POSSDB: AN UNCERTAINTY DATA MANAGEMENT SYSTEM BASED ON CONDITIONAL TABLES

Mustafa Nihat Tartal

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2013

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Mustafa Nihat Tartal**
Entitled:	**PossDB: An Uncertainty Data Management System based on Conditional Tables**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair

______________________________ Examiner

______________________________ Examiner

______________________________ Supervisor

Approved ____________________________________

Chair of Department or Graduate Program Director

__________ 20 _____ ______________________________

Rama Bhat, Ph.D.,ing., FEIC, FCSME, FASME, Interim Dean

Faculty of Engineering and Computer Science

Acknowledgements

I would like to thank my supervisor, Prof. Gösta Grahne, without his constant encouragement, support, understanding and attentive guidance this thesis would not have been completed successfully.

I would also like to express my sincere gratitude to Dr. Adrian Onet, he has provided me lots of information and never hesitated to help, which are precious to me.

Finally, I would like to express my eternal appreciation towards my parents and family members who have always there for me no matter where I am. Thank you for being ever so understanding and supportive.

Abstract

Due to the ever increasing importance of the Internet, interoperability of heterogeneous data sources is as well of ever increasing importance. Interoperability could be achieved for instance through data integration and data exchange. Common to both approaches is the need for the database management system to be able to store and query *incomplete databases*. In this thesis we present PossDB, a database management system capable of storing and querying incomplete databases. The system is a 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.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AID	Alternative Identifier
C-SQL	Conditional Structured Query Language
C-Tables	Conditional Tables
CNF	Conjunctive Normal Form
DBMS	Data Base Management System
DNF	Disjunctive Normal Form
TID	Tuple Identifier
ULDB	Uncertainty Lineage Database
XID	X-Tuple Identifier

Chapter 1

Introduction

1.1 Data Uncertainty and Incomplete Information

Management of uncertain and incomplete 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 (DBMS) 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 or defining new structures capable of capturing missing information.

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.

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 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.* This answer is also a complete database, containing the tuples that appear in the answer to the query *in at least* one possible

world. $Poss(Q(\mathcal{I})) = \bigcup_{I \in Poss(\mathcal{I})} Q(I)$.

1.2 The PossDB System and Conditional Tables

This thesis introduces 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.

The PossDB system is based on conditional tables (c-tables) [24] 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*.

The c-tables support the full relational algebra [24], 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 [19]. 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.

1.3 Motivation

Over the past years the growth of information shared over the Internet reached immense volumes, but unfortunately without having a common schema, data integration became a huge problem. Researchers are trying to find new data models to deal with uncertain information. Unfortunately a lot of research paper claim that the c-tables have not found application in practice. The c-tables appeared in 1984 and have not been implemented, researchers tend to ignore c-tables and try to find new approaches. To the best of our knowledge, PossDB is the first implemented system based on c-tables. Our goal in this thesis is to show that the c-tables can be a data model for a scalable uncertainty management system and show that the c-tables do have applications in practice.

1.4 Running Example

This thesis uses the relations defined below for a running example. Let us assume that there are two companies merging and each one of has a different schema given below.

- **Company 1:** $Emp1$(Name, Marital Status, Dept)
- **Company 2:** $Emp2$(Name, Gender, Marital Status)

The merged company decides to use the schema given below:

$$Emp(\text{Name, Gender, Marital Status, Dept})$$

It is known that in the merged company, all the employees from Company 2 will work under the same department, which will either be 'IT' or 'PR'. Now consider the initial data from both companies:

Emp1

Name	Marital Status	Dept
Alice	married	IT
Bob	married	HR

Emp2

Name	Gender	Marital Status
David	M	married
Ella	F	single

In a standard relational DBMS the instance of the merged company database would be represented as given below table:

Emp

TID	Name	Gender	Marital Status	Dept
1	Alice	`null`	married	IT
2	Bob	`null`	married	HR
3	David	M	married	`null`
4	Ella	F	single	`null`

In order to keep track of the tuples, the tuple id (TID) column is added to the *Emp* relation.

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

Q:
```
Select Name From Emp Where
(Gender = ’M’ AND Marital Status = ’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
```

The expected answer from the first query is to return all employee names, because it is a known fact that a gender of a person can be either male or female, since two employees in the *Emp* relation are married they will satisfy the condition in any cases, other tuples have no unknown data in their gender and marital status columns, hence they satisfy the where condition. The expected answer from the second query is to return the tuple $\{(\textit{David, Ella})\}$, because we know that those employees are coming from the *Company* 2 and it is a known that they will work in the same departement. Unfortunately by the default way null values are treated in standard systems the first query returns the set $\{(\textit{David, Ella})\}$ and the second query would return the an empty set.

The given *Emp* relation above can be represented by using variables instead of nulls. The representation of the *Emp* relation by using variables instead of null values is given below:

Emp

TID	Name	Gender	Marital Status	Dept
1	Alice	x_1	married	IT
2	Bob	x_2	married	HR
3	David	M	married	x_3
4	Ella	F	single	x_4

In the *Emp* relation above, the variables can be assigned to constant values, possible constant values for the variables are given below:

- $x_1 = \{M, F\}$
- $x_2 = \{M, F\}$
- $x_3 = \{IT, PR\}$
- $x_4 = \{IT, PR\}$

Note that by defining the possible constant values, the domain becomes finite, in order to achive an infinite domain, the possible valuations should not be given.

In this thesis 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.

1.5 Thesis Organization

The rest of the thesis is organized as follow: In chapter 2, we introduce the related work and their data models. In chapter 3, we introduce the conditional tables and the PossDB system with its features. In chapter 4, we introduce the query language of the PossDB system and its implementation on top of PossDB system. In chapter 5, we introduce the algorithms that are used to implement conditional tables over the relational database management system. In chapter 6, experimental results show the performance of our system with the performance of other comparable systems. In chapter 7, our conclusions and recommendations for future works are presented.

Chapter 2

Related Work

This chapter surveys previous work in data uncertainty and incomplete information. There has lately been significant efforts by the database research community to develop new data models able to deal with uncertainty. These efforts include not only the theoretical solutions but also the practical system implementations.

2.1 Probabilistic Approach

In order to deal with incomplete information tuples have been annotated with probabilistic measures. Each tuple has a probability which ranges between 0 and 1. Probability 1 means the tuple is a certain tuple in the relation and probability 0 means that tuple should not be in the relation, it can be ignored as if it never existed. The probabilistic representation of our running example is given below:

Emp

TID	Name	Gender	Marital Status	Dept	Probability
1	Alice	M	married	IT	0.5
1	Alice	F	married	IT	0.5
2	Bob	M	married	HR	0.5
2	Bob	F	married	HR	0.5
3	David	M	married	IT	0.5
3	David	M	married	PR	0.5
4	Ella	F	single	IT	0.5
4	Ella	F	single	PR	0.5

Note that probabilities are assumed to be uniformly distributed as the same weight for the each possible world. The important part of this approach is grouping the tuples, where each group has a probability value 1. As it can be seen from the example above, tuples are grouped by the *TID* column, each group has a probability value 1 in sum. In the possible worlds only one tuple can show up from each group. Each tuple has 2 possibilities, since we have 4 tuples, the number of possible worlds are $2^4 = 16$. The challenging part of this approach is representing correlated tuples. Correlated tuples can occur when existence of a tuple depends on the existence of another tuple. As an example let us assume that the employees *David* and *Ella* will work in the same department in the new merged company. In this case there exists a correlation between *Ella* and *David*, which is difficult to express in the probabilistic approach.

2.1.1 Probabilistic Systems

Orion (previously known as U-DBMS) [13] is an extended relational DBMS with uncertainty management functionalities which has built-in support for probabilistic data. The main purpose of the system is to provide uncertainty management for constantly evolving data, such as temperature, pressure or location data. Orion uses an uncertainty data structure where each tuple has an uncertainty attribute with 2 elements:

- An uncertainty interval
- An uncertainty probability distribution function

An uncertainty interval is a value for a uncertain constantly evolving data, that defines the upper and lower bounds of the uncertain data. An uncertainty probability distribution function, uses the distribution of the known data to identify the unknown data distribution. An example uncertainty probability distribution function is the Gaussian distribution or the Uniform distribution, which models the measurement inaccuracy of temperature data. Orion supports both attribute and tuple uncertainty with arbitrary correlations.

