Conditional Tables in practice

Gösta Grahne, Adrian Onet, and Nihat Tartal

Concordia University, Montreal, QC, H3G 1M8, CANADA,
{grahne,a_onet,m_tartal}@cs.concordia.ca

Abstract. Due to the ever increasing importance of the internet, interoperability of heterogeneous data sources is as well of ever increasing importance. Interoperability can be achieved e.g. through data integration and data exchange. Common to both approaches is the need for the DBMS to be able to store and query *incomplete databases*. In this report we present PossDB, a DBMS capable of storing and querying incomplete databases. The system is wrapper over PostgreSQL, and the query language is an extension of a subset of standard SQL. Our experimental results show that our system scales well, actually better than comparable systems.

1 Introduction

Management of uncertain and imprecise data has long been recognized as an important direction of research in data bases. With the tremendous growth of information stored and shared over the Internet, and the introduction of new technologies able to capture and transmit information, it has become increasingly important for Data Base Management Systems to be able to handle uncertain and probabilistic data. As a consequence, there has lately been significant efforts by the database research community to develop new systems able to deal with uncertainty, either by annotating values with probabilistic measures (e.g. MystiQ [7], Orion [15], BayesStore [16], PrDB [14], MayBMS-2 [3]) or defining new structures capable of capturing missing information (e.g. Trio [17] and MayBMS-1 [4]).

Uncertainty management is an important topic also in data exchange and information integration. In these scenarios the data stored in one database has to be restructured to fit the schema of a different database. The restructuring forces the introduction of "null" values in the translated data, since the second schema can contain columns not present in the first. In the currently commercially available relational DBMS's the missing or unknown information is stored with placeholder values denoted `null`. It is well known that this representation has drawbacks when it comes to query answering, and that a logically coherent treatment of the `null` is still lacking from most DBMS's.

To illustrate the above mentioned drawbacks, consider a merger of companies "Acme" and "Ajax." Both companies keep an employee database. Let *Emp*1(*Name,Mstat,Dept*) and *Emp*2(*Name,Gender, Mstat*), where *Mstat* stands for marital status, be the schemas used by Acme and Ajax, respectively. The merged company decides to use the schema *Emp*(*Name,Gender,Mstat,Dept*), and it is known that all the employees from Ajax will work under the same department, which will either be 'IT' or 'PR'. Consider now the initial data from both companies:

Emp1

Name	Mstat	Dept
Alice	married	IT
Bob	married	HR
Cecilia	married	HR

Emp2

Name	Gender	Mstat
David	M	married
Ella	F	single

The merged company database instance would be represented as the following database in a standard relational DBMS:

Emp

Name	Gender	Mstat	Dept
Alice	`null`	married	IT
Bob	`null`	married	HR
Cecilia	`null`	married	HR
David	M	married	`null`
Ella	F	single	`null`

With this incomplete database consider now the following two simple queries:

Q_1:
```
SELECT Name FROM Emp WHERE
(Gender = 'M' AND Mstat = 'married') OR Gender = 'F'
```

Q_2:
```
SELECT E.Name, F.Name FROM Emp E, Emp F
        WHERE E.Dept=F.Dept AND E.Name != F.Name
```

Clearly one would expect that the first query to return all employee names and the second query to return the set of tuples $\{(\textit{Bob, Cecilia}), (\textit{David, Ella})\}$. Unfortunately by the default way null values are treated in standard systems the tuples returned by the first query would return the set $\{(\textit{David, Ella})\}$ and by the second query would return the set $\{(\textit{Bob, Cecilia})\}$.

In this report we introduce a new database management system called PossDB (Possibility Data Base) able to fully support incomplete information. The purpose of the PossDB system is to demonstrate that scalable processing of semantically meaningful null values is indeed possible, and can be built on top of a standard DBMS (PostgreSQL, in our case).

Irrespectively of how an incomplete database instance $\mathcal{I}$ is represented, conceptually it is a (finite or infinite) *set* of possible complete database instances I (i.e. databases without null values), denoted $Poss(\mathcal{I})$. Each $I \in Poss(\mathcal{I})$ is called a *possible world* of $\mathcal{I}$. A query Q over a complete instance I gives a complete instance $Q(I)$ as answer. For incomplete databases there are three semantics for query answers:

1. *The exact answer*. The answer is (conceptually) a set of complete instances, each obtained by querying a possible world of $\mathcal{I}$, i.e. $\{Q(I) : I \in Poss(\mathcal{I})\}$.

