2.2 Example of a Conceptual Table with Data Items as column values……………9

Table 2.3 Example of Table with Expressions(with Uncertainty)…………………...11, 12

Table 3.1 Consumer Table…………………………………………………………...14, 15

Table 3.2 Example Table for ULDB…………………………………………………..24

Table 3.3 Example Table for ULDB…………………………………………………..24

Table 4.1 Example Table to show the various categorizing cases…………………...28, 29

Table 4.2 Comparison Table for Evaluation(without Uncertainty)………………….38, 39

Table 4.3 Comparison Table for Evaluation(with Uncertainty)……………………..42, 43

List of Figures

Figure 5.1 Screenshot of Insert and Select in Postgresql……………………………….48

Figure 5.2 Screenshot of Evaluation of Expressions against Data Items using Evaluate operator………………………………………………………………………………50

Abstract

Expressions are used in a range of applications like Publish/Subscribe, Ecommerce, etc. Integrating support for expressions in a database management system (DBMS) provides an efficient and scalable platform for applications that use Expressions. Support from uncertain data and expressions can be beneficial but not currently provided for.

In this thesis, we investigate how expressions with uncertainty can be integrated in a DBMS like other data. We describe the underlying theory and implementation of UNXS (**UN**certain e**X**pression **S**ystem), a system that we have developed to handle uncertainty in expressions and data. We develop a theoretical model to compare and contrast different previous work in supporting uncertainty in DBMS and Publish/Subscribe systems. We extend the existing approaches to propose new techniques for matching uncertain expressions to uncertain data in UNXS. We then describe an implementation that integrates this support in Postgresql DBMS, which to our knowledge is the first such implementation.

.

Chapter 1 Introduction

Expressions are a good way to model the interests of a user in expected data. Expressions have been used in various application domains, such as, Publish/Subscribe [2, 13], Ecommerce [11], Website Personalization [12], etc. In such application domains, users want to specify their interest in expected data in terms of expressions defined on this data. The application needs to persistently maintain these expressions and match data with these expressions to inform users of items of their interest. For example, in a Publish/Subscribe system [13, 2] for a “Real Estate” application, subscriptions correspond to the interests of the user defined over various attributes of houses. These subscriptions are matched against publications of houses being sold that are published by an information provider.

Example:

Subscription: Notify me of houses with price less than forty thousand that have a garage and have more than two bedrooms.
This subscription can formally be modeled as the following Expression:

price<40000 AND garage=yes AND bedrooms>2

This expression, as well as expressions defined by other users, would be matched with Publications that are given by the Real Estate Agents, who are the information providers.

Many times users find it easier to express their interest in uncertain or vague terms, rather than in precise terms [2], [9]. The need to process uncertain data [7] has also become very prominent in database management system research, especially in recent years because of the wide varieties of sources that data come from [13]. However, currently most database management systems as well as publish/subscribe systems do not support uncertain data. Such data models, that do not support uncertain data, are called *crisp data models*. But, lets look at an example from the application “Real Estate”. A user may be certain that he can pay up to $40,000 for a house and the house must have more than 3 bedrooms. Additionally, even though he would like to have the house to have a garden, but he is willing to consider houses without a garden. Likewise, he would like the house to have a garage, but gives greater priority to a garage than a garden. In case of publications too, a Real Estate Agent might not be absolutely certain as to whether a particular house has a garden. Such Uncertainties may occur because the information may not be available in its entirety at a given point of time. This can happen in a variety of application domains as the information may be gathered from multiple sources, some of which may have more complete information than others. We can modify the previous example to illustrate the need for modeling uncertainty.

Example:

Subscription : Notify me of houses with price less than forty thousand that have two bedrooms. In addition, it would be good if these houses had a garage, even though it is not a required criterion.

This subscription can be formally modeled via the following Expression that supports Uncertainty:

price<40000 1.0 AND garage=yes 0.3 AND bedrooms>2 1.0

Notice that we have attached Confidence values with individual conditions in the expression. The conditions on price and bedrooms have a Confidence value of 1.0 that expresses the user's requirement that any house of interest to them must meet these conditions. The condition on garage has a Confidence of 0.3 and this means that meeting this condition has less priority than the other conditions. Therefore, even if an Agent is not completely certain whether a particular house has a garage, that house should be considered for this subscription. However, matching Uncertain data with Uncertain expressions requires that we have an appropriate theoretical model for determining the match. We will develop such a theoretical model in this thesis.

It has been noted that the ability to manage Expressions in a database management system (DBMS) would be very useful [1]. Management of Expressions in a DBMS would facilitate efficient handling of Expressions by using techniques such as query optimization and indexing to efficiently handle large number of Expressions. A notable work in this area is [1], where ability to store Expressions as data in tables in Oracle 10g is described. Ability to store Expressions under a column of a table is provided by extending an existing data type to support Expressions. To evaluate these Expressions over data, a new EVALUATE operator is introduced. For efficient handling of expressions, special indexing structures are proposed. With these extensions, the DBMS can handle such Expressions, which denote user interests, in addition to all the other data related to a user. Incorporation of support for Expression data type in the DBMS facilitates the management of large number of Expressions in an efficient and scalable manner. Oracle though does not support Uncertainty in either Expressions or data.

In this thesis, we look at how Expressions with Uncertainty can be incorporated in to a Database so that they can be processed like other data. We describe the underlying theory and implementation of UNXS (**UN**certain e**X**pression **S**ystem), a system that we have developed to handle Uncertainty in Expressions and data. We first develop a theoretical model to compare and contrast support for Uncertainty that has previously been developed by different researchers. We extend the existing approaches to propose new techniques for matching Uncertain Expressions to Uncertain data in UNXS. We then describe an implementation that integrates this support in a Database via a new data type called Expression and its associated operator called EVALUATE. The Database that we have used is Postgresql 8.0.7. Postgres (as it is widely called) is an open-source database which provides support for Extendibility as explained in [8]. This means that it is possible for developers to extend the database functionality by adding new functions, user-defined data types, new operators to support such data types etc. To the best of our knowledge, this is the first integration of Uncertain data support within a DBMS as opposed to adding a layer outside the DBMS to translate to and from a *crisp* DBMS.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of the terminology used in the thesis and the formal **syntax** for modeling Expressions as well as for Uncertainty in Expressions. Chapter 3 discusses related work. Chapter 4 introduces a theoretical model for describing the **semantics** of matching Uncertain Expressions and Uncertain data. We use this theoretical model to compare various existing techniques to handle Uncertainty and to explain how we have handled Uncertainty in Expressions in UNXS. Chapter 5 describes the implementation carried out in this thesis. Chapter 6 gives conclusion and future work.

Chapter 2 Expression Syntax and Terminology

In this chapter, we give an overview of terminology used to describe Expressions as well as the formal syntax used to model Expressions, both crisp (i.e., without Uncertainty) and Uncertain. In Section 2.1, we describe the syntax of crisp Expressions as given in [1]. In Section 2.2, we give the extended formal syntax we have developed to model Uncertainty in Expressions and data.

2.1 Syntax of Crisp Expressions

To describe crisp Expressions, we use the syntax developed in [1]. An Expression is defined as a group of predicates that are typically joined by an AND or OR. In this work, we consider only those Expressions that have their Predicates joined with AND. A predicate contains a left hand side (a column name), an operator and a right hand side (value of the column).

The formal syntax of an Expression in BNF notation is given below:

Expression ::= Predicate | Predicate AND Expression

Predicate ::= Identifier Operator Constant

Operator ::= < | > | = | <= | >=

Example:

Here are some example predicates :

Pred1: price > 40000

Pred2: bedrooms = 3

An example expression is:

Exp1: price>40000 AND bedrooms=3

We introduce '**Data Item**' which corresponds to a Publication and can be thought of as a row in a Database where the column names are given along with the values. Each subpart of a Data Item is called a Data Term and each Data Term corresponds to an attribute and its value. Expressions are evaluated against Data Items, which corresponds to Subscriptions being evaluated against Publications.

The formal syntax of a Data Item in BNF notation is given below:

Data Item ::= Data Term | Data Term AND Data Item

Data Term ::= Identifier Operator Constant

Operator ::= =

Note: The only operator allowed in a Data Item is the assignment operator since a Data Item represents actual information.

Example:

An example Data Item is:

price=50000 AND bedrooms=4

An Expression evaluates to *True* if the incoming data-item meets the user's interest, and if the incoming data-item doesn't meet the user's interest then the Expression evaluates to *False*. We will discuss the semantics of such evaluation in Chapter 4 of this thesis.

The example application that we work with is called the '**Real Estate**' application. In a typical Real Estate scenario, there are many users or buyers interested in different types of houses. These buyers have their own specifications on what type of House they want. This is modeled by a Subscription in the case of a Publish/Subscribe. The Real Estate agents have information about the different houses that are put for sale. This information is modeled as the Publications in the Publish/Subscribe system. In our case, each

Expression can be thought of as similar to a Subscription, and each Data Item as similar to a Publication.

Let us look at a sample table called House and an Expression column stored in that table:

HOUSE

Table 2.1 Example of an Expression Table

User ID	Name	Street Name	Interest
1	John Travolta	1st street	price>40000 AND bedrooms=4 AND outpaint=red
2	Denzel Washington	2nd street	price<50000 AND bedrooms>=2 AND restrooms=2 AND outpaint=blue
3	Adam Sandler	3rd street	price=50000 AND garage=yes AND bedrooms=3 AND outpaint=green
4	Jack Nicholson	4th street	garage=yes AND bedrooms=3 AND outpaint=green
5	Russell Crowe	5th street	price<30000 AND garage=yes AND bedrooms=3 AND restrooms>1
6	Bruce Willis	6th street	garage=yes AND bedrooms=3 AND outpaint=green AND floors=2
7	Tom Hanks	7th street	floors=2 AND garage=yes AND bedrooms=3 AND outpaint=yellow
8	Tom Cruise	8th street	price<60000 AND garage=no AND bedrooms=3 AND area=1500

Table 2.1, continued

9	Nicholas Cage	9th street	bedrooms>=2 AND outpaint=blue AND floors=1 AND area=1200
10	Will Smith	10th street	price=50000 AND garage=no AND bedrooms=3 AND floors=2 AND area=2400 AND garden=yes

Example:

Some examples Data Items that can be evaluated against the Expressions in the Interest column of the above table are given below:

Data Item1: price=42000 AND bedrooms=3 AND outpaint=blue
Data Item2: price=40000 AND bedrooms=2 AND floors=1 AND garage=yes
Data Item3: bedrooms=3 AND garage=no AND price=48000 AND floors=1

The below given SQL query evaluates the Expressions in the table against Data Item1:

*Select * from House where (EVALUATE(price=42000 AND bedrooms=3 AND outpaint=blue))=1;*

With the user subscriptions modeled as expressions, each new data item, i.e., a new publication, is matched against all the expressions to find expressions which evaluate to true. Note that this model of matching one data item against many expressions inverts the traditional DBMS model in which one query gets evaluated against many rows. When using the expression data type, each data item (which corresponds to a row in traditional DBMS applications) is evaluated against many expressions (each of which corresponds to a query in traditional DBMS applications). We refer to this as **inverted data-query paradigm.** We explain this concept with the following example.