MystiQ [12] is a system that uses probabilistic approach to find answers in large number of heterogeneous data sources. Unlike the other implemented probabilistic systems, MystiQ does not have a data structure to store data in a structured way, instead it has query semantics to query multiple data sources and answers the queries by adding the probabilities of the tuples appearing in the result [14]. It provides a powerful means to query inconsistent data across multiple data sources. Mainly MystiQ is a working prototype for a new querying paradigm over multiple resources which causes uncertain and incomplete data. BayesStore [31] and PrDB [30] is aimed to capture the uncertainties which have complex correlations among each other that appear in real-world application domains. To achieve that goals, both systems are based on the most popular uncertainty modeling technique called probabilistic graphical models [5] developed by the statistics and machine learning communities.

Trio [4] and MayBMS-2 [22] are also probabilistic systems but they combine probabilistic approach with other approaches, they will be explained in the section 2.3.1 and 2.4.1.

2.2 World-Set Decomposition

The complexity of the probabilistic approach let researchers investigate on new approaches in data uncertainty and incomplete information field. World-set decomposition [26] is one of the accomplished and efficient approach. The approach is based on relational product decomposition. A world-set is basically a relation where each row represents a possible world. The world-set decomposition decomposes the world-set relation into several relations such that their cartesian product gives us the world-set relation. For each world-set a unique representation exists and it can be efficiently computed [26]. The world-set relation representation of our running example is:

(TID)	Name	Gender	Marital Status	Dept
1	Alice	{M,F}	married	IT
2	Bob	{M,F}	married	HR
3	David	M	married	{IT,PR}
4	Ella	F	single	{IT,PR}

When we decompose it we get:

$$\begin{array}{|c|}\hline t_1.\text{Name}\\\hline \text{Alice}\\\hline\end{array}\times\begin{array}{|c|}\hline t_1.\text{Gender}\\\hline \text{M}\\ \text{F}\\\hline\end{array}\times\begin{array}{|c|}\hline t_1.\text{Marital Status}\\\hline \text{married}\\\hline\end{array}\times\begin{array}{|c|}\hline t_1.\text{Dept}\\\hline \text{IT}\\\hline\end{array}\times\begin{array}{|c|}\hline t_2.\text{Name}\\\hline \text{Bob}\\\hline\end{array}\times$$

$$\begin{array}{|c|}\hline t_2.\text{Gender}\\\hline \text{M}\\ \text{F}\\\hline\end{array}\times\begin{array}{|c|}\hline t_2.\text{Marital Status}\\\hline \text{married}\\\hline\end{array}\times\begin{array}{|c|}\hline t_2.\text{Dept}\\\hline \text{HR}\\\hline\end{array}\times\begin{array}{|c|}\hline t_3.\text{Name}\\\hline \text{David}\\\hline\end{array}\times\begin{array}{|c|}\hline t_3.\text{Gender}\\\hline \text{M}\\\hline\end{array}\times$$

$$\begin{array}{|c|}\hline t_3.\text{Marital Status}\\\hline \text{married}\\\hline\end{array}\times\begin{array}{|c|}\hline t_3.\text{Dept}\\\hline \text{IT}\\ \text{PR}\\\hline\end{array}\times\begin{array}{|c|}\hline t_4.\text{Name}\\\hline \text{Ella}\\\hline\end{array}\times\begin{array}{|c|}\hline t_4.\text{Gender}\\\hline \text{F}\\\hline\end{array}\times$$

$$\begin{array}{|c|}\hline t_4.\text{Marital Status}\\\hline \text{single}\\\hline\end{array}\times\begin{array}{|c|}\hline t_4.\text{Dept}\\\hline \text{IT}\\ \text{PR}\\\hline\end{array}$$

The number of possible worlds can be calculated by multiplying the number of tuples from each decomposed relations. In the example above we have $1 \times 2 \times 1 \times 1 \times 1 \times 2 \times 1 \times 1 \times 1 \times 1 \times 1 \times 2 \times 1 \times 1 \times 1 \times 2 = 16$ possible worlds. When some correlation exists between tuples, it is easy to represent in the decomposed world-set representation. Let us assume again *David* and *Ella* will work under the same department. In this case we can merge t_3.Dept and t_4.Dept together, after the merge we get:

t_1.Name
Alice

×

t_1.Gender
M
F

×

t_1.Marital Status
married

×

t_1.Dept
IT

×

t_2.Name
Bob

×

t_2.Gender
M
F

×

t_2.Marital Status
married

×

t_2.Dept
HR

×

t_3.Name
David

×

t_3.Gender
M

×

t_3.Marital Status
married

×

t_3.Dept	t_4.Dept
IT	IT
PR	PR

×

t_4.Name
Ella

×

t_4.Gender
F

×

t_4.Marital Status
single

Since our decomposition changed, the number of possible worlds changed as well. When we calculate the new number of possible worlds we get $1 \times 2 \times 1 \times 1 \times 1 \times 2 \times 1 \times 1 \times 1 \times 1 \times 1 \times 2 \times 1 \times 1 \times 1 = 8$ possible worlds.

2.2.1 World-Set Decomposition Systems

The system that uses world-set decomposition model is called MayBMS. Since there are two MayBMS versions available, the one that uses the world-set decomposition is called MayBMS-1 [7]. MayBMS-1 is built on World Set Decompositions in theory, but in practice, there are some differences. Because of database systems do not support relations of arbitrary arity, MayBMS-1 uses a structure called "Uniform World Set Decompositions". Uniform World Set Decompositions has a fixed schema which stores all possible values. That fixed schema contains 2 relations.

1. F(<u>Relation</u>, <u>TID</u>, <u>Attribute</u>, Component ID)

   The relation F stores the mapping between the tuple fields and component identifiers. Note that the underlined attributes creates an unique key.

2. C(Component ID, Local World ID, Value)

   The relation C stores each value from component together with its local world identifiers. In order to find a component Relation Name, TID and Attribute Name is needed. Local World ID identifies the possible worlds and Value is an ID of the given local possible world.

The Uniform World Set Decomposition representation of our running example is given below:

Emp

TID	Name	Gender	Marital Status	Dept
1	Alice	`null`	married	IT
2	Bob	`null`	married	HR
3	David	M	married	`null`
4	Ella	F	single	`null`

F

Relation	TID	Attribute	Component ID
Emp	1	Gender	C1
Emp	2	Gender	C2
Emp	3	Dept	C3
Emp	4	Dept	C4

C

Component ID	Local World ID	Value
C1	1	M
C1	2	F
C2	1	M
C2	2	F
C3	1	IT
C3	2	PR
C4	1	IT
C4	2	PR

The Uniform World Set Decomposition can be seen as an extension of the Or-tables [23]. MayBMS-1 also can be extended to probabilistic database easily by adding one more relation to the fixed schema which maps the probabilities to the local worlds.

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, MayBMS-1 works exactly like classical DBMS's. The biggest disadvantage of the MayBMS-1 system is each operation needs to join with F and C relations, even if there is a simple select operation, join needs to be performed behind the scenes.

2.3 X-Relations

X-Relations [29] is a specific formalism for uncertain databases, it is also known as ULDBs (Uncertainty Lineage Databases) data model. X-Relations are comprised of x-tuples where x-tuples consist of one or more alternatives. Since ULDB relations are comprised of x-tuples it is called as x-relations. Unlike the probabilistic approach, in the x-relations a tuple can represent one or more possible worlds. Another important property of an x-tuple is that it can be annotated by a maybe (?) annotation, which shows that the uncertainty of the tuple even though each attributes in the tuple has only one valuation. The representation of our running example in x-relations is given below:

Emp

TID	Name	Gender	Marital Status	Dept	
1	Alice	M || F	married	IT	?
2	Bob	M || F	married	HR	?
3	David	M	married	IT || PR	?
4	Ella	F	single	IT || PR	?

As it can be seen from the above example, the first and the second tuple contains alternatives in their *Gender* attribute, also the third and the fourth tuple contains alternatives in their *Dept* attribute. This approach is clearly more space efficient approach than probabilistic approach but unfortunately it does not provide an efficient structure for correlated tuples.