The answer should be represented in the same way as the input database, e.g. as a relation with meaningful nulls.

2. *The certain answer.* This answer is a complete database containing only the (complete) tuples that appear in in the query answer in *all* possible worlds. In other words, $Cert(Q(\mathcal{I})) = \bigcap_{I \in Poss(\mathcal{I})} Q(I)$.

3. *The possible answer.* $Poss(Q(\mathcal{I})) = \bigcup_{I \in Poss(\mathcal{I})} Q(I)$.

The PossDB system is based on conditional tables (c-tables) [11] which generalize relations in three 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, and it represents the *same* unknown value wherever it occurs. A c-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. Each valuation v then gives rise to a possible world $v(T)$. The second generalization is that each tuple t is associated with a *local condition* $\varphi(t)$, which is a Boolean formula over equalities between constants and variables, or variables and variables. The final generalization introduces a *global condition* $\Phi(T)$, which has the same form as the local conditions. In obtaining complete instances from a table T, we consider only those valuations v, for which $v(\Phi(T))$ evaluates to *True*, and include in $v(T)$ only tuples $v(t)$, where $v(\varphi(t))$ evaluates to *True*.

In our previous example the merged incomplete database would be represented as the following c-table.

Emp

Name	Gender	Mstat	Dept	$\varphi(t)$
Alice	x_1	married	IT	*True*
Bob	x_2	married	HR	*True*
Cecilia	x_3	married	HR	*True*
David	M	married	x_4	*True*
Ella	F	single	x_4	*True*

The global condition Φ(Emp) is $(x_i =$ 'M'$) \vee (x_i =$ 'F'$)$, for $i = 1, 2, 3$, and $(x_4 =$ 'IT'$) \vee (x_4 =$ 'PR'$)$. Under this interpretation PossDB will return the expected results for both queries. Note that in this example the exact, possible, and certain answers are the same.

The c-tables support the full relational algebra [11], and are capable of returning the possible, the certain and the exact 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. The exact answer of a query Q on a c-table T is a c-table $Q(T)$ such that $v(Q(T)) = Q(v(T))$, for all valuations v.

C-tables are the oldest and most fundamental instance of a *semiring-labeled* database [10]. By choosing the appropriate semiring, labeled databases can model a variety of phenomena 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 other semiring based databases.

To the best of our knowledge, PossDB is the first implemented system based on c-tables. In the future we plan to extend our system to support the Conditional Chase [8], a functionality which is highly relevant in data exchange and information integration.

In order to gauge the scalability of our system, we have run some experiments comparing the performance of PossDB with MayBMS-1 [4]. The MayBMS-1 system uses a representation mechanism called *World Set Decompositions*, which is fundamentally different from c-tables. For details we refer to [4]. Similarly to PossDB, the MayBMS-1 system is build on top of PostgreSQL, an open source relational database management system. In the case where there is no incomplete information, both PossDB and MayBMS-1 work exactly like classical DBMS's. However, at this point we have restricted, similarly to MayBMS-1, our data to be encoded as positive integers. In the future we will extend the allowed data types to include all the base data types.

In this current stage PossDB allows the following operations:
(1) : Creation of c-tables,
(2) : Querying c-tables,
(3) : Inserting into c-tables,
(4) : Materializing c-tables representing the exact answers to queries, and
(5) : Testing for tuple possibility and certainty in c-tables.

All these operations are expressed using an extension of the ANSI SQL language called C-SQL (Conditional SQL).

2 Features of PossDB

The PossDB system has system specific operations and functions related to c-tables. To illustrate these operations, let us continue with the example from the previous section. The global condition in our example is $\Phi(Emp) =_{\mathsf{def}} \{(x_i = \text{'M'} \vee x_i = \text{'F'}) : i = 1, 2, 3\} \cup \{x_4 = \text{'IT'} \vee x_4 = \text{'PR'}\}$. This set corresponds to a CNF formula, where each disjunct contain all possible values for a given variable. It is stored in a hash structure such that for each variable the hash function will return all possible values for that variable. This representation speeds up the processing then checking for contradictory and tautological local conditions.

Next, we present the operations of the PossDB system. Note that none of these operations affect the global condition.