For the Data Items mentioned in the query above, a conceptual table is given below. Each Expression can be thought of as a query against this table

Table 2.2 Example of a Conceptual Table with Data Items as column values

price	bedrooms	Outpaint	garage	floors
42000	3	Blue	NULL	NULL
40000	2	NULL	yes	1
48000	3	NULL	no	1

2.2 Syntax for Modeling Uncertainty in Expressions and Data Items

As explained in the Introduction, there is a need to model Uncertainty in Expressions whenever the user wishes to express interest in data via conditions that have different priorities. Similarly, if the data provider is uncertain about some attributes of the data, there is a need to model Uncertainty in Data Items. In UNXS, uncertainty in Expressions and Data Items is syntactically modeled via Confidence values for each predicate of an Expression, and each Data Term of a Data Item.

We extend the BNF notation for Expressions and Data Items to incorporate the Uncertainty model of UNXS :

Predicate ::= Identifier Operator Constant : Confidence
Operator ::= < | > | = | <= | >=

Data Term ::= Identifier Operator Constant : Confidence

Operator ::= =

Here, *Confidence* values are in the interval (0,1] including 1 but excluding 0. A 1 corresponds to a value about which the user is absolutely certain. The default Confidence is always 1. Note that crisp Expressions correspond to the case where all Predicates have a Confidence value of 1. Similarly, crisp Data Items correspond to a Data Item where all Data Terms have Confidence value of 1.

We also note here that the Confidence values attached to Expressions and Data Items need not be given explicitly by users or agents respectively. An application may provide a Graphical User Interface (GUI) to users to specify their interests and relative priority for the conditions in those interests. E.g., a user may specify that he would like a garage and a garden, but considers a garage to be more important. Similarly a GUI may be available for information providers to specify data in vague terms. E.g., a provider may specify that the price is *low* or *high*. The GUI can then convert this vague information into actual Confidences for the Expressions and Data Items that are used in the DBMS. The focus of our thesis is in the processing of Uncertainty where the DBMS is already provided the Confidences. The development of an appropriate GUI and API is thus outside the scope of this thesis.

Now let us go back to the "**Real Estate**" application and try to understand what Confidences could mean in that scenario. The application "**Real Estate**" has customers who want a particular type of house. The Customers express their preferences using Expressions with each predicate denoting a property of the house desired by a user. The user or customer may also give "Confidences" to the predicates that he specifies. This could mean that the user is unsure about whether he wants some of these properties, so he specifies a priority for them. So, if a user specifies that he wants a garage with Confidence 0.5, it means he is looking for houses that have a 50% chance of having a garage. The Real Estate agent will give the details of a house with its properties and this is denoted by the Data Item. Now, the Uncertainty that could occur is a situation where the Agent is not 100% sure whether the properties of a house are true or not. He is only partially sure about some of the information, which could happen because the information provided to him was partial at the particular point of time. In this case, he gives each of the properties of the House a "Confidence", which denotes the confidence with which he is sure about the information. The exact semantics of how an Expression with Uncertainty should be matched against a Data Item with Uncertainty is an issue that we discuss in Chapter 4, Section 4.2.

Example:

An example of Predicate and Data Term with Confidence:

Predicate : bedrooms>4 : 0.5

Data Term : bedrooms=6 : 0.1

Example:

Let us also look at a complete example of an Expression and a Data Item with Confidences.

Data Item: price=50000 : 0.6 AND bedrooms=3 : 0.4 AND outpaint=red : 0.4 AND garage=yes : 0.8

Expression: price>40000 : 0.5 AND bedrooms=2 : 0.6 AND outpaint=red 0.3 AND garage=no : 0.6

Here for the first predicate of the Expression, the user is saying he wants a House which costs more than 40000 and that he wants only matches that have at least a 50% surety that the price is greater than 40000. The Data Term corresponding to price has price = 50000, and the Agent (information provider) is 60% sure that this is true.

Now, lets look at the table *HOUSE* again with the interest column now containing Expressions with Confidences attached to them.

HOUSE

Table 2.3 Example of Table with Expressions(with Uncertainty)

User ID	Name	Street Name	Interest
1	John Travolta	1st street	price>40000 : 0.4 AND bedrooms=4 : 0.5 AND outpaint=red : 0.7
2	Denzel Washington	2nd street	price<50000 : 0.5 AND bedrooms>=2 : 0.3 AND restrooms=2 : 0.8 AND outpaint=blue : 0.6

Table 2.3, continued

3	Adam Sandler	3rd street	price=50000 : 0.6 AND garage=yes : 1.0 AND bedrooms=3 : 1.0 AND outpaint=green : 0.8
4	Jack Nicholson	4th street	garage=yes : 1.0 AND bedrooms=3 : 0.6 AND outpaint=green : 0.7
5	Russell Crowe	5th street	price<30000 : 0.5 AND garage=yes : 0.6 AND bedrooms=3 : 1.0 AND restrooms>1 : 1.0
6	Bruce Willis	6th street	garage=yes : 1.0 AND bedrooms=3 : 0.8 AND outpaint=green : 0.5 AND floors=2 : 0.3
7	Tom Hanks	7th street	floors=2 : 0.7 AND garage=yes : 1.0 AND bedrooms=3 : 1.0 AND outpaint=yellow : 1.0
8	Tom Cruise	8th street	price<60000 : 0.6 AND garage=no : 0.2 AND bedrooms=3 : 0.5 AND area=1500 : 1.0
9	Nicholas Cage	9th street	bedrooms>=2 : 1.0 AND outpaint=blue : 1.0 AND floors=1 : 0.9 AND area=1200 : 0.8
10	Will Smith	10th street	price=50000 : 1.0 AND garage=no : 0.7 AND bedrooms=3 : 1.0 AND floors=2 : 1.0 AND area=2400 : 0.8 AND garden=yes : 0.5

Example:

Here are a few example Data Items with Confidences that can be evaluated against the Expressions in the interest column of *HOUSE* :

Data Item1: price=42000 : 0.4 AND bedrooms=3 : 0.6 AND outpaint=blue : 0.7

Data Item2: price=40000 : 0.6 AND bedrooms=2 : 1.0 AND floors=1 : 1.0 AND garage=yes : 1.0

Data Item3: bedrooms=3 : 1.0 AND garage=no : 1.0 AND price=48000 : 0.6 AND floors=1 : 0.7

The following query shows how the Expressions in the table are evaluated against Data Item1.

*Select * from House where (EVALUATE(price=42000 : 0.4 AND bedrooms=3 : 0.6 AND outpaint=blue : 0.7))=1;*

Execution of this query should return those Expressions that satisfy the evaluation against the given Data Item.

Chapter 3 Related Work

In this chapter, we initially look at the work done by the Oracle team [1] on Expressions and its handling inside a Database. In the later subsections we look at some of the significant methods employed in handling Uncertainty in Databases.

3.1 Previous work on Expressions: The Oracle Method

This section describes one of the most significant work done on Expressions in the recent past. The work [1] was done by a team from Oracle and it mainly involved the incorporation of Expressions in the Oracle Database framework. The work focused on storing an Expression in a column of a table. The work also involved the creation of an EVALUATE operator to Evaluate Expressions given a Data Item.

3.1.1 How Oracle stores Expressions

The Oracle approach enables capability to store Expressions in a column of a Database table. Let us look at an example table from [1] which would store Expressions.

CONSUMER Table

Table 3.1 Consumer Table

CID	Zip Code	Price	Interest(Expression Column)
1	70122	2000	Model = ‘Taurus’ and Price <15000 and Mileage < 2500
2	70123	3000	Model = ‘Mustang’ and Year > 1999 and Price < 20000

Table 3.1, continued

3	70144	5000	Price < 20000 and Color = ‘Red’
4	70323	17000	Price > 15000 and Model =Toyota

All the Expressions related to the Consumer table are stored in the Interest column of the table. The individual attributes in the Expressions may or may not be column attributes of the table.

3.1.2 Evaluation of Expressions (EVALUATE Operator):

In order to evaluate an Expression against a Data Item an EVALUATE Operator is used. The EVALUATE operator would be evaluating to 1 if an Expression is evaluated to True, 0 if it is Evaluated to False. An EVALUATE operator takes two arguments: the column of the conditional Expression and the data items for the Expression. Lets look at an example.

Example:

```
Select CID From Consumer WHERE EVALUATE (consumer. interest,
‘ Model => ‘ ‘ Mustang ‘ ‘, Price => 22000,
Mileage => 1800,Year => 2000 ‘) = 1;
```

This query should return the CIDs of those rows whose Interest column satisfies the given Data Item in the query.

3.1.3 Expression Data Type and Handling : Oracle Approach

Each Expression is evaluated using an EVALUATE operator against incoming Data Items which are specified in SQL WHERE clauses. The Data Items are Expression like structures which are specified by the user. For an incoming Data Item, every Expression is evaluated to either True or False. If the Expression and the Data Item have predicates on the same attributes and if the Predicates lie in the same data ranges, then the Expression evaluates to True, else False. Expressions that evaluate to True return 1 and the rest return 0.