2.3.1 X-Relation Systems

Trio [4] [32] is a DBMS which combines data, uncertainty, and lineage. Trio uses ULDB data model (X-Relations) which is explained in the previous section. Trio manages uncertainty with probabilistic measures but it also has lineage functionality which makes Trio different than other implemented probabilistic systems. By the lineage functionality of the system, it is also possible to keep track of where the data is derived from. Trio is implemented on top of PostgreSQL DBMS and x-relations are represented in the relational tables. Trio supports a SQL-based query language called TriQL. TriQL queries given by the user are converted to SQL queries automatically by the Trio system. The core system is implemented in Python. The Trio system can be used with probability or without probability. Trio system converts ULBS to the relational model by using two identifiers. One of them is AID (globally unique alternative identifier), which identifies the alternatives with a unique id and the other one is xid (x-tuple identifier), which identifies the tuple with a unique tuple id. It can be noted that even though the theory of the ULDB seems practical, implementing that theory on top of recent DBMS requires more work.

2.4 U-Relations

U-Relations [6] represent uncertainty on the attribute level, u-relations decomposes attribute level uncertainty vertically. Basically u-relations consist of discrete independent (random) variables, a tuple id column, a set of variable and value assignment($V \mapsto D$), and a set of value columns. In u-relations each possible world defined by the values assigned to the variables. We can represent our running example in u-relations as given below:

$U_{Emp[Name]}$

$V \mapsto D$	TID	Name
	1	Alice
	2	Bob
	3	David
	4	Ella

$U_{Emp[Gender]}$

$V \mapsto D$	TID	Gender
$x \mapsto 1$	1	M
$x \mapsto 2$	1	F
$y \mapsto 1$	2	M
$y \mapsto 2$	2	F
	3	M
	4	F

$U_{Emp[MaritalStatus]}$

$V \mapsto D$	TID	Marital Status
	1	married
	2	married
	3	married
	4	single

$U_{Emp[Dept]}$

$V \mapsto D$	TID	Dept
	1	IT
	2	HR
$z \mapsto 1$	3	IT
$z \mapsto 2$	3	PR
$t \mapsto 1$	4	IT
$t \mapsto 2$	4	PR

In order to represent possible worlds, we need to choose a valuation for each variable. As an example one possible world is given by the valuation $\{x \mapsto 1, y \mapsto 2, z \mapsto 1, t \mapsto 1\}$.

Let us assume again *David* and *Ella* will work under the same department, in this case we can use the same variable z for the tuple 4 instead of using variable t. Our new decomposed $U_{Emp[Dept]}$ table will be:

$U_{Emp[Dept]}$

$V \mapsto D$	TID	Dept
	1	IT
	2	HR
$z \mapsto 1$	3	IT
$z \mapsto 2$	3	PR
$z \mapsto 1$	4	IT
$z \mapsto 2$	4	PR

2.4.1 U-Relation Systems

MayBMS-2 [22] is also a probabilistic database implementation which uses the U-Relations. MayBMS-2 has the same system architecture as MayBMS-1 but it uses a different data model than MayBMS-1. MayBMS-2 is aimed to develop as a probabilistic database by supporting complex probabilistic measures unlike MayBMS-1 which only supports naive probabilistic approach. MayBMS-2 has has its own query language called I-SQL [8]. I-SQL is designed for managing uncertain and incomplete information and is an extension of SQL language.

2.5 Other Systems

There are other systems not covered in detail in this thesis, which are considered as uncertain database systems but their purpose is more different than our problem domain. These systems are MauveDB [15], MCDB [25]. Their main purpose is censor data management [20], where the data occur repetitively.

2.6 Summary

From the prior works explained in this chapter, it becomes apparent that there is growing interest in incomplete information and uncertainty management. Each approach explained in this chapter has its own unique characteristics, but all of them serve the same purpose. All the explained models require some work to implement on top of a DBMS. Since the implementation requires more work, the final product needs more knowledge to do operations on it.

In the light of these facts, we are proposing the conditional tables for uncertainty management, which can be adapted to recent DBMS's with minor extensions. C-Tables require less space than any of the other approaches explained above. Since the space efficiency plays a huge role in execution time, the uncertainty management system based on c-tables could be faster than other approaches.

Chapter 3

The PossDB System

The PossDB system is a DBMS system based on conditional tables. The PossDB system has notions, system specific operations, and functions related to c-tables. This chapter explains these features. To illustrate these features, our running example will be used.

3.1 Conditional Tables

Conditional Tables(c-tables)[24][17] have characteristics given below:

- In the entries in the columns, variables, representing unknown values, are allowed in addition to the usual constants.
- Each tuple t is associated with a *local condition* denoted by $\varphi(t)$, which is a Boolean formula over equalities between constants and variables, or variables and variables.
- Database contains a Boolean formula called *global condition* which is com-

mon to a individual c-tables in the database, since the same variable can occur in several tables. *Global condition* is denoted by $\Phi(T)$.

- 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 valuations in a c-table $(v(T), v(\varphi(t)))$ must not be contradictory to the *global condition*.

3.2 The Global Condition

The *Global Condition* is associated with the entire database. Even though the global condition in our example is written as $\Phi(Emp)$, it is associated with the entire database instead of associating with only the *Emp* relation. The *Global Condition* consists of a Boolean formula. In the PossDB system the formula is stored as CNF formula, which allows the system to make faster satisfiability check along with the *Local Conditions*.

The global condition in our running example is:

$$\varphi(Emp) =_{\mathsf{def}} \{(x_i = \text{'M'} \lor x_i = \text{'F'}) : i = 1, 2\} \cup \{x_3 = \text{'IT'} \lor x_3 = \text{'PR'}\}$$

This set corresponds to a CNF formula, where each conjunct contain all possible values for a given variable.

The CNF form of our global condition is given below:

$$\Phi(Emp) = (x_1 = \text{'M'} \vee x_1 = \text{'F'}) \wedge (x_2 = \text{'M'} \vee x_2 = \text{'F'}) \wedge (x_3 = \text{'IT'} \vee x_3 = \text{'PR'})$$

It can be clearly seen that each variable occupies one of the conjuncts of the CNF formula, and in each disjunct in each conjunct gives the valuation for that variable. For instance the variable x_2 occupies the second conjunct of the CNF formula.

$$\underbrace{(x_1 = \text{'M'} \vee x_1 = \text{'F'})}_{First\ Conjunct} \wedge \underbrace{(x_2 = \text{'M'} \vee x_2 = \text{'F'})}_{Second\ Conjunct} \wedge \underbrace{(x_3 = \text{'IT'} \vee x_3 = \text{'PR'})}_{Third\ Conjunct}$$

The valuations for the varaible x_2 are stored in the disjuncts of the second conjunct of the CNF formula.

$$(\underbrace{x_2 = \text{'M'}}_{First\ Disjunct} \vee \underbrace{x_2 = \text{'F'}}_{Second\ Disjunct})$$

Since the CNF formula is formed of conjuncts which each of them are related to one unique variable, the CNF formula can be stored in a hashed structure. Storing the CNF formula in a hashed structure speeds up the process of evaluating satisfiability and tautology. For each variable the hash function will return all the possible values for that variable. Although the hashed global condition structure seems restricted, it fulfills all the requirements in an uncertainty management system. The PossDB system depends on this hashed global condition structure and it gains its power from this structure.

In our running example the merged incomplete relation *Emp* would be represented as the following c-table with the global condition given below.

$$\Phi(Emp) = (x_1 = \text{'M'} \vee x_1 = \text{'F'}) \wedge (x_2 = \text{'M'} \vee x_2 = \text{'F'}) \wedge (x_3 = \text{'IT'} \vee x_3 = \text{'PR'})$$

Emp

TID	Name	Gender	Marital Status	Dept	$\varphi(t)$
1	Alice	x_1	married	IT	$True$
2	Bob	x_2	married	HR	$True$
3	David	M	married	x_3	$True$
4	Ella	F	single	x_3	$True$

3.3 C-Table Creation

C-Table creation is almost the same as table creation in recent DBMS. The PossDB system automatically adds the local condition column to each relation whenever a new relation is created. That local condition column stores the local conditions in string data type.

Example 3.1 *Creating the Emp relation in our running example.*

The user should create the relation with given schema below:

$$Emp(\text{Name, Gender, Marital Status, Dept})$$

but in the PossDB, created relation will have the schema given below:

$$Emp(\text{Name, Gender, Marital Status, }\underline{\text{Condition}})$$

Note that the column *Condition* automatically added to the relation by the system.

3.4 Selection

Select(σ) operator for the c-tables returns the tuples with the their local conditions from the given relation. Select statement in the PossDB not only returns the exact answer, but 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 the formula $\Phi(T) \rightarrow \varphi(t)$ is a tautology.
A local condition $\varphi(t)$ in the result of a select statement is the conjunction of the local condition and the select condition. Let say for the tuple t, the local condition is λ and the select statement is: $\sigma_\theta(R)$, where θ is a Boolean formula, in this case $\varphi(t) = \theta(t) \wedge \lambda$. Note that $\theta(t)$ is the select condition(θ) where all the attribute names are replaced with the attribute values in that tuple t.