Relational Selection The select statement generalizes the standard SQL select statement. The generalized select statement will work on c-tables rather than relations with `null`'s. Beside returning the exact answer, the select statement also optimizes the c-table by removing tuples t, where $\varphi(t) \wedge \Phi(T)$ is a contradiction, and replacing with *true* local conditions of tuples t, where $\Phi(T)$ implies $\varphi(t)$.

Consider e.g. the query that returns all employee from the 'IT' department:

```
SELECT * FROM Emp WHERE Dept = 'IT';
```

The query results in the following c-table:

Name	Gender	Mstat	Dept	$\varphi(t)$
Alice	x_1	married	IT	*True*
David	M	married	x_4	$x_4 =$ 'IT'
Ella	F	single	x_4	$x_4 =$ 'IT'

Note that the query returns a representation of the exact answer, that is a c-table that represents the set of all possible answer instances. This pertains to all query operations in the PossDB system.

Relational Projection operation is implemented, as expected, as an extension of the `SELECT` statement.

Relational Join The join and cross product operations work similarly with their standard SQL counterparts. For example consider the following project-join query that returns all pairs of employee names that work in the same department such that the first employee is male and the second employee is female:

```
SELECT e1.Name as Name1, e2.Name as Name2
FROM Emp e1 INNER JOIN Emp e2 ON e1.Dept=e2.Dept
WHERE e1.Gender='M' AND e2.Gender='F'
```

The exact answer for this query is:

Q

Name1	Name2	$\varphi(t)$
Alice	Ella	$x_1 =$ 'M' $\wedge x_4 =$ 'IT'
Bob	Cecilia	$x_2 =$ 'M' $\wedge x_3 =$ 'F'
~~Bob~~	~~Ella~~	~~$x_2 =$ 'M' $\wedge x_4 =$ 'HR'~~
Cecilia	Bob	$x_3 =$ 'M' $\wedge x_2 =$ 'F'
~~Cecilia~~	~~Ella~~	~~$x_3 =$ 'M' $\wedge x_4 =$ 'HR'~~
David	Alice	$x_1 =$ 'F' $\wedge x_4 =$ 'IT'
~~David~~	~~Bob~~	~~$x_2 =$ 'F' $\wedge x_4 =$ 'HR'~~
~~David~~	~~Cecilia~~	~~$x_3 =$ 'F' $\wedge x_4 =$ 'HR'~~
David	Ella	*True*

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 were joined, and the condition induced by the "where" clause in the select statement. The over-striked tuples are deleted by the system, since they have local conditions that are not satisfiable together with the global condition. Also note that the local condition for the last tuple is a tautology as both employees "David" and "Ella" share the same variable as department.

Insertion C-SQL extends the standard SQL Insert statement by allowing the users to also specify a local condition associated with the inserted tuple. In case the `CONDITION` clause is not specified in the `INSERT` statement, by default we consider the local condition tautological, i.e. *True*. The following example shows the syntax used to insert the tuple (Smith, M, single, x) with the local condition "x =HR or x =PR" in the Emp table.

```
INSERT INTO Emp
VALUES ('Smith','M','single',x)
CONDITION (x ='HR' or x ='PR')
```

Query Answers We return the exact answer as a c-table. This is comparable with MayBMS-1 that returns all the tuples that can occur in the query answer or some complete instance corresponding the input database. This has the drawback that the answer may contain two mutually exclusive tuples. On the other hand PossDB returns a c-table representing the exact answer. In some cases this c-table might have convoluted local conditions, and it might be difficult for the user to understand the structure. In order to overcome this, we have included two additional functions `IS POSSIBLE` and `IS CERTAIN`. Both functions work in polynomial time.