As the Expressions are stored as a regular column in the user table they can be also inserted, updated or deleted with the help of standard SQL statements. All the Expressions stored in a user table share a common set of attributes. All the Expressions under one user table, that share a common set of attributes form an **Expression set**. This set of attributes plus any functions that are used in the **Expressions** comprise the metadata for all the Expressions in a particular user table. This metadata is referred to as the *Expression Set Metadata*. This consists of the elementary attribute names and their data types and any functions used in the **Expressions**. In order to insert a new Expression or to modify an existing Expression validation is required. Validation would be crosschecking if the new Expression or the modifications to the existing Expression comply with the Expression metadata. So all the Expressions present under an Expression set are bound to use the attributes and functions defined under the attribute set.

Oracle has created an Expression data type which is essentially a CLOB or VARCHAR data type used to hold the conditional Expression. A native Expression data type was not built, so the data type is just like any other CLOB or VARCHAR value. The association of the corresponding Expression Set Metadata is achieved by defining a special *Expression constraint* on the column storing expressions. This constraint enforces the validity of the expressions stored in the column as well as provides the necessary metadata for expression evaluation.

3.2 Uncertainty Handling by different Systems

Uncertainty in DBMS is a topic that has long been of interest. There has been lots of research done in this area for example [2], [3], [4], [6], [9] etc. In this subsection, we discuss some of the research that has looked at the different types of Uncertainties in DBMS and how they have come up with solutions to support Uncertainty. In Chapter 4, we present a framework to compare and contrast these methods with each other and with the UNXS system developed by us.

3.2.1 Automated Ranking

Ranking in Databases is a widely researched area and it is used mainly to prioritize results. Ranking can be very useful in two situations. The first situation occurs when a query is too selective and returns no answers. This is termed the "Empty Answers" problem. Ranking techniques can be employed for the "Empty Answers" problem to return a ranked list of tuples that approximately match the conditions specified in the query. The second situation in which result ranking is useful is when a query is very unselective. In this situation the query returns too many results, most of which may not be of interest to the user. This is termed the "Many Answers" problem. In such cases a ranking of results assigns some priority to tuples and the best matching tuples are returned. Since result ranking is done using approximate matching between data and queries, these techniques are worth investigating in the context of Uncertainty.

Research undertaken at Microsoft Research for Automated Ranking of query results in a database is described in [4] The paper deals mainly with the "Empty Answers" problem although it also discusses the "Many Answers" problem. The paper takes motivation from Information Retrieval (IR) solutions to rank results. Two ranking or *similarity functions* are proposed in the paper. These two functions are the IDF (Inverse Document Frequency) similarity function and the QF (Query Frequency) similarity function. Based on the results from these functions, the ranking of results of a query is done.

IDF has been used in IR to suggest that commonly occurring words convey less information about user's need than rarely occurring words. So, for every value t in the domain of an attribute A_k the definition of $IDF_k(t)$ is $\log(n/F_k(t))$, where n is the number of tuples in the database and $F_k(t)$ is the frequency of tuples in the database where A_k=t. For any pair of values u and v in A_k's domain, $S_k(u,v)$ is defined as the $IDF_k(u)$ if u=v, and 0 otherwise. The sum of the IDF values over each of the attributes is used to calculate the similarity between sets of tuples and queries. Lets look at an example in terms of Predicates and Data Terms. Note here that Predicates would conceptually refer to the predicates in a Query, and Data Term would refer to a column name and its value in a table.

Example:

Consider a Database with 1000 tuples. Let us take the following simple example of an Expression and two Data Items.

Expression: bedrooms=3 AND price=40000
Predicate1: bedrooms=3
Predicate2: price=40000

Let the Data Items be:
Data Item1: bedrooms=3
Data Item2: price=40000
Thus,
Data Term1: bedrooms=3
Data Term2: price=40000

For Predicate1 and Data Term1:
u and v are the values of the Predicate and Data Term, respectively(in this case 3). So, $S_k(u,v)$ is IDF(u) because u=v.
IDF(u) = log(n/F(u)) {Now n, i.e., number of tuples is 1000. Also assume that F(3), i.e., the frequency of tuples in the Database with value of bedroom as 3 is 200}

Thus, IDF(u) = log(1000/200) = log(5) ~= 0.69.
Thus, for Data Item1 and the Expression, the total of the similarity value is 0.69 since the IDF for price is 0.

Now, for Data Item2, Predicate2 would be used and IDF for bedroom is 0 since bedroom is not present in Data Term2.
Let us assume that F(40000) is 500.
Thus, IDF(u) = log(1000/500) = log(2) ~=0.30.

Thus, the similarity value for the combination of Expression and Data Item1 is 0.69, and for Expression and Data Item2 is 0.30. Data Item1 would be ranked higher. This technique can also be used for ranking multiple Expressions against one Data Item.

The second type of similarity function used is QF similarity. This similarity function is used when there is a realization that a data value maybe important for ranking in spite of its frequency of occurrence in the database. QF similarity uses workload information, i.e. the occurrence of an attribute value in the workload is considered in this case. It is calculated as follows. If RQF(q) is the raw frequency of occurrence of value q of attribute A in the query string and RQF_{max} is the raw frequency of the most frequently occurring value in the workload, then the Query Frequency QF(q) is defined as $RQF(q)/RQF_{max}$. The similarity coefficient S(t,q) is QF(q) if q=t, and 0 otherwise. The Similarity for the whole Expression against the whole Data Item is the sum of the similarities for individual attributes. An example similar to the one given for IDF similarity is given below.

Example:
Consider a Database with 1000 tuples. Let us take the following simple example of an Expression and two Data Items.

Expression: bedrooms=3 AND price=40000
Predicate1: bedrooms=3
Predicate2: price=40000

Let the Data Items be:

Data Item1: bedrooms=3

Data Item2: price=40000

Thus,

Data Term1: bedrooms=3

Data Term2: price=40000

For Predicate1 and Data Term1:

S(q,t) = QF(t) if q=t. Since q=t, S(q,t) = QF(t) = QF(3) . Let us assume that RQF(3) for bedrooms is 10 and RQF_{max} is 100.

So QF(3) = 10/100 = 0.1

Since there is no Data Term corresponding to Predicate2 in Data Item1, QF value is 0.

Thus, Similarity value for combination of Expression and Data Item1 is 0.1

For Predicate2 and Data Term2:

S(q,t) = QF(40000)(since q=t). Let us assume RQF(40000) for price is 30.

So, QF(40000) = 30/100 = 0.3

The total Similarity value here is 0.3, for the combination of Expression and Data Item2.

We see here that Data Item2 would be ranked higher since it has greater similarity.

Both the similarity functions discussed above dealt with the Empty Answers problem. To deal with the Many Answers problem, the paper proposes to look at attributes in the table that are not mentioned in the query. Such attributes are called “missing attributes.” The approach used by the paper first computes the “global” importance of these attributes using QF similarity function and workload information. The global importance of missing attribute value t_k is taken as $QF_k(t_k)$. This value is then used to rank different tuples by assigning weights to a tuple in accordance with the global importance of this tuple's missing attributes.

Example:

We modify Data Item1 and Data Item2 from the previous example.

Data Item1: bedrooms=3 AND restrooms=2

Data Item2: price=40000 AND area=3000

Consider that the similarity scores for the combinations of Data Item1-Expression and Data Item2-Expression were the same, for example 0.3. This would create a problem in Ranking. This is where a "missing attribute" can come in to play. In the case of Data Item1-Expression, we can consider the extra attribute in the Data Item which is 'restrooms=2' as the missing attribute. Lets calculate QF(2) for restrooms.

$QF(q) = RQF(q)/RQF_{max}$. Let $RQF(2) = 50$; and RQF_{max} for restrooms = 100.

Therefore, $QF(2) = 50/100 = 0.5$. If there were more "missing attributes", we sum the values.

Now, in case of Data Item2-Expression, the "missing attribute" can be taken as 'area=3000'. Let us calculate QF(3000) for area.

Let RQF(3000) for area = 30. Let RQF_{max} for area = 45.

Thus, $QF(3000) = 30/45 = 0.66$. As said before, if more "missing attributes" were present, a summation of the values would be taken.

We see that the case of Data Item2-Expession combination has a higher value of QF(q) for the missing attributes. Thus, Data Item2 is ranked higher than Data Item1.

3.2.2 Uncertainty in Publish/Subscribe systems

Currently most the Publish/Subscribe systems support the crisp model where the publications or subscriptions have no kind of Uncertainty in them. A Publish/Subscribe system that handles Uncertainty is described in [2].

In many cases, exact information on publications may not be available, and also subscribers may express their subscriptions with vague constraints. Examples for publication would be the case where, for an apartment, an agent describes age as “old”. For subscription, a consumer may describe her preference for a courtyard as “long”. In both these cases, there is uncertainty. The paper [2] uses Fuzzy set theory and possibility theory to handle this uncertainty.

A Fuzzy set M on a Universal set U is a set that specifies for each element x of U a degree of membership to the Fuzzy set M. It is defined by a membership function.

$$\boldsymbol{\mu}(M) : U \rightarrow [0,1]$$

We can use membership functions to represent predicates in subscriptions that contain uncertain and vague concepts such as “age is old” etc. Similar to this is the Possibility theory, where there are possibility measures which express confidence in the possibility that a particular x is A. These are used by Publications to represent the possibility that the value of a particular attribute x is A.

In a subscription, each predicate e.g. “x is A” has a Fuzzy set A which represents a Fuzzy constraint on all possible values of A. Each predicate can be expressed by the membership function “$\boldsymbol{\mu}$”. Similarly, each attribute of a publication can be expressed using possibility distribution. We have an attribute, value pair, “(A, $\prod$(x))”, A is the attribute and $\prod$ is the possibility degree that value of A is x.

Example:

Lets take a subscription predicate as “size is medium” for House.

A possible membership function would be :

$\mu_{\mathbf{medium}}(x) = \{ 0 \text{ if } x<1500; 0.5 \text{ if } 1500<x<2500; 1 \text{ if } X>2500\}$

For a subscription, let us assume one of the predicates is (price, cheap). This is expressed as a possibility function as (price, $\prod_{\mathbf{cheap}}$).

$\prod_{\mathbf{cheap}}(x) = \{ 0 \text{ if } x<30000; 0.3 \text{ if } 30000<x<50000; 0.7 \text{ if } 50000<x<100000\}$

3.2.3 Trio (ULDB)

The TRIO group at Stanford developed the ULDB, the Uncertainty-Lineage Database [3] that looks at Uncertainty of data among other things. Of interest to this thesis is the Uncertainty part of the ULDB.