Example 3.2 *The query that returns all employees from the 'IT' department*

$$\sigma_{Dept='IT'}(Emp)$$

The query results is in the following c-table:

TID	Name	Gender	Marital Status	Dept	$\varphi(t)$
1	Alice	x_1	married	IT	$True$
~~2~~	~~Bob~~	~~x_2~~	~~married~~	~~HR~~	~~$False$~~
3	David	M	married	x_4	$x_4 =$ 'IT'
4	Ella	F	single	x_4	$x_4 =$ 'IT'

As it can be seen above, during the evaluation t_1 has a local condition $\varphi(t_1) =$ ('IT' $=$ 'IT'$\wedge True$) since 'IT' $=$ 'IT' returns $True$, the new local condition keeps its value $\varphi(t_1) = True$.
For tuple t_2 the local condition is: $\varphi(t_2) =$ ('HR' $=$ 'IT'$\wedge True$), since 'HR' $=$ 'IT' returns $False$, $\varphi(t_2) = (False \wedge True) = False$, which represents a contradiction, hence t_2 is removed from the result. Similarly for t_3 and t_4, the same steps are applied, and as a result the new local conditions for t_3 and t_4 are set to local conditions in the result.

3.5 Projection

The projection operation has the same property of table creation in the PossDB system. It works as projection in recent DBMS. The projection operation returns the local condition with the projected columns.

Example 3.3 *Projecting the Emp relation with attributes Name and Gender in our running example.*

$$\pi_{Name,Gender}(Emp)$$

In the PossDB, the result will have the schema given below:

$$Emp(Name, Gender, Condition)$$

3.6 Join

The join and cross product operations work similarly to their standard SQL counterparts. The local conditions for each resulted tuple is a conjunction of the local conditions of the tuples that contributed by join, the condition induced by the select condition of the select statement and the join condition. Resulted local conditions are being checked for satisfiability and tautology. If the local condition is a contradiction, it is removed from the result and if it is a tautology, the local condition is replaced by $True$.

Example 3.4 *The following Project-Join query that returns all pairs of names of employees that work in the same department such that the first employee is a male and the second employee is a female. Note that the / operator is the renaming operator.*

$$C_1 \leftarrow \pi_{Name1/Name,Gender1/Gender,Dept1/Dept}$$

$$C_2 \leftarrow \pi_{Name2/Name,Gender2/Gender,Dept2/Dept}$$

$$\pi_{Name1,Name2}(\sigma_{Dept1=Dept2 \wedge Gender1='M' \wedge Gender2='F'}(C_1 \times C_2))$$

The exact answer for this query is:

TID	Name1	Name2	$\varphi(t)$
1	Alice	Ella	$x_1 =$ 'M' $\wedge x_4 =$ 'IT'
~~2~~	~~Bob~~	~~Ella~~	~~$x_2 =$ 'M' $\wedge x_4 =$ 'HR'~~
3	David	Alice	$x_1 =$ 'F' $\wedge x_4 =$ 'IT'
~~4~~	~~David~~	~~Bob~~	~~$x_2 =$ 'F' $\wedge x_4 =$ 'HR'~~
5	David	Ella	*True*

Unlike the example 3.2, second and the fourth tuples are eliminated because of the global condition. The conjunction of the local condition and the global condition produce contradiction.

For t_2,

$$\varphi(t) \wedge \Phi(T) = (x_2 = \text{'M'}) \wedge \underbrace{x_4 = \text{'HR'} \wedge (x_4 = \text{'IT'} \vee x_4 = \text{'PR'})}_{Contradiction} \wedge (x_2 = \text{'M'} \vee \ldots).$$

For t_4,

$$\varphi(t) \wedge \Phi(T) = x_2 = \text{'F'} \wedge \underbrace{x_4 = \text{'HR'} \wedge (x_4 = \text{'IT'} \vee x_4 = \text{'PR'})}_{Contradiction} \wedge (x_2 = \text{'M'} \vee \ldots).$$

Also the local condition for t_5 is a tautology as both employees “David” and “Ella” share the same variable as department even though the name of the department is unknown. The local condition $x_4 = x_4$ produces *True*.

3.7 Insertion

Insertion as expected inserts a tuple with the local condition, the local condition, which can be empty or in other words *True*. The important function of the insert operation is evaluating and converting the local condition. In the first step the given local condition is being evaluated. If it is satisfiable, in the next step it is being checked for tautology. If it is tatutology, the local condition is replaced by $True$. If the local condition is not a tautology but satisfiable then the local condition is converted into Disjunctive Normal Form (DNF) and stored in that format for faster process in the future operations. If the given local condition is not satisfiable, the given insert statement will be ignored by the PossDB.

3.8 Special Functions

The PossDB system returns the exact answer as a c-table. This has the drawback that the answer may contain two mutually exclusive tuples. 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, the PossDB system has two new functions *Is Possible* and *Is Certain*. These functions are used to query for certainty and possibility of a tuple in a c-table.

3.8.1 Is Possible

One of the unique functionality of the PossDB system is, checking the possibility of a tuple in a database instance. *Is Possible* function in the PossDB takes a tuple from the user and decides if the tuple is possible in a c-table. Intuitively a tuple is possible in a given c-table if there exists a valuation for the c-table that contains that tuple.
To check if the given tuple if it is possible in a c-table, the PossDB system takes values given in a tuple and decides if they are possible or not.

Example 3.5 *Check if the following tuple is possible in the **Emp** relation in our running example.*

Name	Gender	Marital Status	Dept
David	M	married	IT

In order to check the possibility, the PossDB system checks all given attribute values in the *Emp* c-table. First it retrieves all the data from the *Emp* which satisfies the *Name = 'David' And Gender = 'M' And Marital Status = 'married' And Dept = 'IT'* condition. Since there is only one tuple which satisfies that condition, the PossDB only considers that tuple in the following steps. Note that in the first step the system automatically satisfies the variable constant equations, in this example the system satisfies the equation x_3 = 'IT' because it contains a variable. The next step is checking the variables. Since the *Dept* attribute contains a variable, the PossDB system needs to check if the given constant value satisfies the variable stored in the *Dept*. The stored value in the *Dept* attribute is x_4, hence x_4 = 'IT' needs to be check against the global and the local condition. Since the local condition is empty it is considered as $True$. The conjunction of global and the local condition is: $\varphi(t) \wedge \Phi(T) = True \wedge (x_1 = \text{'M'} \vee x_1 = \text{'F'}) \wedge (x_2 = \text{'M'} \vee x_2 = \text{'F'}) \wedge (x_3 = \text{'M'} \vee x_3 = \text{'F'}) \wedge (x_4 = \text{'IT'} \vee x_4 = \text{'PR'})$
The Boolean formula $(x_4 = \text{'IT'}) \wedge \varphi(t) \wedge \Phi(T)$ is satisfiable, hence the tuple given above is possible in the *Emp* c-table.

3.8.2 Is Certain

Similarly to *Is Possible* function the *Is Certain* function takes a tuple as a parameter, and returns $True$ if the tuple is certain in the given c-table. Certain means that the tuple appears under all possible interpretations of the nulls. An easy example of a certain tuple in *Emp* relation is:

Name	Gender
David	M

Under any interpretation of nulls, that tuple appears, hence the given tuple is a certain tuple in our c-table. Even though it seems that certainty depends on constants to constants mapping from given tuple to a c-table, it is not the case. Let us consider there is a name and surname database and in the database there is a name that we are not sure if it is written as $Denis$ or $Dennis$. One way of representing this uncertainty is given below with a global condition $True$.