Special functions We have two new functions unique to PossDB. These functions are used to query for certainty and possibility of a tuple in a c-table or in the result of a query.

```
IS POSSIBLE(Tuple) IN C-Table Name | Query
```

The `IS POSSIBLE` is a Boolean function takes a tuple `Tuple` and decides if the tuple is possible in the c-table given by the `C-Table Name` or in the result of the given `Query`. Intuitively a tuple is possible in a given c-table if there exists a valuation for the c-table that contains that tuple. The "Tuple" has to be specified as a list of (Name, Value) pairs. As an example consider the following function call:

```
IS POSSIBLE(Name, 'Bob', Gender, 'M',
       Mstat, 'married', Dept, 'HR') IN Emp
```

With the data from the previous example the `IS POSSIBLE` function returns *true*, because given tuple is possible in the system. However, it is not certain because it depends on the condition (x_2 = 'M').

```
IS CERTAIN(Tuple) IN C-Table Name | Query
```

Similarly to `IS POSSIBLE` the `IS CERTAIN` function takes as parameter a tuple, and a c-table name or query and returns *True* if the tuple is certain in the given c-table or the result of the given query. Certain means that the tuple appears under all possible interpretations of the nulls. The following is an example of the usage of the `IS CERTAIN` function

```
IS CERTAIN (Name, 'Bob', Dept, 'HR') IN
```

```
SELECT Name, Dept FROM Emp
```

This function returns *True*, because given tuple appear under any interpretation for the nulls. Note also that the function would return *False* if we also included the Gender column in the query.

This could be extended so that the user could ask if a *set* of tuples is possible or certain. Thus we could also determine whether two possible tuples are mutually exclusive, by issuing `IS POSSIBLE` t_1, `IS POSSIBLE` t_2, and `IS POSSIBLE` $\{t_1, t_2\}$. If the first two answers are $True$ and the third answer is $False$, it means that both t_1 and t_2 are possible tuples, but they are mutually exclusive (i.e. cannot co-exist in the same possible world). We note that the `IS CERTAIN` would still run in polynomial time in this generalization, as would also the `IS CERTAIN` function, provided the number of tuples in the set were fixed [2].

3 Implementation

Without loss of generality, the information in our conditional tables are encoded as integers. Positive integers denote constants and negative integers denote nulls. Consequently, without variables, the PossDB system works as a regular RDBMS, and the performance in this case will be the same as that of PostgreSQL. With this encoding we need to be sure that the queries are properly evaluated. Thus, each equality condition of the form $A = c$ part of a C-SQL query, where A is a column name and c a constant, is rewritten as $(A = c \vee A < 0)$ in SQL. This is necessary in order to check that the column A is either constant c or that it represents a null value, here encoded as negative integers. In order to check for satisfiability of a local conditions and its conjunction with the global condition, the local conditions are converted into DNF (Disjunctive Normal Form). To make a faster satisfiability test we store the global condition as hash based representation of its CNF (Conjunctive Normal Form).

After the Satisfiability and Tautology checks, the system decides which tuple to show in the result of the query by adding some annotations in the local condition column. Our application has a GUI capable of generating the query result in a human readable interface. From a technical perspective PossDB is a two-layer system, the application layer built in Java and as a database layer it uses PostgreSQL database engine. When the user types a C-SQL query, the system interprets it and execute it by a series of SQL statements against the database and a series of Java calls needed to make sure that the c-tables are correctly manipulated and displayed to the user.

System Architecture. PossDB system is built on top of PostgreSQL. On the middle tier Java® and ANTLR [12] are being used. ANTLR is used to parse the C-SQL queries and database conditions, while Java is used to implement the C-SQL processing part, displaying the results, evaluating conditions, and connecting to the PostgreSQL database server.

This Java application is working with input and output streams, hence it can be easily ported to the any kind of application server or simply used through a console. The connection between the Java middle tier and PostgreSQL database server is done through JDBC.

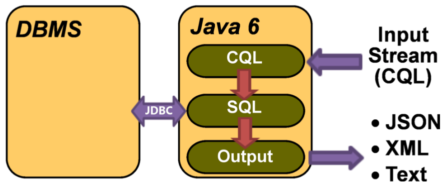


Fig. 1. System Workflow

4 Experimental Results

In order to check the scalability of our system we wanted to compare it with another system that has the most similar features. Thus we compared PossDB with MayBMS-1, as MayBMS-1 also returns the exact answer to queries, and the scalability of MayBMS has been proven [4]. Furthermore, both PossDB and MayBMS-1 are built on top of PostgeSQL.

Our experiments are based on the queries and data which were used for the MayBMS-1 experimental evaluation [4]. Their experiment used a large census database encrypted as integers [13]. Noise was introduced by replacing some values with variables that could take between 2 and 8 possible values. A noise ratio of $n\%$ meant that $n\%$ of the values were perturbed in this fashion. In our experiments we used to same data and noise generator as MayBMS-1. The MayBMS-1 system and the noise generator were obtained from [1].