The TRIO system extends the SQL model with Alternatives, Maybe annotations, numerical confidence values and Lineage. Alternatives represent Uncertainty about the contents of a tuple. A tuple in a ULDB is called an X-tuple. Each X-tuple consists of one or more Alternatives, where each tuple is a regular tuple over the schema of the relation. In ULDB, uncertainty about the existence of a tuple(more generally an X-tuple), is denoted by a "?" annotation on the X-tuple. This indicates that the entire tuple may or may not be present. Thus it is called a maybe X-tuple. Numerical Confidences can be attached to the Alternatives of an X-tuple. This gives the Confidence with which a user thinks the tuple value is true.

Example:

An example of a couple of tables with some X-tuples:

SAW:

Table 3.2 Example Table for ULDB

ID	SAW(witness, car)
1	(Cathy, ford) : 0.5 || (Cathy, Honda) : 0.6

OWNS:

Table 3.3 Example Table for ULDB

ID	OWNS(owner, car)
1	(frank, ford) : 0.7)
2	(Carmen, jaguar) : 0.9
3	(Joe, Honda) : 0.8

In the first table SAW, the X-tuple signifies that Cathy either saw a Ford or a Honda, and if she saw a Ford, she was 50% sure it was a Ford; and if she saw a Honda, she was 60% sure that it was a Honda. The same intuition applies for the OWNS table also. This is an effective way to represent Uncertainty in the data.

TRIO also has a SQL-like language called TriQL which uses almost the same syntax as SQL but supports queries with Confidences and Lineage. The current implementation has ULDBs represented by regular relational tables, and TriQL queries and commands are rewritten automatically into SQL commands evaluated against the representation. Thus TRIO implements Uncertainty via a layer outside the DBMS.

An example query in ULDB is:
Select OWNS. owner as person INTO RESULT from SAW, OWNS where SAW. car = OWNS. car AND conf.Int(SAW)>0.5 AND conf.Int(OWNS)>0.8

Chapter 4 Semantics of Matching Uncertain Expressions and Uncertain Data

In this chapter, we develop a theoretical framework to describe the semantics of evaluating Uncertain Expressions against Uncertain Data Item. We then use this framework to describe the uncertainty model used in the different approaches discussed in Section 3.2 and the uncertainly model provided in UNXS. This framework is developed by first identifying the different cases for matching a crisp Expression to a crisp Data Item. We also introduce some variables to formally describe these cases. We then turn our attention to Uncertain Expressions and Uncertain data. In Section 4.2, we use this framework to describe the semantics of match between Predicates and Data Term in each of the approaches discussed in Section 3.2. We also present two new approaches for such matches developed in UNXS. In Section 4.3, we describe the semantics of match between Expressions and Data Items in the existing approaches as well as in UNXS.

4.1 Evaluating Crisp Expressions against Crisp Data Items

Evaluation is the procedure by which we decide whether an incoming Data Item(e.g., a Publication) matches an Expression(e.g., a Subscription). Evaluation of an Expression involves evaluation of each Predicate with the Data Item. This is evaluation at the Predicate level. We look at the various cases for evaluation of a Predicate with a Data Item. In the later sections, we extend this to include Uncertainty and come up with our Expression level evaluation.

4.1.1 Semantics of Matching at the Predicate Level

When a Crisp Expression is being evaluated against a Crisp Data Item, there are four interesting cases of evaluation of Predicates in the Expression against Data Terms in the Data Item.

1. Predicate Evaluates to TRUE **:**

In this case, the Data Item includes a Data Term which has the same identifier as the Predicate, and the Data Term value(RHS) has a matching value for the Predicate value(RHS).

2. Predicate Evaluates to FALSE :

   This is the case where a Predicate's identifier appears in a Data Term, however the value of the Data Term(RHS) does not have a matching value with the Predicate(RHS).

3. Predicate cannot be Evaluated against any Data Term in the given Data Item.

   This is the case where the Data Item does not contain any Data Term that matches the Predicate Identifier(LHS).

4. Data Term does not get evaluated against any Predicate in the Expression.

   This is the case where the Expression does not contain any Predicate whose identifier matches the Data Term Identifier(LHS). In conventional Relational Theory, this case is not considered important. However, as we will discuss later, this case is relevant to managing Uncertain data.

Let us take an example which will give us a better idea of all these cases.

Example:

Expression: price>40000 AND garage=yes AND area=3000

Data Item: price=50000 AND garage=no AND bedrooms=3

This example of Expression and Data Item covers all the cases explained above. The *price* attribute gives an example of case 1. The attribute identifiers are same and the values(RHS) also matches. The *garage* attribute gives an example of case 2. The *area* attribute is an example of case 3, since it is present in the Expression but not in the Data Item. Similarly, *bedrooms* attribute gives an example of case 4.

4.1.2 Formal Model for Evaluating Crisp Expressions against Crisp Data Items

4.1.2.1 Variables to describe Evaluation cases:

Now that we have considered the different cases when evaluating the Predicates in Crisp Expressions with Data Terms in Crisp Data Items, we introduce some variables which will be useful to formally describe these cases.

P :	Total number of predicates in the expression
D :	Total number of data terms in the data item
Pt :	Number of predicates that evaluate to true
Pf :	Number of predicates that evaluate to false
Pne :	Number of predicates that did not match any data term.
Dnu :	Number of Data terms that did not match any Predicate.

Lets look at an example to illustrate some of these terms.

Example:

Data Item: price=50000 AND bedrooms=3 AND outpaint=red AND garage=yes AND area=300

Expression: price>40000 AND bedrooms=2 AND outpaint=red AND garage=no AND restrooms=2

Here :

P = 5 ; **D** = 5 ; **Pt** = 2 ; **Pf** = 2; **Pne** = 1 ; **Dnu** = 1

4.1.2.2 Categorizing cases of Crisp Expression-Data Item Evaluation

We now use the variables we introduced in the previous section to categorize evaluation of Crisp Expressions against Crisp Data Items into eight interesting categories based on different combinations of Pne, Pf and Dnu. We have put this into a tabular format for better understanding.

Table 4.1 Example Table to show the various categorizing cases

	Variable Combination	Data Item	Expression

1	Pne=0
	Pf=0
	Dnu =0
		price=50000 AND bedrooms=3 AND outpaint=red	price>40000 AND bedrooms=3 AND outpaint=red

2	Pne=0
	Pf=0
	Dnu>0
		price=50000 AND bedrooms=3 AND outpaint=red	price>40000 AND bedrooms=3

3	Pf=0
	Dnu=0
	Pne>0
		price=50000 AND bedrooms=3	price>40000 AND bedrooms=3 AND outpaint=red

4	Pf=0 Pne>0 Dnu>0	price=50000 AND bedrooms=3 AND garage=yes	price>40000 AND bedrooms=3 AND outpaint=red

Table 4.1, continued

5	Pne=0 Pf>0 Dnu>0	price=50000 AND bedrooms=3 AND outpaint=blue AND garage=yes	price>40000 AND bedrooms=3 AND outpaint=red
6	Pne>0 Pf>0 Dnu>0	price=50000 AND bedrooms=3 AND outpaint=blue AND pool=yes	price>40000 AND bedrooms=3 AND outpaint=red AND garage=yes
7	Pne=0 Pf>0 Dnu=0	price=50000 AND bedrooms=3 AND outpaint=blue	price>40000 AND bedrooms=3 AND outpaint=red
8	Pf>0 Dnu=0 Pne>0	price=50000 AND bedrooms=3 AND outpaint=blue	price>40000 AND bedrooms=3 AND outpaint=red AND garage=yes

4.2 Semantics of Matching Uncertain Predicates Against Uncertain Data Terms

The eight categories that were mentioned in the previous section are important also in the context of Uncertainty. This is because some of the combinations of Pf, Pne and Dnu values can be used to solve the "Empty Answers" and "Too many Answers" problems that we looked at in the AutoRanking scenario.

We had explained Predicate level match for crisp data earlier. The notion of Predicate level match needs to be explained separately for Uncertain data and alternate semantics need to be developed. In UNXS, as mentioned earlier, Uncertainty is based on Confidences for both the Predicate of the Expression and the Data Term of the Data Item.

Let us introduce two more variables in addition to the ones we introduced in the last section. These two variables differentiate between the Confidences of the Data Term and the Predicate.

Alpha: The Confidence for each Data Term ($Alpha_1$, $Alpha_2$ etc)
Beta: The Confidence for each Predicate ($Beta_1$, $Beta_2$ etc)

Example:

Expression: price>40000 : 0.4 AND bedrooms=4 : 0.5 AND outpaint=red : 0.7

Data Item: price=42000 : 0.4 AND bedrooms=3 : 0.6 AND outpaint=blue : 0.7
Here,
$Alpha_1$=0.4, $Alpha_2$=0.6, $Alpha_3$=0.7
$Beta_1$=0.4, $Beta_2$=0.5, $Beta_3$=0.7

We now look at the Alternate semantics that the different systems have when there is Uncertainty, and also the one that we developed for UNXS. Each of the various methods of handling Uncertainty, respectively, TRIO, Automatic Ranking and Pub/Sub Uncertainty has its own way of handling each combination of Predicate and Data Term. Here, we describe match semantics for each of these methods within the framework that we have introduced earlier. We also describe match semantics for the UNXS system that we have developed.

Lets call the match semantic for Predicate and Data Term evaluation in general as $Match_p$. The set {0,1} denotes the set that contains 0 and 1. The interval [0,1] denotes the set that contains all decimals between 0 and 1 (and also 0 and 1). “exact” refers to an exact match between the predicate and the data term being the criteria for match. “inexact” refers to an approximate match between the Predicate and the Data Term.

1. $Match_p1$

Form: exact(pred, data term) $\rightarrow$ {0,1}
Use: Traditional DBMS

In this case, the value of the predicate is compared with that of the data term and its either a match or a non-match i.e. a 0 or 1. This is the case with Traditional DBMS.

2. $\text{Match}_p 2$

Form: exact(pred, data term) * scaling function $\rightarrow$ [0,1]
Use: Automated Ranking [4]

In this case, the match is exact. But a scaling function such as IDF or QF is used to scale the result thus producing an output in the interval [0,1].