TID	Name	Surname	Condition
1	Denis	Brown	$x = 1$
2	Dennis	Brown	$x \neq 1$

If there exist a correlation with those tuples to other tuples the variable x can be used. Whenever $x = 1$ is being used in some other tuple, it means the existence of that tuple depends on existence of the tuple with TID 1. Albeit it is not the best representation, it is a valid c-table representation, hence it makes the certainty check more complicated than possibility check. Let us assume the tuple:

Name	Surname
Denis	Brown

Since each attribute value of the tuple maps with a constant in the c-table, it does not cause a certainty, because it depends on a condition $x = 1$. Nevertheless, not all the conditions yield uncertainty, there might be some conditions which yield certainty. Let us consider the c-table below with a global condition $True$:

TID	Name	Surname	Department	Condition
1	George	Costanza	y	$y =$ 'Sales'
2	George	Costanza	y	$y \neq$ 'Sales'

In the given c-table below there exist 2 employees who have the same name but one of them probably works in the *sales* department, and the other one works in a department other than *sales*. When we check the certainty of a tuple below:

Name	Surname
George	Costanza

When the system checks the certainty, it retrieves the c-table given below:

Name	Surname	Condition
George	Costanza	$y =$ 'Sales'
George	Costanza	$y \neq$ 'Sales'

Both tuples depend on a condition, but an alternative representation of the above c-table is:

Name	Surname	Condition
George	Costanza	$y =$ 'Sales' $\vee\, y \neq$ 'Sales'

As it can be seen easily $y =$ 'Sales' $\vee\, y \neq$ 'Sales' generates $True$, hence the given tuple is certain tuple in the given c-table.

3.9 System Architecture

The PossDB system is built on top of PostgreSQL DBMS. On the middle tier Java® is being used. Java is used to implement the query 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.

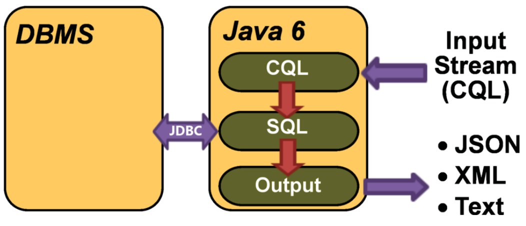

Figure 3.1: System Workflow

Chapter 4

C-SQL

This chapter describes the language that is used to operate the PossDB system. The language is called "C-SQL", which is the abbreviation of *Conditional Structured Query Language*. C-SQL is an extension of the ANSI SQL. The goal of extending the ANSI SQL language is to provide the PossDB users a familiar and adaptable language environment. Since the C-SQL is an extension of the ANSI SQL, a standard DBMS user can easily adapt to C-SQL language with a little effort. The C-SQL is a language which is being converted to an SQL behind the scenes and the converted language is executed on the DBMS. The query conversion from CSQL to SQL is called as translation rule in this thesis. This chapter will explain the query language for conditional tables with the translation rules from C-SQL to SQL.

Translation from CSQL to SQL is achieved by using the ANTLR parser generator [27] and to parse Boolean formulas we used the ZQL parser [2]. Note that the PossDB system is currently based on integer numbers, constant values are represented as positive integer numbers and variables are represented as negative integers, hence the C-SQL is implemented as typeless language, the type integer is automatically completed by the system if necessary during the conversion.

4.1 Table Creation

The *Create Table* statement is used to create a c-table in the PossDB system. Since the PossDB system is an uncertainty management system, c-tables are the only tables can be created in the system.

C-SQL Create Table Syntax:

```
CREATE TABLE <Table_Name>(
    <Column_1>,
    <Column_2>,
    ⋮,
    <Column_n>
)
```

In order to illustrate creating a c-table with C-SQL let us consider the given example below:

Example 4.1 *Creating the Emp relation as a c-table in our running example.*

```
CREATE TABLE Emp(
    Name,
    Gender,
    Marital Status,
    Dept
)
```

The translation rule for the create table from C-SQL to SQL is given below:

```
CREATE TABLE <Table_Name>(  →  CREATE TABLE <Table_Name>(
   <Column_1>,              →     <Column_1> integer,
   <Column_2>,              →     <Column_2> integer,
   ⋮                        →     ⋮
   <Column_n>               →     <Column_n> integer,
                                  Condition text
)                           →  )
```

As it can be seen above, the column type integer is being added to the column names and as a last column the *Condition* column is being added to the newly created table. During the translation process the C-SQL statement turned into an SQL statement, which can be executed on the PostgreSQL.

4.2 Selection

The *Select* statement is used to select data from the c-table in the PossDB system.

C-SQL Select Syntax:

```
Select *
  From <Table_Name>
 Where <Select_Condition>
```

The Where section in the syntax can be omitted from the syntax. It is being used to apply condition to the select statement, the <Select_Condition> should have one of the structure which are defined below.

Note that supported operators in the PossDB system are; the equality operator

"=" and the unequality operator "≠".

- <Column_Name> <Operator> <Column_Name>
- <Column_Name> <Operator> <Constant>
- <Constant> <Operator> <Column_Name>
- <Constant> <Operator> <Constant>
- Disjuncts or Conjuncts of any items in this list

In order to illustrate selecting from a c-table with C-SQL let us consider the example below:

Example 4.2 *Select all the attributes of the employees from the Emp relation, who has the name "Alice" or "Bob", and who works under the 'IT' department.*

```
Select *
  From Emp
 Where (Name = 'Alice' OR Name = 'Bob')
   And Dept = 'IT'
```

The translation rule for the selection from C-SQL to SQL is given below.

```
Select *                      →  Select *
  From <Table_Name>           →    From <Table_Name>
 Where <Select_Condition>     →   Where <New_Select_Condition>
```

The translation rule above introduces a new notion *<New_Select_Condition>*, this is the select condition for the c-tables. Since c-tables may contain variables represented as negative integers, the PossDB system needs to retrieve those tuples to validate against the global condition. In order to retrive those tuples the select condition should be extended. The negative integers are variables and the positive integers are constants in the PossDB system, thus the following steps need apply to the each predicate in the select condition.

- If there is a column A in relation R and the condition is $R(A) = a$, where a is a constant value, replace it with $R(A) = a \vee R(a) < 0$
- If there is a column A and B in relation R and there exist an equality such $R(A) = R(B)$, replace it with $(R(A) = R(B)) \vee (R(A) < 0) \vee (R(B) < 0)$

As an example let assume that we have a select condition as $Age = 24$, the given condition should be replaced by $(Age = 24) \vee (Age < 0)$, this condition gives us all the tuple which has 24 in their age attribute, but also it gives us the tuples which has a variable in their age attribute.

4.3 Projection

The project operation is implemented, as expected, as an extension of the `SELECT` statement. It can be used to select some attributes instead of selecting all the attributes from the c-tables.

C-SQL Project Syntax:

```
Select <Column_1>,
       <Column_2>,
       ⋮
       <Column_n>
  From <Table_Name>
 Where <Select_Condition>
```

Example 4.3 *Select all the employee names who works in the 'IT' department.*

```
Select Name
  From Emp
 Where Dept = 'IT'
```

The translation rule for the projection from C-SQL to SQL is given below:

```
Select <Column_1>,           →  Select *
       <Column_2>,
       ⋮
       <Column_n>
  From <Table_Name>          →    From <Table_Name>
 Where <Select_Condition>    →   Where <New_Select_Condition>
```

In the translation rule above, in order to select all the columns, the selected columns are replaced with the asterisk (*) symbol. The reason is that there might be some conditions in the select condition depends on the non-selected columns. In order to process the variables in the non-selected column, we need

to retrieve the data from those columns to process them. Also it will retrieve the *Condition* column as well. After the variable checking the projection will be applied as a last step.
The <*Select_Condition*> is replaced by <*New_Select_Condition*>, which was explained in section 4.2.

4.4 Join

The join and cross product operations work similarly with their standard SQL counterparts. The PossDB system only supports inner joins.

C-SQL Join Syntax:

```
Select <Table_Name_(1-2)>.<Column_1>,
       <Table_Name_(1-2)>.<Column_2>,
       ⋮
       <Table_Name_(1-2)>.<Column_n>
  From <Table_Name_1>
 Inner Join <Table_Name_2> ON
       <Join_Condition>
       ⋮
 Where <Select_Condition>
```

Since Join statement is an extended version of the Select statement, all the rules apply to Select statement also apply to the join statement. The important difference between the Join statement and the Select statement is each selected column needs to be specified with the table name.

