PossDB: An Uncertainty Data Base Management System based on Conditional Tables

Gösta Grahne
Concordia University
Montreal, Canada, H3G 1M8
grahne@cs.concordia.ca

Adrian Onet
Concordia University
Montreal, Canada, H3G 1M8
a_onet@cs.concordia.ca

Nihat Tartal
Concordia University
Montreal, Canada, H3G 1M8
m_tartal@encs.concordia.ca

1. INTRODUCTION

The PossDB (Possibilty Data Base) system is a Database Management System fully supporting incomplete information. The system is based on conditional tables [4] which generalize relations in two ways. First, in the entries in the columns, variables, representing unknown values, are allowed in addition to the usual constants. The same variable may occur in several entries. A conditional table T represents a set of complete instances, each obtained by substituting each variable with a constant, that is, applying a valuation v to the table, where v is a mapping from the variables to constants. The second generalization is that each tuple is associated with a condition, which is a Boolean combination of atoms of the form $x = y, x = a, a = b$, for x, y null values (variables), and a, b constants. In obtaining a complete instance $v(T)$, only tuples whose local conditions evaluate to *true* by v are included in $v(T)$. Below is an example of a conditional table T and two complete instances it represents.

A	B	CONDITION
a	x	$x = b$
x	d	$x = c$
x	e	*true*

$v(x) = b$

A	B
a	b
b	e

$v(x) = c$

A	B
c	d
c	e

The conditional tables support the full relational algebra [4] and both possible and certain answers. A (complete) tuple t is in the possible answer to a query Q, if $t \in Q(v(T))$ for *some* valuation v, and t is in the certain answer if $t \in Q(v(T))$ for *all* valuations v, Examples of select-project-join queries and their answers are given in Section 2.

Uncertainty management is an important topic in data exchange and information integration. In these scenarios the data stored in one data base has to be restructured to fit the schema of a different database, which forces the introduction of null values in the translated data, since the second schema can contain columns not present in the first. Conditional tables are the oldest and most fundamental instance of a *semiring-labelled* database [2]. By choosing the appropriate semiring, labelled databases can model a variety of phenomenons in addition to incomplete information. Examples are probabilistic databases, various forms of database provenance, databases with bag semantics, etc. It is our view that the experiences obtained from the PossDB project will also be applicable to all semiring based databases. Notable other systems similar to ours include MayBMS - A Probabilistic Database Management System [3] and Trio - A System for Integrated Management of Data, Uncertainty, and Lineage [5] However, neither of these systems are based on conditional tables, and to the best of our knowledge, PossDB is the first implemented system based on conditional tables. In the future we plan to extend our system to support the Conditional Chase [1], a functionality which is highly relevant in data exchange and information integration.

We have built the PossDB system on top of Postgres, which is an open source extensible relational database management system. In case there are no variables and conditions, PossDB acts as a classical DBMS. However, at this point we have restricted the allowed datatypes to strings. (note that most of the other datatypes can be simulated with strings). In the future we will extend the allowed datatypes to include all the base datatypes.

When compared with any other classical DBMS, our system allows the following extra operations:

- Creation of conditional tables
- Querying conditional tables
- Materializing views as conditional tables
- Testing for tuple possibility and certainty in conditional tables

In order to facilitate all these operations we extended the ANSI SQL language and created a new language called CSQL (Conditional SQL).

The structure of this demonstration paper is as follows: Section 2 provides the features and functionalities of the system, Section 3 provides a brief software architecture and implementation overview, and Section 4 describes the demonstration process.

2. FEATURES OF POSSDB

The PossDB system has system specific operations and functions related to conditional tables. As a running example we will use two conditional tables: Employees and

Departments, with the schemas shown in Table 1 and Table 2 respectively. Let us suppose that we have two universities merging, and one university has a Employees table which contains Firstname, Lastname and Department, while the Employees table of the other university has only Lastname and Department. When we merge these two tables we will have null values in the Firstname column for those tuples that came from the second university's Employee table. Suppose that the first university has an employee in a CS department with a Firstname John and Lastname Smith and other university has an employee in CS department with Lastname Smith. Assuming that full names are unique, we know that these two employees are distinct. The university has decided to retain only one of these employees in the merged department, but we don't yet know which one. Thus the two Smith tuples are mutually exclusive, which is captured by their local conditions. Additionally, there is a third employee in the Math department of the second university, for whom we know that his or her Firstname is not Newman.