Example:

Refer to Example in Section 3.2.1, page 25, 26.

3. $\text{Match}_p 3$

Form: exact(pred, data term) * Boolean($\text{Alpha}_i >= \text{Beta}_i$) $\rightarrow$ {0,1}
Use: TRIO(ULDB) [3]

Here, apart from the exact match between the predicate and the data term, the Confidences are also checked. If the Confidence of the data term is greater than or equal to that of the predicate, the result is 1 else 0.

Example:

Refer to Example in Section 3.2.3, page 29, 30

4. $\text{Match}_p 4$

   Form: inexact(pred, data term, Beta) → [0,1]
   Use: Publish/Subscribe system [2]

   Here, the match between the predicate and data term is approximate. In addition, the degree of match is checked with an additional threshold value (similar to Beta values) to come up with an answer in the interval [0,1].

   Example:
   We take the simple example of one Predicate in the Expression and one Data Term in the Data Item.
   Predicate1: price = low
   Data Term1: price =20000

   Let us assume we have computed the degree of match between the Predicate and the Data Term and let it be 0.9. For each predicate there is an additional value against which the degree of match is checked. Let this Beta value be 0.8. Since the degree of match is greater than this value, the Predicate is said to match the Data Term.

5. $\text{Match}_p 5$

   Form: inexact(pred, data term) * scaling function → [0,1]
   Use: Automated Ranking for Numeric Attributes [4]

   Since Numeric Attributes might have approximate matches, inexact match is used. The distance between the two values is evaluated and the closer the two values are, the higher the match value give to this Predicate-Data Term combination. Also, a scaling function similar to Match_p2 is used to obtain a value in the interval [0,1].

Example:

Predicate: price=30000

Data Term1: price=31000

Data Term2: price=36000

In this example, Data Term1 will be ranked higher than Data Term2.

Now, we look at the two matches that are developed in this thesis for handling Uncertainty in UNXS. In our implementation, we have Expressions stored in the columns of a Database and Data Items are supplied in queries. So, our Evaluation context refers to evaluating multiple Expressions against one Data Item.

6. Match_p6

Form: exact(pred, data term) * scaling function(Alpha_i, Beta_i, q, t) $\rightarrow$ [0,1]

Use: UNXS

(Here, q refers to the Predicate, t refers to the Data Term)

In UNXS, we have Confidences attached for each Data Term of the Data Item and each Predicate of the Expression. Our goal was to devise a strategy so that evaluation uses the Confidences level. The methods used in AutoRanking [4] provided a good base for exploring options, and we came up with the following extension.

We extend the QF similarity function to include Alpha, Beta in addition to q and t to calculate QF similarity. Unlike AutoRanking [4], we also allow inequality operators in the Predicates, i.e., in q. After q and t are compared, we check if Alpha>=Beta. Based on both the results, the result maybe QF(q) or 0. This result again is in the interval [0,1].

Example:

Consider a Database with 1000 tuples. Let us take the following simple example of a Data Item and two Expressions.

Expression1: bedrooms>2 : 0.4 AND price<30000 : 0.5
Expression2: price=40000 : 0.6
Expression1 has two Predicates:
Predicate1: bedrooms>2 : 0.4
Predicate2: price<30000 : 0.5
For Expression2, we have only one Predicate;
Predicate3: price=40000 : 0.6

Let the Data Item be :
Data Item1: bedrooms=3 : 0.6 AND price=40000 : 0.8
We have two Data Terms in the Data Item;
Data Term1: bedrooms=3 : 0.6
Data Term2: price=40000 : 0.8

Similarity measure S(q,t) = QF(t) if q=t in the AutoRanking method.
In our extension, first we check if the values of q and t are satisfied not only for equality, but also for range clauses. Then we also check whether Alpha>=Beta. If these conditions are satisfied, then S(q,t) = QF(t).
Note: Here also, q refers to a particular Predicate's value and t refers to the corresponding Data Term's value.
Thus,
S(q,t) = QF(t) if {q matches t, and Alpha_q >= Beta_t }

Expression1 and Data Item1:

For Predicate1 and Data Term1:

Here, q matches t since Predicate1 is bedrooms>2, and Data Term1 is bedrooms=3. Here, $Alpha_q = 0.6$ and $Beta_t = 0.4$. Thus, $Alpha_q >= Beta_t$.
Since both the conditions are satisfied, S(q,t) = QF(t) = QF(3).
Let us assume that RQF(3) for bedrooms is 10 and RQF_{max} is 100.
So QF(3) = 10/100 = 0.1.

For Predicate2 and Data Term2:
Here, q and t do not match since Predicate2 is price<30000 and Data Term2 is price=40000. Thus, the S(q,t) = 0.

The net Similarity value for Expression1 and the Data Term is given by taking the average of the Similarity values for each Predicate-Data Term combination. Thus, here it is : (0.1+0)/2 = 0.05.

Expression2 and Data Item1:

For Predicate3 and Data Term2:
Here too, q and t match and $Alpha_q >= Beta_t$. Thus, S(q,t) = QF(40000). Let us assume RQF(40000) for price is 30.

So, QF(40000) = 30/100 = 0.3
Since there is no Predicate corresponding to Data Term1 in the Expression, the Similarity value is 0.
The net Similarity value for Expression2 and the Data Item is again given by taking the average of 0.3 and 0. The value is 0.15.
In terms of a Ranking scenario, Expression2 will have higher priority over Expression1.

7. Match$_p$7

Form: exact(pred, data term) * scaling function(Alpha$_i$, Beta$_i$, q, t) → [0,1]
Use: UNXS

We have also developed an alternative Match semantic which does not use IDF or QF. The motivation for this simpler method is that it does not require statistics on the workload or the data stored in the database.

In this case also, an exact match is done for Predicate – Data Term combination, i.e. whether q matches t is checked. We also check whether Alpha>=Beta. Apart from this, we also compute the average of Alpha and Beta values and return this as the match value. The value returned will be in the interval [0,1].

Example:
Let us consider the same example as in Match$_p$6.

Expression1: bedrooms>2 : 0.4 AND price<30000 : 0.5
Expression2: price=40000 : 0.6
Predicate1: bedrooms>2 : 0.4
Predicate2: price<30000 : 0.5
Predicate3: price=40000 : 0.6

Data Item1: bedrooms=3 : 0.6 AND price=40000 : 0.8
Data Term1: bedrooms=3 : 0.6
Data Term2: price=40000 : 0.8

Expression1 and Data Item:

For Predicate1 and Data Term1:

Here, the Data Term and Predicate values match, i.e., q matches t, since bedrooms>2 satisfies bedrooms=3. Alpha_q >= Beta_t, since 0.6>=0.4. Now, let us compute the average of 0.6 and 0.4, which is 0.5.

For Predicate2 and Data Term2:
Here, since q and t do not match. Thus, the Similarity value is 0.

Now, for the combination of Expression1 and the Data Item, the Similarity score will be the average of the two scores. (0.5+0)/2 = 0.25.

Expression2 and Data Item:

For Predicate3 and Data Term2:
Here, q matches t, and Alpha_q >= Beta_t. So, we take the average of Alpha_q(0.8) and Beta_t(0.6) is taken, which is 0.7. This is the value that is returned for the Evaluation.

Since there is no Predicate in the Expression that corresponds to Data Term1 in the Data Item, the Similarity is taken as 0.

The net Similarity value is the average of both values. (0.7+0)/2 = 0.35.

We see again that Expression2 has a higher Similarity value, which means it will be ranked higher if there was any kind of Ranking.

4.3 Evaluation Semantics for Expressions and Data Items with Approximate Matching and Uncertainty

In this section, we describe the semantics of matching between Expressions and Data Items in TRIO, Publish/Subscribe, Automated Ranking and UNXS for the eight categories of combinations of variables that we listed in Section 4.1. We will describe these semantics using the semantics of Predicate-Data Term level match semantics introduced in Section 4.2. In Section 4.3.1, we consider the match between crisp Expressions and crisp Data Items, but allowing for approximate match at the Predicate-Data Term level. In Section 4.3.2, we will describe the semantics of match for Uncertain Expressions and Uncertain Data Items.

4.3.1 Crisp Data Items, Crisp Expressions

The following table gives a comparison of the various techniques for handling Uncertainty including our technique, UNXS. The values in the table are values of Expression-Data Item evaluation match.

Table 4.2 Comparison Table for Evaluation(without Uncertainty)

	Cat_1	Cat_2	Cat_3	Cat_4	Cat_5	Cat_6	Cat_7	Cat_8
	Pne=0 Pf=0 Dnu=0	**Pne=0 Pf=0 Dnu>0**	**Pne>0 Pf=0 Dnu=0**	**Pne>0 Pf=0 Dnu>0**	**Pne=0 Pf>0 Dnu>0**	**Pne>0 Pf>0 Dnu>0**	**Pne=0 Pf>0 Dnu=0**	**Pne>0 Pf>0 Dnu=0**
Traditional DBMS	1	1	0	0	0	0	0	0
Trio (ULDB)	1	1	0	0	0	0	0	0

Table 4.2, continued

Automatic Ranking	IDF or QF Similarity	Σ (QF(tk)= RQFq/RQFmax)	n/a	n/a	IDF similarity $\Sigma S_k(t_k, q_k)$	n/a	IDF	n/a
Pub/Sub Uncertainty	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
UNXS	1	Σ (QF(tk)= RQFq/RQFmax)	IDF	IDF	IDF	IDF	IDF	IDF

Lets analyze each result :

Traditional DBMS :

This is self-explanatory. The match here corresponds to Match_p1 in Section 4.2.1. Categories 1 and 2 correspond to a Query (or equivalently Expression) being run on a Table with columns (Data Item). The result is 1 since all predicates match all the data terms. In the second case, there are extra data terms, but these can be seen as extra columns in the table that don't affect the result. For the rest of the cases, the result will be 0 since they either have Pne>0, which means there are predicates in the query which have a NULL value in the Table, or Pf>0 which means one of the predicates did not match.

TRIO(ULDB) :

The results for TRIO default to the results of a Traditional DBMS as there is no Uncertainty. The match corresponds to Match_p3 in Section 4.2.1. Thus, the results are the same.