In order to illustrate join operation with C-SQL let us consider the example below:

Example 4.4 *Retrieve all the pair of employee names that work in the 'IT' department.*

```
Select Emp1.Name as Name1,
       Emp2.Name as Name2
  From Emp Emp1
 Inner Join Emp Emp2 ON
       Emp1.Name != Emp2.Name
 Where Emp2.Dept = 'IT'
```

The translation rule for the projection from C-SQL to SQL is given below:

```
Select                               →  Select
  <Table_Name_(1-2)>.<Column_1>,     →    <Table_Name_1)>.*,
  <Table_Name_(1-2)>.<Column_2>,     →    <Table_Name_(2)>.*
  ⋮                                  →   Merge_Condition(
  <Table_Name_(1-2)>.<Column_n>      →     <Table_Name_1>.Condition,
                                           <Table_Name_2>.Condition
                                         ) As Condition
  From <Table_Name_1>                →    From <Table_Name_1>,
 Inner Join <Table_Name_2> ON        →        <Table_Name_2>
       <Join_Condition>              →   Where <New_Select_Condition>
 Where <Select_Condition>            →     And <New_Join_Condition>
```

The translation rule above uses the asterisk symbol, for the same reason as the *Project* operation explained in the section 4.3.

The translation rule above introduces three new notions:

1. **<Join_Condition>**

   It is used to join two c-tables with given columns, it is the same join condition in the standard SQL.

2. **<New_Join_Condition>**

   It is the extended <*Join_Condition*> for c-tables, which is is extended as the same way as the <*Select_Condition*> extended to the <*New_Select_Condition*>.

3. **<Merge_Condition()>**

   It is a user defined function coded in the PostgreSQL, which takes any number of parameters as a string and outputs the conjunction of the given Boolean formulas. Let us assume that given parameters are P, Q and R, the output of the function depending on the given parameters is $P \wedge Q \wedge R$.

4.5 Insertion

The *Insert* statement is used to insert data into a c-table in the PossDB system. 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 the PossDB system considers the local condition tautological, i.e. $True$.

C-SQL Insert Syntax:

```
Insert Into <Table_Name>
Values (<Column_1_Value>
   <Column_1_Value>,
   <Column_2_Value>,
   ⋮
   <Column_n_Value>)
Condition (<Local_Condition>)
```

The <*Local_Condition*> in the syntax refers to the local condition associated with the inserted tuple. In order to illustrate inserting a tuple to a c-table with C-SQL let us consider the example below.

Example 4.5 *Insert the Employee "Ella" to the Emp relation in our running example.*

```
Insert Into Emp
Values ('Ella','F','single')
```

In the given example above, since there is no local condition, the condition statement is not being used. The C-SQL statement below is the equivalent statement as the above C-SQL statement.

```
Insert Into Emp
Values ('Ella','F','single')
Condition (True)
```

The translation rule for the Insert statement from C-SQL to SQL is given below:

```
Insert Into <Table_Name>          →  Insert Into <Table_Name>
Values (<Column_1_Value>,         →  Values(<Column_1_Value>
        <Column_2_Value>          →          <Column_2_Value>
           ⋮                      →             ⋮
        <Column_n_Value>)         →          <Column_n_Value>,
Condition(<Local_Condition>)      →          <New_Local_Condition>)
```

The translation rule above introduces new notion <*New_Local_Condition*>, it is the DNF formula of the local condition given by the user. If the `CONDITION` keyword has not been used in the syntax, <*New_Local_Condition*> considered as $True$.

4.6 Tuple Possibility

The *Is Possible* statement is used to trigger the tuple possibility function on c-tables. The *Is Possible* function takes a tuple and a c-table as a parameter, the c-table parameter can be given in two ways

1. A Select Statement
2. A C-Table Name

C-SQL Is Possible Syntax:

```
Is Possible(<Column_1>,<Column_1_Value>,
            <Column_2>,<Column_2_Value>,
            ⋮
            <Column_n>,<Column_n_Value>)
In <C-Table>
```

The <*C-Table*> in the syntax can be written as a c-table name or as a select statement, which returns a c-table. In order to illustrate *Is Possible* function in the PossDB system let us consider the example below:

Example 4.6 *Is there exist a male employee who works under the 'HR' department and named as 'Bob' in the Emp relation of our running example.*

```
Is Possible (Name, 'Bob', Gender, 'M', Dept, 'HR')
In Emp
```

The C-SQL statement below is the equivalent statement as the above C-SQL statement.

```
Is Possible (Name, 'Bob', Gender, 'M', Dept, 'HR')
In Select * From Emp
```

The translation rule for the *Is Possible* statement from C-SQL to SQL is given below:

```
Is Possible(<Column_1>,           →  Select * From <C-Table>
            <Column_1_Value>,         Where(
                                     (<Column_1> = <Column_1_Value>
                                     Or <Column_1> < 0)
                                     And
            <Column_2>,           →  (<Column_2> = <Column_2_Value>
            <Column_2_Value>,        Or <Column_2> < 0)
                                     And
                 ⋮                →   ⋮
            <Column_n>,           →  (<Column_n> = <Column_n_Value>
            <Column_n_Value>         Or <Column_n> < 0)
In <C-Table>                      →  )
```

Since the PossDB system based on integers, and the variables are encoded as negative integers, the given syntax above is sufficient for retrieving all the tuples which satisfy the conditions or contain variables. Please note that converted SQL statement is not enough for possibility checking, more process is required to check if the tuples satisfy the global condition, necessary processes will be explained in chapter 5.

4.7 Tuple Certainty

The *Is Certain* statement is used to trigger the tuple certainty function on c-tables. It has the same syntax as the *Is Possible* statement, only difference is it answers the certainty instead of possibility.

C-SQL Is Certain Syntax:

```
Is Certain(<Column_1>,<Column_1_Value>,
           <Column_2>,<Column_2_Value>,
           ⋮
           <Column_n>,<Column_n_Value>)
In <C-Table>
```

Since the *Is Certain* statement has the same syntax as the *Is Possible* statement, let us consider a different and more complicated example than the example described in the section 4.6. Consider the example below:

Example 4.7 *In the Emp relation in our running example, is there exist a certain data which states that, a single employee works in a department, which the department has a married employee.*

```
Is Certain (MStat1, 'single', MStat2, 'married')
        In Select e1.Marital Status as MStat1,
                  e2.Marital Status as MStat2
          From Emp e1,
               Emp e2
         Where e1.Dept = e2.Dept
```

The translation rule for the *Is Certain* statement from C-SQL to SQL is given below:

```
Is Certain(<Column_1>,          →  Select * From <C-Table>
           <Column_1_Value>,        Where(
                                   (<Column_1> = <Column_1_Value>
                                    Or <Column_1> < 0)
                                   And
         <Column_2>,            →  (<Column_2> = <Column_2_Value>
         <Column_2_Value>,          Or <Column_2> < 0)
                                   And
         ⋮                      →  ⋮
         <Column_n>,            →  (<Column_n> = <Column_n_Value>
         <Column_n_Value>           Or <Column_n> < 0)
In <C-Table>                    →  )
```

Please note that converted SQL statement is not enough for certainty checking, more process is required to check if the tuples are certain with the global condition, necessary processes will be explained in chapter 5.

Chapter 5

The Algorithms

This chapter describes the algorithms used in the PossDB system implementation. The algorithms related to the PossDB features and are applied after the C-SQL queries are transformed to SQL queries and executed on the PostgreSQL DBMS. The algorithms implemented for extending DBMS functionalities to an uncertainty management system functionality. In the recent DBMSs all the data are certain and each operation is implemented for the certain data. Since in the PossDB system we deal with the uncertain data, we need to extend each functionality. As an example the *Select* operation in the DBMS should be extended to deal with uncertain data.

If there is no algorithm explained in this chapter it means that the converted C-SQL statement is sufficient to fulfill promised functionality.

5.1 Preliminaries

5.1.1 Global Condition

The global condition in the PossDB system, as it explained in the chapter 3, has the structure of CNF formula, each variable only belongs to one conjunct of the CNF formula. Since the *Global Condition* has this structure, each variable can be considered as a set, each member of the set is considered as the possible valuation of that variable.

Let us consider our running example, the *global condition* of the running example is: $\Phi(T) = (x_1 = \text{'M'} \vee x_1 = \text{'F'}) \wedge (x_2 = \text{'M'} \vee x_2 = \text{'F'}) \wedge (x_3 = \text{'IT'} \vee x_3 = \text{'PR'})$. The set representation of the global condition is given below.

$x_1 = \{M, F\}$

$x_2 = \{M, F\}$

$x_3 = \{IT, PR\}$

In order to get the possible valuations of a variable from a global condition, the ***Global***(<*variable name*>) function has been used. This function returns all possible values of the given variable. If the function returns the empty set, it means that the given variable does not have any valuation in the global condition.