Table 1: Employees

Firstname	Lastname	Dept	Condition
John	Smith	CS	$x = John$
x	Smith	CS	$x \neq John$
y	Jones	Math	$y \neq Newman$

In the same merging example we assume that universities haven't yet decided the location of the Physics department, except that it will not be located on Campus C. In the tables an empty Condition value represents a condition that is always *true*.

Table 2: Departments

Dept	Location	Condition
CS	Campus A	
Math	Campus B	
z	Campus C	$z \neq Physics$

2.1 Operations

Selection. The select statement generalizes the standard SQL select statement. By the generalization the select statement will be able to return a conditional table when querying an existing conditional table. The new extended select statement also does optimizations on the resulting local conditions. If the local condition is a tautology the condition is replaced with *true* and when the local condition is a contradiction the corresponding tuple will not be part in the result. Let us consider the following example that returns the employees with the name John:

```
SELECT *
FROM EMPLOYEES
WHERE FIRSTNAME = 'John';
```

Running the query results in the following conditional table:

Firstname	Lastname	Dept	Condition
John	Smith	CS	$x = John$
~~x~~	~~Smith~~	~~CS~~	~~$x = John \wedge x \neq John$~~
y	Jones	Math	$y \neq Newman \wedge y = John$

The second row is automatically eliminated from the result since the local condition is a contradiction, and the tuple will not be present in any complete instance represented by the table.

Projection. The projection operation is implemented, as expected, with the use of Select statement. The system will automatically merge duplicate tuples by considering the disjunction of the two existing local conditions. Consider the following query:

```
SELECT LASTNAME, DEPT
FROM EMPLOYEES
```

The result will be

Lastname	Dept	Condition
Smith	CS	~~$x = John \vee x \neq John$~~ *true*
Jones	Math	$y \neq Newman \wedge y = John$

The local condition for the first tuple is replaced by *true* because it is a tautological formula after merging the formulae from both tuples with the same SURNAME and DEPT. Thus, the end user will not see the tautological local conditions.

Join. The operation joins two or more conditional tables and return a new conditional Table. To join the conditional tables Employees and Departments we write the following query:

```
SELECT EMPLOYEES.NAME,
       DEPARTMENTS.DEPT,
       DEPARTMENTS.LOCATION
  FROM EMPLOYEES
 INNER JOIN DEPARTMENTS ON
 EMPLOYEES.DEPARTMENT = DEPARTMENTS.DEPARTMENT
```

The resulting conditional table is:

Firstname	Dept	Location	Condition
John	CS	Campus A	$x = John$
John	z	Campus C	$x \neq John \wedge$ $z \neq Physics \wedge z = CS$
x	CS	Campus A	$x \neq John$
x	z	Campus C	$x \neq John \wedge$ $z = CS \wedge z \neq Physics$
y	Math	Campus B	$x \neq John$
y	z	Campus B	$y = Newman \wedge$ $z = Math \wedge z \neq Physics$

It can be noted that in the join case the local conditions for each resulted tuple is a conjunction of the local conditions of the tuples that contributed by join to that tuple.

Create Table Conditional tables are defined using the same syntax used to create regular tables in DBMS. Thus, the following statement will create a conditional table:

```
CREATE TABLE DEPARTMENTS(
   DEPT,
   LOCATION
)
```

Materialized View In our system we allow materialization of queries over conditional tables. This means that the system will create a new conditional table with the given schema and with the tuples and conditions given by the result of the given select statement. For example, if one needs to create a materialized view that contains all the tuples and associated local conditions returned by `SELECT * FROM EMPLOYEES WHERE DEPT='CS'` will run the following statement that will materialize the result as a new conditional table named 'CS_EMPLOYEES'

```
CREATE TABLE CS_EMPLOYEES(
       NAME,
       SURNAME,
       DEPT) FOR
SELECT *
  FROM EMPLOYEES
 WHERE DEPT = 'CS'
```