Automated Ranking :

The match here can correspond to both Match_p2 and Match_p5 in Section 4.2.1.

Cat_1 corresponds to a situation with Many Answers problem.

Example:

Data Item : price=50000 AND Bedrooms=3 AND outpaint=red

Expression : price>40000 AND Bedrooms=3 AND outpaint=red

IDF similarity equation is used to compute the similarity value in this case. This is done with some assumptions made

1. n-no of tuples in the database

2. $F_k(t)$-frequency of tuples in the database where the predicate occurs with the corresponding value.

Let an attribute be A_k.. For any pair of values, *u* and *v* in A_k's domain, let the quantity $S_k(u, v)$ be $IDF_k(u)$ if $u = v$, and 0 otherwise.

So, the final similarity measure will be = $\Sigma(S_k(t_k, q_k))$

Where t and q are the attributes of the tuple and query respectively.

Here, $IDF_k(t) = n/F_k(t)$ as described in a previous section. Note: QF similarity can also be used in this case.

Category 2 corresponds to the "missing attribute" scenario where the Data term(s) corresponding to Dnu>0 can be thought of as the missing attributes. This computation also requires that some assumptions be made about the values of RQF_q and RQF_{max}.

The categories 5 and 7 correspond to the "Empty Answers" problem. These two cases have either one predicate not matching the constant of a Data Term, or a Data Term attribute not having a corresponding Predicate. In both the cases, approximate matching is done by using QF or IDF for Numeric Attributes

($Match_p5$). The rest of the categories with Pne>0 are not applicable for the AutoRanking scenario. Pne>0 typically means that there is a Predicate in the Expression which does not have any matching Data Term (no column for that Predicate in the table). This is not handled by the AutoRanking method.

Publish/Subscribe:

The Predicate match here corresponds to $Match_p4$. In this case, as there is no uncertainty to work with, the case does not apply.

UNXS :

This corresponds to $Match_p2$ and $Match_p5$ in Section 4.2.1. Since we do not have Confidences to work with, we here use methods from Automated Ranking to produce results.

The ideal output for the first category would be Pt/P = 1. This is because the "too many answers" problem does not map to our model. The reason for this is that Data Items are given one at a time in a query. Thus, there is no chance of too many Data Items at one point of time. A better way to handle this would be to make the Expression stricter, i.e., include more predicates in the Expression or just make each predicate more selective. Category 2 is a case of probable "too many answers" with "missing attributes" in the query. The result is also 1 for the same reason as explained above.

The rest of the categories can be computed using either QF or IDF similarity. But recall that IDF similarity calculations require information about values in the Database(Data Items in our case), which is not readily available as it is given on the fly. However, QF similarity requires workload information (Expressions in our case), and we have this information stored in the Database. We choose to use QF similarity for this reason.

Example:

Data Item : price=50000 AND bedrooms=3
Expression : price>40000 AND bedrooms=3 AND outpaint=red

The similarity is defined as S(t,k) = QF(q) if q=t , and 0 otherwise
We extend this for predicates with inequality also.

Let us assume RQF_{max} for the three predicates price, bedrooms and outpaint to be 60, 30 and 20 respectively ; and RQF(q) for the predicate values is 40, 10, 5 respectively.

The final value is Σ(S(t,k)). For the last predicate corresponding to Pne>0, the corresponding value for Data term is taken as 0.
QF(q) is RQF_q/RQF_{max} .

So, we have 40/60 + 10/30 + 0 = 60/60 = 1 which is the value of the result.

4.3.2 Uncertain Data Items, Uncertain Expressions

Table 4.3 Comparison Table for Evaluation(with Uncertainty)

	Cat_1	Cat_2	Cat_3	Cat_4	Cat_5	Cat_6	Cat_7	Cat_8
	Pne=0 Pf=0 Dnu=0	**Pne=0 Pf=0 Dnu>0**	**Pne>0 Pf=0 Dnu=0**	**Pne>0 Pf=0 Dnu>0**	**Pne=0 Pf>0 Dnu>0**	**Pne>0 Pf>0 Dnu>0**	**Pne=0 Pf>0 Dnu=0**	**Pne>0 Pf>0 Dnu=0**
Traditional DBMS	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Trio (ULDB)	{0,1}	{0,1}	0	0	0	0	0	0

Table 4.3, continued

Automatic Ranking	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Pub/Sub Uncertainty	{0,1}	{0,1}	0	0	0	0	0	0
UNXS	QF_{new}	$QF(t_k)$= RQFq/RQFmax} for missing attributes	QF_{new}	QF_{new}	QF_{new}	QF_{new}	QF_{new}	QF_{new}

Let us analyze each result for this table :

Traditional DBMS :

Since Confidences are not part of a Traditional DBMS, it is not applicable in this scenario.

TRIO(ULDB) :

In the TRIO method, Confidence was used for evaluation besides the values of the predicates and data term. So, for the first two categories, it would be either 1 or 0 depending upon the match of the Confidences for the data terms and predicates. For the rest of the categories, it would be 0 since these cases will correspond to Traditional DBMS and the values of the Confidences would not matter.

Automated Ranking:

Automated Ranking is not applicable to this scenario since predicates or data terms having Confidences is not considered by this method.

Publish/Subscribe :

For the first two categories, this result can be either 1 or 0. If the Confidences of all the Predicates satisfy the Confidences of the corresponding Data Terms, the value is 1, else it is 0. The values are 0 for the rest of the Categories since these correspond to either a no match or a case where the query (Expression) has Predicates that are not present in the column of a table (Data Item).

UNXS :

We can use either Match_p6 or Match_p7 to process all the cases.

The first category should now use QF similarity extended (using Alpha and Beta) which is explained in Section 4.2.1 for Match_p6. We call it "$\mathbf{QF_{new}}$" and this is used to obtain the results.

The second category namely Cat_2 uses the same result as "$\mathbf{QF_{new}}$", but since there are extra terms in the Data Item, the result for the "missing attribute" problem's solution formula is used, i.e., $QF(t_k)$ where t_k refers to the "missing attribute" is also taken in to account in the calculations.

Example:

If we refer back to the example in Section 4.2.1 for Match_p6:

Expression1: bedrooms>2 : 0.4 AND price<30000 : 0.5
Expression2: price=40000 : 0.6

Data Item1: bedrooms=3 : 0.6 AND price=40000 : 0.8

In this case, for **Expression2-Data Item1** combination, bedrooms=3 will not have 0 as the similarity value, but will have QF(3) for bedrooms.

Let us assume that RQF(3) for bedrooms = 40; and RQF_{max} for bedrooms = 100. Therefore, QF(3) = RQF(3)/ RQF_{max} = 40/100 = 0.4.

The other similarity value as obtained before is 0.3. Thus, the net similarity value is (0.3+0.4)/2 = 0.35. This value is larger than the 0.15 that we got earlier in $Match_p6$, when we did not consider "missing attributes".

Chapter 5 Supporting Uncertain Expressions In a DBMS

As discussed in Chapter 1, integrating of Expressions in a DBMS can be very beneficial. This way, we can process these Expressions more efficiently, and it also allows Indexes to be built over them for faster access. This is especially useful for large amounts of data, which is typically what the case would be for most applications. This Chapter describes the implementation done to manage Uncertain Expressions in the Postgresql Database.

5.1 Expression Data Type

The implementation was done on Postgresql 8.0.1 that was installed on an Ubuntu Linux machine. The decision to use Postgresql was based on the fact that it is an extendible database where it is possible to add new User Defined Types (UDTs) and user defined functions to the DBMS [10]. The source code for adding data types was in C. Once the file is compiled and linked, all the functions needed for the data type are available in Postgresql and the data type is also available for use. Once this is done, a table can be created with some columns of this data type, rows with values for this data type can be inserted and queries performed on such tables. A subset of UNXS is also supported by our implementation.

5.2 Details of Implementation

In Postgresql, any data type has an EXTERNAL form and an INTERNAL form. The External form of a data type defines how the user enters a value and how a value is displayed to the user. The Internal form defines how a value is represented inside the DBMS. When we choose the internal form for any data type, we want to choose a representation that makes it easy to define and implement operations (mathematical, relational or logical). This also helps in adding the supporting functions if any, because the actual values that need to be evaluated can be obtained in the Internal form.

With this concept in mind, we decided on the External form that is the same as the BNF notation introduced in Chapter 2. We repeat it here:

Predicate ::= Identifier Operator Constant : Confidence

Operator ::= < | > | = | <= | >=

Data Term ::= Identifier Operator Constant : Confidence

Operator ::= =

Example:

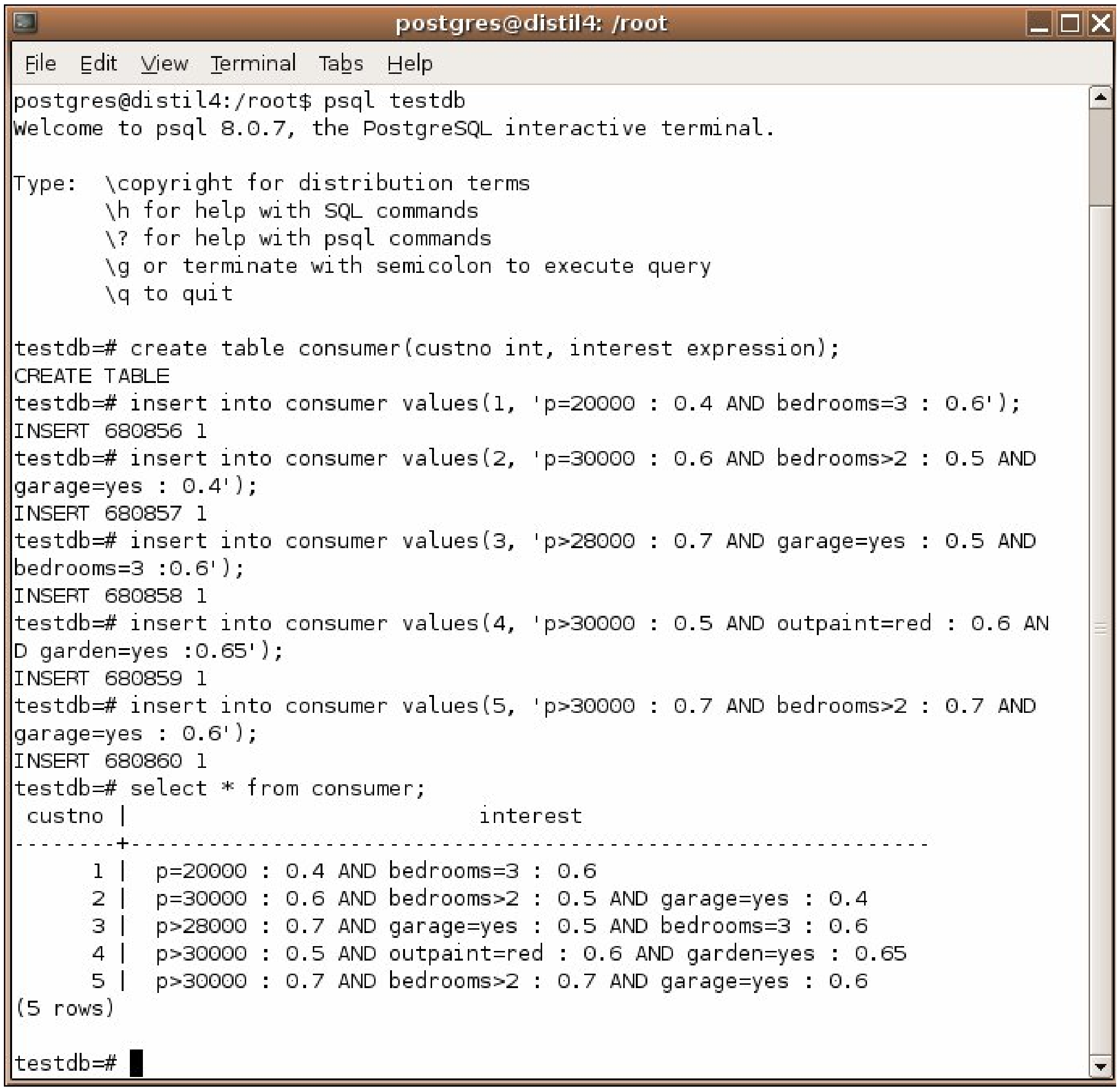
bedrooms=4 : 0.5 AND price>40000 : 1.0

In order to evaluate the Expressions against Data Item, we need to first carry out evaluation at the level of Predicates and Data Terms. Hence, we need to break down each Expression into its separate predicates. Each part of a predicate also has to be identified and stored, so that the match can be carried out. For this, we created an Internal representation where the whole Expression is stored inside an array. Each element of the array is a structure with 4 main attributes. The four attributes store the four parts of a predicate, namely the Identifier, Operator, Constant and Confidence. The size of an array equals the number of predicates in that particular Expression. This is the Internal representation of the Expression Data Type.

The definition of a new Data Type in Postgresql requires mechanism to convert the Internal representation to External and vice-versa. For this, we need two functions: the Input function and the Output function. The Input function converts from External to Internal representation; and the Output function converts from the Internal representation to the External representation. These two functions were added to Postgresql and allow us to perform basic INSERT and SELECT statements.

Below is a screenshot of a few INSERT statements and an example SELECT statement in PSQL. PSQL is the name for the SQL syntax supported in Postgresql.

Figure 5.1 Screenshot of INSERT and SELECT in Postgresql



```
postgres@distil4:/root$ psql testdb
Welcome to psql 8.0.7, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=# create table consumer(custno int, interest expression);
CREATE TABLE
testdb=# insert into consumer values(1, 'p=20000 : 0.4 AND bedrooms=3 : 0.6');
INSERT 680856 1
testdb=# insert into consumer values(2, 'p=30000 : 0.6 AND bedrooms>2 : 0.5 AND
garage=yes : 0.4');
INSERT 680857 1
testdb=# insert into consumer values(3, 'p>28000 : 0.7 AND garage=yes : 0.5 AND
bedrooms=3 :0.6');
INSERT 680858 1
testdb=# insert into consumer values(4, 'p>30000 : 0.5 AND outpaint=red : 0.6 AN
D garden=yes :0.65');
INSERT 680859 1
testdb=# insert into consumer values(5, 'p>30000 : 0.7 AND bedrooms>2 : 0.7 AND
garage=yes : 0.6');
INSERT 680860 1
testdb=# select * from consumer;
 custno |                           interest
--------+-------------------------------------------------------------
      1 |  p=20000 : 0.4 AND bedrooms=3 : 0.6
      2 |  p=30000 : 0.6 AND bedrooms>2 : 0.5 AND garage=yes : 0.4
      3 |  p>28000 : 0.7 AND garage=yes : 0.5 AND bedrooms=3 : 0.6
      4 |  p>30000 : 0.5 AND outpaint=red : 0.6 AND garden=yes : 0.65
      5 |  p>30000 : 0.7 AND bedrooms>2 : 0.7 AND garage=yes : 0.6
(5 rows)

testdb=# 
```

To support evaluation of Uncertain Expressions over Uncertain Data Items, we needed to provide relevant operators or functions. A function was needed which should be able to evaluate a Data Item, against the Expressions stored in a table. For this an EVALUATE operator is added. We have implemented a subset of UNXS. But, the existing implementation can be extended to implement all the theoretical extensions introduced by us.

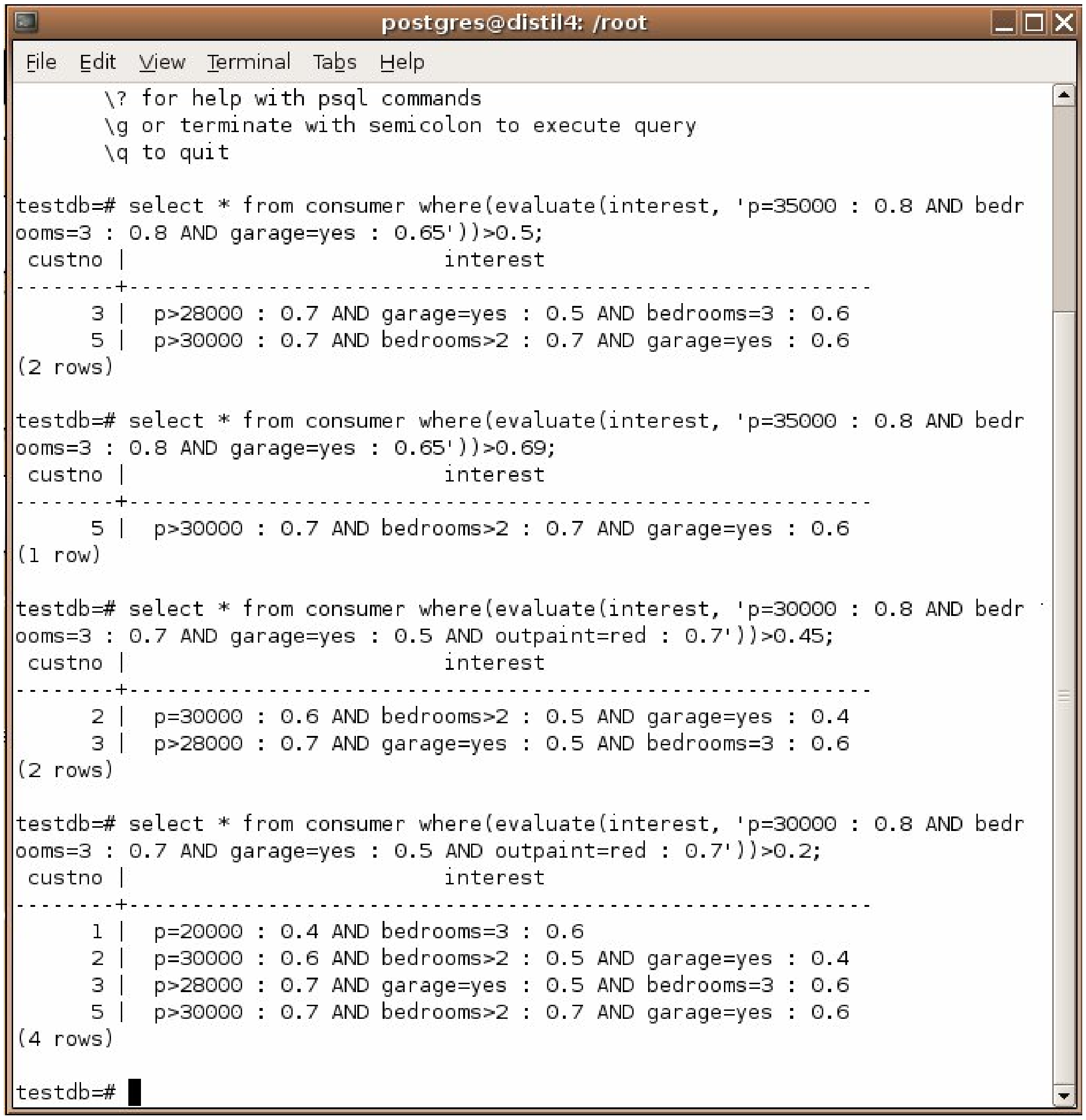
We have implemented an Evaluate operator corresponding to Match_p7, where for each combination of Predicate and Data Term, if the Predicate value was equal to the Data

Term value, and the Confidence of the Data Term was greater than or equal to that of the Predicate, then the average of the Confidences is given as the Similarity value. The overall match value for the Expression is then computed as the average of each of these similarity values, where the average is over P, i.e., the number of Predicates.

Note: The Evaluate operator comes with an option to give a *Threshold* value along with a comparison operator (>, <, =, <=, >=). The Similarity value computed for each Expression is compared with the Threshold and those rows for which the Expression satisfies the Threshold value are given in the result.

Lets look at a screenshot of a few SELECT statements using the EVALUATE operator. The table here is the same table shown in the previous screenshot with the same five rows.