5.1.2 Boolean Expression Tree

In the PossDB system, the Boolean formulas are kept as a Boolean expression trees. The Boolean expression tree consists of nodes, where all the nodes are connected to at least one another node. A node that has no children is called leaf node. On top of the tree there exist a node called root node. The root node contains one of the operators given below.

- Equality Operator ($=$)
- Unequality Operator ($\neq$)
- And Operator ($\wedge$)
- Or Operator ($\vee$)

A node can contain an operator, a leaf can contain an operand. In the PossDB system, there exist two types of operands, one of them is a variable and the other one is a constant. Let us consider the example below:

Example 5.1

$$(x = 1 \vee y = 3 \vee z = 5) \wedge (w \neq 2)$$

The Boolean expression tree for the given example is given below.

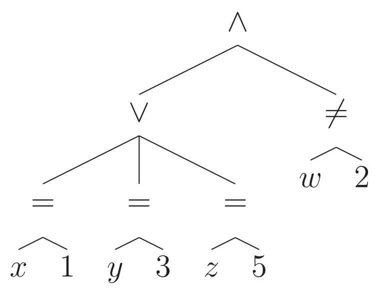

Figure 5.1: A Boolean Expression Tree

In the expression tree a leaf node can only contain an operand. The Boolean Expression Tree helps to convert Boolean formula to different forms. Also the Boolean Expression Tree helps us to annotate variables or operators. Annotating the nodes or the leafs provides a fast traversing characteristic in a Boolean formula.

5.1.3 Disjunctive Normal Form (DNF)

Disjunctive Normal Form is a normalization of a Boolean formula, the abbreviation "DNF" is being used for this form. DNF is a normal form which consists of disjunction of clauses, each disjunctive clause contains a conjunctive clauses. Even though the DNF is disjunction of conjuncts, in some cases a Boolean formula might not contain a disjunct, hence a DNF formula can also consist of only conjuncts. A DNF formula can be in one of the form given below:

- p
- $p \wedge q \wedge \ldots$
- $p \vee q \vee \ldots$
- $(p \wedge q) \vee (r \wedge s \wedge \ldots) \vee \ldots$

Note that p, q, r, s are Boolean formulas over equalities between constants and variables, or variables and variables or their negations.

5.1.4 Conjunctive Normal Form (CNF)

Conjunctive Normal Form is, similarly as DNF, a normalization of a Boolean formula, the abbreviation "CNF" is being used for this form. CNF is a normal form which consists of conjunction of clause, each conjunctive clause contains a disjunctive clause. Even though the CNF is conjunction of disjuncts, in some cases a Boolean formula might not contain a conjunct. A CNF formula can exist in one of the form given below:

- p
- $p \wedge q \wedge \ldots$
- $p \vee q \vee \ldots$
- $(p \vee q) \wedge (r \vee s \vee \ldots) \wedge \ldots$

Note that p, q, r, s are Boolean formulas over equalities between constants and variables, or variables and variables or their negations.

5.2 Auxiliary Algorithms

Algorithms explained in this section are the algorithms that used to support main functions in the PossDB system. Main functions in the PossDB system are the functions which extend the DBMS functionalities, such as Selection, Projection.

5.2.1 Check Satisfiability

Check Satisfiability functionality in the PossDB system, checks if the given condition is satisfiable or not. The given condition also should not be contradictory when it is used with the *global condition*. Each condition in the PossDB system needs to be checked against the *global condition*. The satisfiability check depends on other functions and an operator, which are:

1. **Check Global Satisfiablility**: Checks the generated substitutions along with the global condition, and decides if the substitutions are consistent with global condition or not.

2. $\bigotimes$ **Operator**: Conjugates the substitutions of the AND operator members.

3. **Generate Substitution**: Generates possible substitutions from the given Boolean formula.

4. **Is Satisfiable**: Returns if the condition is satifsiable or not, it is the entry point of satisfiabilily check functionality, it calls necessary auxiliary functions.

The main purpose of these four algorithms is to check if the given condition is satisfiable in the system. Since the PossDB system based on c-tables, and the c-tables are formed with Boolean formulas, these algorithms plays an important role.

Algorithm 1 Check Global Satisfiablility

Input: Substitution Set: ϑ

```
1: function G_SAT($\vartheta$)
2:     for all elements $\upsilon$ in $\vartheta$ do
3:         if $\upsilon$ is $x_1/x_2$ then
4:             return $Global(x_1) \cap Global(x_2) \neq \emptyset$
5:         else if $\upsilon$ is $x_1/a$ then
6:             return $a \in Global(x_1)$
7:         else if $\upsilon$ is a set then                    ▷ $\{x_1/x_2, x_3/a, \ldots\}$
8:             for all elements $\zeta$ in $\upsilon$ do
9:                 if G_SAT($\zeta$) is $False$ then
10:                    return $False$
11:            return $True$
12:        else                                                ▷ e.g. $\emptyset$
13:            return $False$
```

The algorithm 1 checks the given variable valuations if they are contradictory with the global condition or not. Note that the *Global*() function in the line 4 and 6 returns the valuation set of the given variable from the *Global Condition*.

Let us assume that the given substitution is x_1/a. The algorithm checks if there exist a valuation which is $x_1 = a$ in the global condition. If there is no x_1 in the global condition, it means that the given valuation is not satisfiable. If there exist x_1 in the global condition but there is no $x_1 = a$, it also means that the given valuation is not satisfiable and the algorithm returns $False$.

Example 5.2 *Assume that the global condition is:*

$\Phi(T) = (x = 1 \lor x = 2 \lor x = 3) \land (y = 6 \lor y = 7)$

And the input substitution set is: $\{x/3, y/7\}$.

Note that the substitution set $\{x/3, y/7\}$ comes from the Boolean formula $(x = 3 \land y = 7)$, which will be explained in the algorithm 3. In this example, the input is a set of substitutions, hence each substitution in the set should have a valuation in the global condition. The algorithm checks the substitution $x/3$ if there exist $v(x) = 3$ in the global condition, since $x = 3$ is in the global condition, $x/3$ substitution is not contradictory. The same process apply for the substitution $y/7$. Since both substitutions are satisfiable in the global condition, the function returns $True$, which means that the substitution set is not contradictory with the global condition.

The algorithm 2 is used to conjugate valuations in case when a Boolean expression tree contains an **AND** operator. Since **AND** operator is an operator which may yields contradiction, each valuation needs to be checked with the other valuations. As an example let us assume that the given Boolean formula is $x = 1 \land y = 1 \land x = 3$. When we split the formula into three, each subformula is satisfiable, but when we check the each sub formula with other subformulas, we

Algorithm 2 $\bigotimes$ Operator

Input: Left Operand (Substitution): ξ
Input: Right Operand (Substitution): ψ

```
function $\bigotimes(\xi, \psi)$
    if $\xi$ and $\psi$ have the same variable then                $\triangleright\ \xi \rightarrow x = a, \psi \rightarrow x = b$
        return $\xi \cap \psi$
    else if $\xi$ and $\psi$ have different variables then         $\triangleright\ \xi \rightarrow x = a, \psi \rightarrow y = b$
        return $\xi \cup \psi$
    else if $\xi$ and $\psi$ have inverse variables then
        if $|\xi \cap \psi| > 0$ then                                $\triangleright\ \xi \rightarrow x = a, \psi \rightarrow x^- = a$
            return $\emptyset$
        else                                                       $\triangleright\ \xi \rightarrow x = a, \psi \rightarrow x^- = b$
            return $\xi \cup \psi$
```

can see that $x = 1 \wedge x = 3$ yields a contradiction, hence the given formula is not satisfiable. Note that in the algorithm 2 on line 6 the notion **inverse variables** has been used. The inverse variable refers to a variable which is used in a formula with the unqeuality and the equality operator. As an example let us assume the formula $x \neq 2$, the substitution of the given formula will be $x^-/2$, where x^- is considered as inverse variable of x.

Example 5.3 *Consider three different cases used in the algorithm 2.*

1. $x/3 \bigotimes x/4$

   Note that it comes from the Boolean formula $x = 3 \wedge x = 4$, which will be explained in the algorithm 3.

   In this case, the algorithm returns an empty set, because this formula is not satisfiable, a variable can only be assigned to one constant value.

2. $x/3 \bigotimes y/5$

Note that it comes from the Boolean formula $x = 3 \wedge y = 5$. In this case, the algorithm returns substitutions $x/3$ and $y/5$, because they do not contradict each other.