Insert We extended the standard SQL Insert statement by allowing the users to also specify a local condition associated with the inserted tuple. The null values needs to be specified as the null label preceded by the # sign (given this constraint the system does not allow the # as part of regular strings). Thus, the string "#x" specifies a new null value labeled as "x". This notation is necessary in order for the system to be able to delimit the labeled nulls from regular text information. In case the CONDITION clause is not specified in the INSERT statement, by default we consider the local condition tautological, that is *true*. The following example shows the syntax used to insert the tuple (x,"Smith","CS") with the local condition x='John' in the Employees table (note how the labeled null 'x' is specified):

```
   INSERT INTO EMPLOYEES
   VALUES ('#x','Smith','CS')
CONDITION ('#x = John')
```

The Boolean expression used in the local condition are constructed based on the equality (=) and inequality (!=) atoms connected by the conjunction (AND) and disjunction (OR) operator. For example the expression
$x = John \wedge (y \neq Smith \vee y \neq Brown)$, where x and y are variables, is expressed by the following string:

```
#x='John' AND (#y!='Smith' OR #y!='Brown')
```

2.2 Functions

We have two new functions defined in PossDB. These functions are used to query for certainty and possibility of a tuple in a conditional table.

IS POSSIBLE(<Tuple>, Conditional Table Name) Takes a tuple "Tuple" and decides if the tuple is possible in the conditional table given by the conditional table name. Intuitively a tuple is possible in a given conditional table if there exists a valuation for the conditional table that contains that tuple. The "Tuple" has to be in a format such Name of the Column 1, Value of the Column 1, Name of the Column 2, Value of the Column 2, ... If the tuple is possible in the given conditional table it returns true otherwise it returns false.

As an example onsider the following function call:

```
IS POSSIBLE(<'Name','John', 'Surname','Smith',
            'Dept','CS'>, 'Employees')
```

In our example this function returns true, because given tuple is possible in the system, it is not certain because it depends on the condition $x = John$.

IS CERTAIN(<Tuple>,Conditional Table Name) It takes tuple and name of the conditional table as a parameter as the same way IsPossible does. Certain means that the tuple does not depend on any condition, this functions determines if the tuple is certain or not in the given table. If we ask to the system if a tuple is certain in the table, we write:

```
IS CERTAIN (<'Surname','Smith', 'Dept','CS'>,
            'Employees')
```

This function returns true, because given tuple doesn't depend on any condition, that's why it is certain.

3. IMPLEMENTATION

3.1 Implementation Overview

For the conditional tables we used regular relational tables and added one more text data typed column that will store the local condition associated with the tuple specified by the other columns to each table. The data types of the tables are user defined data types called c_varchar (Conditional varchar). The only difference between regular character types and c_varchar is that c_varchar can store variables. If a c_varchar data type contains a text which starts with an number sign (#) this data type treats it as a variable. In order to cope with the variables we had also to override the equality operator. If the left-hand or right-hand of the equality is variable, the equality is automatically satisfied, otherwise the regular equality operator does its normal job. Consequently, without variables the PossDB system works as a regular RDBMS. If the equality has a variable the system adds the equality to the corresponding local condition, and sends the local condition to java method to check if the local condition is satisfiable, a tautology, or a contradiction. In order to check for satisfiability, the local condition is converted to DNF (Disjunctive Normal Form)

$$\underbrace{(E_1 \wedge E_2 \wedge E_3)}_{\text{First Disjunct}} \vee \underbrace{(E_4 \wedge E_5 \wedge \ldots)}_{\text{Second Disjunct}} \vee ....$$

Here E_i denotes an equality or inequality. If one of the disjunct in DNF is satisfiable, we do not need to go over every disjunct Only if the Boolean combination is contradiction we have to make sure that all the disjuncts are contradictions. In this case the java method returns *false* and DBMS eliminates that tuple. For checking if a local condition is a tautology we have to make sure that it is satisfiable first, that's why checking for tautology comes after checking the satisfiability. For checking if a local condition is a tautology is easier if it is in CNF (Conjunctive Normal Form):

$$\underbrace{(E_1 \vee E_2 \vee E_3)}_{\text{First Conjunct}} \wedge \underbrace{(E_4 \vee E_5 \vee \ldots)}_{\text{Second Conjunct}} \wedge \ldots.$$

If one of the conjuncts in the CNF formula is not tautology, it means the CNF formula is not tautology. If the CNF formula is a tautology, every conjunct has to be inspected to see if it is a tautology. In order to reduce the run-time and complexity, we don't convert the DNF formula into CNF formula directly, instead we generate CNF conjuncts one by one from the DNF. If a generated conjunct is not tautology we terminate the process because it means that the formula is not a tautology. If the generated conjunct is a tautology, we continue to generate conjuncts until a non-tautological one is encountered. If no generated conjunct is a tautology it means that the whole formula is a tautology.