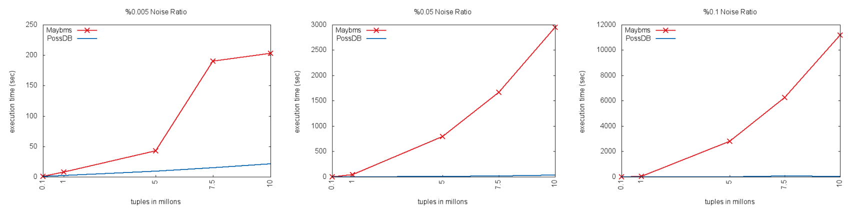
We tested both systems with up to 10 million tuples. The charts below contain the result of the test using queries Q_1 and Q_2 from the experiments in [4]. The results show that PossDB clearly outperforms MayBMS-1.

System configuration. We conducted all our experiments on Intel®Core$^{\text{TM}}$i5-760 processor machine with 8 GB RAM, running Windows 7 Enterprise and PostgreSQL 9.0.

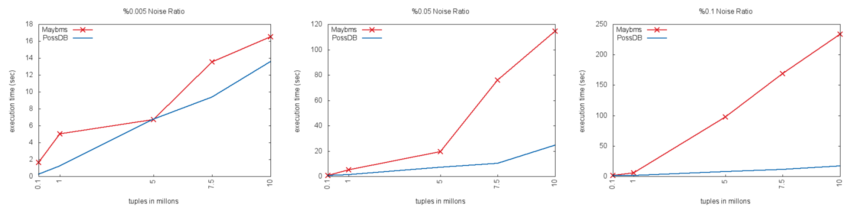
5 Conclusions & Further Work

The system presented here is capable to store incomplete data using Conditional Tables. This structure, even if well known, has not been implemented before, although many probabilistic systems essentially use probabilistic versions of c-tables.

In this report we show not only that the conditional table is a good candidate for storing incomplete information but we also show that that the system indeed is scalable. For now PossDB is able to process positive queries. We are in the



Q1: `SELECT * FROM R WHERE VETSTAT = 8 AND CITIZEN = 9`



Q2: `SELECT STATEFIP,OCC1990,CITIZEN,SUBFAM FROM R`
`WHERE STATEFIP = OCC1990 AND CITIZEN = 1 AND SUBFAM > 4`

process of extending the system to allow general SQL queries, including also certain/possible nested subqueries. This requires non-trivial extensions to the current C-SQL language. Another extension is to integrate a state-of-the-art SAT-solver, e.g. [5] or [6]. The SAT-solver would then handle the satisfiability and tautology tests, which is likely to further improve the performance of the system. Finally, we will also implement the chase based procedure on conditional tables [9] in order for the new system to be also usable in other applications, such as Data Exchange, Data Repair and Data Integration.

References

1. Maybms-1 system and the noise generator. `http://pdbench.sourceforge.net/`.
2. S. Abiteboul, P. C. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):158–187, 1991.
3. L. Antova, C. Koch, and D. Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.
4. L. Antova, C. Koch, and D. Olteanu. 10ˆ10ˆ6 worlds and beyond: efficient representation and processing of incomplete information. *The VLDB Journal*, 18(5):1021–1040, Oct. 2009.
5. G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, pages 399–404, 2009.
6. A. Biere. Preprocessing and inprocessing techniques in sat. In *Haifa Verification Conference*, page 1, 2011.
7. J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. Mystiq: a system for finding more answers by using probabilities. In *SIGMOD Conference*, pages 891–893, 2005.
8. 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.

9. G. Grahne and A. Onet. Closed world chasing. In *LID*, pages 7–14, 2011.
10. 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.
11. T. Imielinski and W. Lipski. Incomplete information in relational databases. *J.ACM*, 31(4):761–791, September 1984.
12. T. J. Parr, T. J. Parr, and R. W. Quong. Antlr: A predicated-ll(k) parser generator, 1995.
13. S. Ruggles. Integrated public use microdata series: Version 3.0, 2004.
14. P. Sen, A. Deshpande, and L. Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.
15. S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. The orion uncertain data management system. In *COMAD*, pages 273–276, 2008.
16. D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.
17. J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical Report 2004-40, Stanford InfoLab, August 2004.