Figure 5.2 Screenshot of Evaluation of Expressions against Data Items using Evaluate operator



```
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=# select * from consumer where(evaluate(interest, 'p=35000 : 0.8 AND bedr
ooms=3 : 0.8 AND garage=yes : 0.65'))>0.5;
 custno |                          interest
--------+------------------------------------------------------------
      3 |  p>28000 : 0.7 AND garage=yes : 0.5 AND bedrooms=3 : 0.6
      5 |  p>30000 : 0.7 AND bedrooms>2 : 0.7 AND garage=yes : 0.6
(2 rows)

testdb=# select * from consumer where(evaluate(interest, 'p=35000 : 0.8 AND bedr
ooms=3 : 0.8 AND garage=yes : 0.65'))>0.69;
 custno |                          interest
--------+------------------------------------------------------------
      5 |  p>30000 : 0.7 AND bedrooms>2 : 0.7 AND garage=yes : 0.6
(1 row)

testdb=# select * from consumer where(evaluate(interest, 'p=30000 : 0.8 AND bedr
ooms=3 : 0.7 AND garage=yes : 0.5 AND outpaint=red : 0.7'))>0.45;
 custno |                          interest
--------+------------------------------------------------------------
      2 |  p=30000 : 0.6 AND bedrooms>2 : 0.5 AND garage=yes : 0.4
      3 |  p>28000 : 0.7 AND garage=yes : 0.5 AND bedrooms=3 : 0.6
(2 rows)

testdb=# select * from consumer where(evaluate(interest, 'p=30000 : 0.8 AND bedr
ooms=3 : 0.7 AND garage=yes : 0.5 AND outpaint=red : 0.7'))>0.2;
 custno |                          interest
--------+------------------------------------------------------------
      1 |  p=20000 : 0.4 AND bedrooms=3 : 0.6
      2 |  p=30000 : 0.6 AND bedrooms>2 : 0.5 AND garage=yes : 0.4
      3 |  p>28000 : 0.7 AND garage=yes : 0.5 AND bedrooms=3 : 0.6
      5 |  p>30000 : 0.7 AND bedrooms>2 : 0.7 AND garage=yes : 0.6
(4 rows)

testdb=#
```

Addition of other match semantics, e.g., Match$_p$6, can be done by adding a different Evaluate function that implements those semantics. There is also the option of adding other Evaluate operators if additional Match semantics are developed in the future.

Chapter 6 Conclusions and Future work

Expressions are used in a range of applications like Publish/Subscribe[2, 13], Website Content Personalization[12], Ecommerce[11], etc. Integrating support for Expressions in a DBMS provides an efficient and scalable platform for applications that use Expressions. Such support is being provided by Oracle 10g[1]. Current DBMS though only support crisp data model and cannot support Uncertainty in Expressions or Data.

In this thesis, we describe how DBMS can be extended to manage Uncertain Expressions and Data. To achieve this goal, we developed a theoretical framework to compare and contrast popular schemes to handle Uncertainty in DBMS and Publish/subscribe systems. We then described UNXS, the system we have developed to manage Uncertain Expressions and Data in DBMS. In addition to developing the theoretical framework, we also implemented a subset of UNXS in the Postgresql DBMS by adding a new Expression Data Type to Postgresql. The Expression Data Type enables Expressions to be stored in a column of a database table. We also implemented an EVALUATE operator which evaluates Uncertain Expressions against Uncertain Data Items. To the best of our knowledge, this is the first integration of Uncertain data support(in terms of Expressions) within a DBMS as opposed to current efforts where a layer outside the DBMS is used to translate to and from a crisp DBMS.

The work accomplished in the thesis also highlights many possible future extensions. In this thesis, Uncertainty occurring in Expressions is modeled in terms of a single value for each of the Predicates. A possible extension could support a probability distribution, instead of this single value. Another important assumption that we made was that of independence of each Predicate. Thus the match for the overall Expression is computed without considering any effect of the match of a Predicate on the match of other Predicates in the Expressions. This assumption may not be justified in all cases. Hence, a more complex formal model that considers statistical dependence should be investigated in future.

The retrieval of Expressions and their processing can be made much faster if Expressions could be indexed. In [1], the authors have described a special indexing structure for indexing crisp Expressions. Development of appropriate indexing techniques for UNXS appears a fruitful research area. The GIST functionality in Postgresql provides extensible support for adding custom index structures. If new indexing structures are developed for Uncertain Expressions, the GIST functionality could be used to implement and evaluate the performance of these indexing structures.

References

[1]. Yalamanchi, A., Srinivasan, J., and Gawlick, D., "Managing Expressions as Data in Relational Database Systems", CIDR Conference, Asilomar, 2003.

[2]. Liu, H., and Jacobsen, H-A., "Modeling Uncertainties in Publish/Subscribe Systems", Proceedings of the 20th International Conference on Data Engineering, 2004.

[3]. Benjelloun, O., Das Sarma, A., Hayworth, C., Widom, J., "An Introduction to ULDBs and the Trio System", in IEEE Computer Society Technical Committee on Data Engineering, March 2006, 5-16.

[4]. Agarwal, S., Chaudhuri, S., Das, G., Gionis, A., "Automated Ranking of Database Query Results", Proceedings of the CIDR Conference, 2003.

[5]. Chaudhuri, S., Das, G., Hristidis, V., Weikum, G., "Probablistic Ranking of Database Query Results", 30th VLDB Conference, Toronto, Canada, 2004.

[6]. Das Sarma, A., Benjelloun, O., Halevy, A., Widom, J., "Working Models for Uncertain Data". 22nd International Conference on Data Engineering (ICDE'06).

[7]. Das Sarma, A., U. Nabar, S., Widom, J., "Representing Uncertain Data: Uniqueness, Equivalence, Minimization, and Approximation", *http://dbpubs.stanford.edu/pub/2005-38*.

[8]. Stonebraker, M., A.Rowe, L., "The Design of Postgres", 1986, International Conference on Management of Data**,** Proceedings of the 1986 ACM SIGMOD international conference on Management of data.

[9]. H. Liu and H.-A. Jacobsen. A-topss – a publish/subscribe system supporting approximate matching. In *28 th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.

[10]. Postgresql 8.2 Beta 1 Documentation., *http://momjian.us/main/writings/pgsql/sgml/*.

[11]. Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Evaluation of cooperation models for electronic business. In Information Systems for E-Commerce, Conference of German Society for Computer Science / EMISA, Austria, November 2000.

[12]. Ceri, S., Fraternali, P., and Paraboschi, S. "Datadriven one-to-one web site generation for data- intensive applications", Proc. 25th International Conference on Very Large Databases 1999 : 615-626.

[13] Garofalakis, M., Suciu, D (co-editors). Bulletin of IEEE Computer Society Technical Committee on Data Engineering. Special Issue on Probabilistic Data Management. March 2006.

APPENDIX

The code for the Input function mainly consists of parsing the Expression and storing each of the predicates one by one in the structure array. The structure also had a fourth attribute called 'Done' which would take a '0' or a '1' depending upon whether we have reached the end of the Expression or not. This will help in retrieving the complete Expression later on in the Output function. The Input function is conveniently called **Expression_in1**( ) and the Output function is called **Expression_out1**( ). There is also a third function called **Evaluate**( ) which does the Evaluation of Expression when a Data Item is used with the Evaluate function in the Query. The Evaluate( ) function will basically get each Data Term of a Data Item and compare it with each Predicate of an Expression for each of the rows in the table. All this code in the C language also used some Postgresql Macros which basically fetched the input Expressions to be used in the code and also displayed the required output Expressions on the user screen. The C file containing all this code was called **"Expression.C".**

The **"Expression.C"** file which is stored in the path /mogin/postgresql/src/tutorial now needs to be compiled and then stored as a ".so" (shared object) file which is the format that postgresql understands. The session with the commands is shown below:

```
root@distil4:/mogin/postgresql-8.0.7/src/tutorial # gcc -
     I/mogin/psql/include/postgresql/server -fpic -c
expression.c

root@distil4:/mogin/postgresql-8.0.7/src/tutorial # gcc -
     I/mogin/psql/include/postgresql/server -shared -o
expression.so expression.o
```

Once the file is compiled and a shared object corresponding to the C file has been made successfully as shown above, we start up the Postgresql database and create a new SQL session. This is shown below:

```
root@distil4:~ # su postgres
postgres@distil4:/root$ /mogin/psql/bin/postmaster -D
     /mogin/psql/data

LOG:  database system was shut down at 2006-04-07 12:14:01
CDT
LOG:  checkpoint record is at 0/AA4F64
LOG:  redo record is at 0/AA4F64; undo record is at 0/0;
shutdown TRUE
LOG:  next transaction ID: 575; next OID: 33614
LOG:  database system is ready
```

We start a PSQL session in another terminal:

```
root@distil4:~ # su postgres
postgres@distil4:/root$ export PATH=$PATH:/mogin/psql/bin
postgres@distil4:/root$ psql testdb
Welcome to psql 8.0.7, the PostgreSQL interactive terminal.
```

In this session, we register the functions **Expression_in1**( ), **Expression_out1**( ) and **Evaluate**( ) in Postgresql. The commands used for this purpose are :

```
testdb=# create or replace function expression_in1(cstring)
returns expression as '/mogin/postgresql-
8.0.7/src/tutorial/expression' language c immutable strict;
NOTICE:  type "expression" is not yet defined
DETAIL:  Creating a shell type definition.
CREATE FUNCTION
```

```
testdb=# create or replace function expression_out1(expression)
returns cstring as '/mogin/postgresql-
8.0.7/src/tutorial/expression' language c immutable strict;
NOTICE:  argument type expression is only a shell
CREATE FUNCTION

testdb=# create or replace function evaluate(expression,
expression) returns float as '/mogin/postgresql-
8.0.7/src/tutorial/expression' language c immutable strict
CREATE FUNCTION

                                    ^
```

Once the functions are registered, we need to specify the command to create a new Data type called Expression into Postgresql.

```
testdb=# create type expression(internallength=variable,
input=expression_in1, output=expression_out1, alignment=double);
CREATE TYPE
```

Now, we can create a table with one of the attributes of Data Type Expression and store Expressions in the column with that attribute.

Vita

Moginraj Mohandas was born in the Kannur District of the Kerala State in India, on May 2, 1981. He completed his Higher Secondary Schooling in 1999 at Ideal Indian School, Doha, Qatar. Later in June 2003, he received his Bachelors Degree in Computer Science and Engineering from the Lal Bahadur Shastri College of Engineering, Kasaragod, Kerala. He completed his Masters Degree in Computer Science from the University of New Orleans in May 2007. He sees his future in the Software Development Industry.