As an example, suppose we have a DNF formula:

$$(x{=}\mathrm{A} \wedge y{=}\mathrm{B}) \vee (x \neq \mathrm{A} \wedge y{=}\mathrm{D} \wedge z{=}\mathrm{E})$$

First, we take the first conjunct from the first disjunct of the DNF formula and the first conjunct of the second disjunct, forming the first conjunct $(x = A) \vee (x \neq A)$ of the CNF formula. Since $(x = A) \vee (x \neq A)$ indeed is a tautology we continue by forming the second conjunct $(x = A) \vee (y = D)$, which is not a tautology. Thus we can terminate the processing of the local condition and return the fact that the local condition is not a tautology.

After the Satisfiability and Tautology java method finishes its duty, PossDB prepares the result of the query by adding some annotations in the conditional column which our Java GUI engine can understand and PossDB returns it to the GUI engine. The GUI engine generates the output stream to the user, or to a graphical interface. In general the PossDB system works on Postgres with user defined functions. When the user types CSQL query, the system generates a related SQL query by using the user defined functions. The SQL query is then run on Postgres.

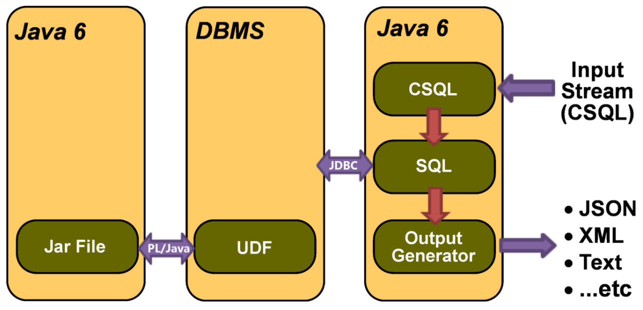

Figure 1: System Workflow

3.2 System Architecture

PossDB system is built as a system on top of Postgres RDBMS. We used PL/Java to introduce the user defined functions. Java$^{\mathrm{TM}}$6 and open source java sql parser ZQL is being used for parsing the conditions and than evaluating them. On the application layer Java is being used for parsing and converting user input and output. This java application is working with input and output streams that's why it can be easily ported to the any kind of application server or it can be simply used through a console. The connection between the java middle tier and PostgreSQL database server is done through JDBC.

4. DEMONSTRATION

For the demonstration we will use two data sets. One small dataset, similar with the one used in this paper, is used to clearly demonstrate the system capabilities. The second set contains large conditional tables. This large data was generated by us with the help of a data generator that distributes nulls in the table in various ways. The second data set is intended to show the scalability of the system.

Demonstration Steps

1. **Basics** We begin by introducing basic concepts such as conditional tables, how the local conditions work, how variables are represented. We then demonstrate a number of Select-Project-Join queries, the insertion of tuples into the tables, and the creation of materialized views.

2. **PossDB related Functions** We continue by demonstrating the Is Possible and Is Certain function functions.

3. **Large Data** The last part of the demonstration will be on very large data set with lots of missing information, in order to show the scalability of our system.

5. REFERENCES

[1] G. Grahne and A. Onet. Closed world chasing. In *Proceedings of the 4th International Workshop on Logic in Databases*, LID '11, pages 7–14, New York, NY, USA, 2011. ACM.

[2] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[3] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: a probabilistic database management system. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *SIGMOD Conference*, pages 1071–1074. ACM, 2009.

[4] T. Imielinski and W. Lipski. Incomplete information in relational databases. *J.ACM*, 31(4):761–791, September 1984.

[5] J. Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data.* Springer, 2008.