3. $x/3 \bigotimes x^{-}/3$

   Note that it comes from the Boolean formula $x = 3 \wedge x \neq 3$. In this case, the algorithm returns an empty set, because this formula is not satisfiable. On the other hand if the Boolean formula was $x = 3 \wedge x \neq 4$, the operator would return substitutions $x/3$ and $x^{-}/4$, because they do not contradict each other.

Algorithm 3 Generate Substitution

Input: Condition: $\varphi(t)$

```
 1: function SUBS($\varphi(t)$)
 2:     if $\varphi(t)$ is $x = a$ then                ▷ Variable and Constant Equality
 3:         return $x/a$
 4:     else if $\varphi(t)$ is $x = y$ then           ▷ Variable and Variable Equality
 5:         return $\{x/y, y/x\}$
 6:     else if $\varphi(t)$ is $x \neq a$ then        ▷ Variable not equal to Constant
 7:         return $x^{-}/a$
 8:     else if $\varphi(t)$ is $x \neq y$ then        ▷ Variable not equal to Variable
 9:         return $\{x^{-}/y, y^{-}/x\}$
10:     else if $\psi \vee \xi$ then                   ▷ Disjunct of atoms
11:         return {SUBS($\psi$),SUBS($\xi$)}
12:     else if $\psi \wedge \xi$ then                 ▷ Conjunct of atoms
13:         return SUBS($\psi$) $\bigotimes$SUBS($\xi$)
```

The algorithm 3 generates the substitution of the given formula. The generated substitution can be a single substitution or a set which contains all the substitutions of the given formula. Note that returned substitution in a set is different from the substitution which is not in a set. If the formula is returned in a set it means that the substitution is coming from an **OR** operator and if one of the substitution is satisfiable there is no need to check other substitutions in the set. On the other hand if the returned substitution is not a set, it means that it is coming from an **AND** operator and each substitution needs to be checked for the satisfiability.

Example 5.4 *Consider six different cases used in the algorithm 3.*

1. $x = 2$

   Returns the substitution $x/2$.

2. $x = y$

   In this case both operands are variables, hence the function returns the substitutions $x/y, y/x$.

3. $x \neq 4$

   Returns the substitution $x^-/4$.

4. $x \neq y$

   In this case both operands are variables, hence the function returns the substitutions $x^-/y, y^-/x$.

5. $x = 3 \vee x = 5$

   In this case the function returns substitutions in a set. The return set is $\{x/3, x/5\}$.

6. $x = 3 \land y = 5$

   In this case the function splits the Boolean function and generates the substitutions separately. Since the **AND** operator might yield contradiction, more work needed while generating the substitution, for this reason algorithm 2 is needed to generate this substitution. The return value of the function with the given Boolean formula is: $x/3, y/5$.

Algorithm 4 Is Satisfiable

Input: Local Condition: $\varphi(t)$

Ensure: $\varphi(t)$ is in Disjunctive Normal Form

```
1: function SAT($\varphi(t)$)
2:     $\varphi(t) = Calculate\_Transitivity(\varphi(t))$
3:     if $\varphi(t)$ is in $(a = b \land c = d) \lor (e = f \land g = h \land \ldots) \lor \ldots$ form then
4:         for all each disjunct $\rho$ in  do
5:             if G_SAT(SAT(SUBS($\rho$))) is $True$ then
6:                 return $True$
7:         return $False$
8:     else
9:         return G_SAT(SUBS($\varphi(t)$))        ▷ DNF formula $\varphi(t)$ is in other form
```

Note that in the algorithm 4, *Calculate_Transitivity* function reveals the hidden relations between the variables and the constants. As an example let say the given formula is $x = 1 \land x = y$, after the *Calculate_Transitivity* function applied to the formula it will be changed to $x = 1 \land y = 1 \land x = y$. Satisfiability function works only with functions which are in form DNF, formulas given in other forms fails the algorithm.

Example 5.5 *Assume that the global condition is:*

$$\Phi(T) = (x = 1 \lor x = 2 \lor x = 3) \land (y = 4 \lor y = 5) \land (z = 6 \lor z = 7)$$

And we want to check if the condition given below is satisfiable or not.

$(x = 1 \wedge y = 6) \vee (x = 5 \wedge y = 5 \wedge z = 6) \vee (x = y \wedge x = 1) \vee (x = 1 \wedge y = 4 \wedge z = 7)$

In this example since the given condition is DNF, converting to DNF is not needed. As a first step we need to call the function **SAT**. The **SAT** function first calculates the transitivity and the given condition formula turns into a formula given below:

$$(x = 1 \wedge y = 6) \vee (x = 5 \wedge y = 5 \wedge z = 6) \vee (x = 1 \wedge y = 1 \wedge x = y) \vee (x = 1 \wedge y = 4 \wedge z = 7)$$

After that, **SAT** function splits the DNF into four subformulas and iterates over each subformula.

First subformula is $x = 1 \wedge y = 6$ and the substitutions should be generated over this formula. The substitution set generated by the **SUB** function is $(x/1, y/6)$, the next step is checking the substitutions if they are contradictory with the global condition or not. Since there is no $y = 6$ in the global condition, these substitutions are not satisfiable and we need to continue to checking other subformulas.

Second subformula is $x = 5 \wedge y = 5 \wedge z = 6$ and the substitutions are $(x/5, y/5, z/6)$. When we check the substitutions by using the **G_SAT** function, the function returns $False$ because in the global condition $x = 5$ does not exist.

Third subformula is $x = 1 \wedge y = 1 \wedge x = y$ and the substitution of the formula is $(x/1, y/1, x/y, y/x)$. Since there is no $y = 1$ in the global condition, this subformula is also not satisfiable.

Last subformula is $x = 1 \vee y = 4 \vee z = 7$ and the substitution of the formula is $(x/1, y/4, z/7)$. Since all of them are in the global condition, this subformula is satisfiable and it returns $True$, which means that the given condition in this example is a satisfiable condition.

5.2.2 Is Tautology

Is Tautology is a Boolean function, which returns if the given condition is $True$ for all valuations. Satisfiability is the preprocess of Tautology control. A Boolean formula can be a tautology in two ways in the PossDB system. One of them is the Boolean formula itself can be a tautology, such as $x = 3 \vee x \neq 3$, and the other one is when $\Phi(T) \rightarrow \varphi(t)$ produces a tautology. The algorithms used for the tautology check are given below. Note that, the some of the functions called in these algorithms were explained in the previous sections.

Algorithm 5 Check Global Tautology

Input: Valuation Set: ϑ

```
function G_TAUT($\vartheta$)
    for all variable $x_n$ in $\vartheta$ do
        Create a set $G'_{x_n} = \emptyset$
    for all elements $\upsilon$ in $\vartheta$ do
        if $\upsilon$ is $x_1/a$ then
            $G'_{x_1} = G'_{x_1} \cap \{a\}$
        else if $\upsilon$ is $x_1^-/a$ and $a \in Global(x_1)$ then
            return $False$
    for all Created set $G'_x$ do
        if $G'_x \neq Global(x)$ then
            return $False$
    return $True$
```

The algorithm 5 checks if the given formula is already in the global condition.

Instead of using the whole global condition, this algorithm uses the valuations and checks one by one in the global condition. This one by one checking speeds up the process because it does not require to go over on the whole global condition. Consider the example below:

Example 5.6 $\Phi(T) = (x = 1 \vee x = 2 \vee x = 3) \wedge (y = 6 \vee y = 8 \vee y \neq 10) \wedge (z = 4)$
$\varphi(t) = (z = 4)$

In the given example above, in theory we need to apply $\Phi(T) \rightarrow \varphi(t)$, which is $\neg\Phi(T) \vee \varphi(t) = ((x \neq 1 \wedge x \neq 2 \wedge x \neq 3) \vee (y \neq 6 \wedge y \neq 8 \wedge y = 10) \vee (z \neq 4))$ $\vee (z = 4)$. As it can be seen $(z \neq 4) \vee (z = 4)$ yields to tautology but we needed to use the whole global condition. The algorithm 5 checks tautology by using the substitution in the local condition and checks just the local condition substitutions one by one. Since the global condition is stored in a hashed structure, the algorithm gains speed while checking the tautology.

The algorithm 6 solves the tautology which is caused by the condition itself. In this case we do not need to check the tautology with the global condition. As it shown in the algorithm 6 on line 8, a Boolean formula can be tautology in three cases, they are :

1. $x = a \vee x \neq a$

   In this case, Boolean formula is tautology regardless the possible valuations of the variable x.

2. $x = x$

   In this case, Boolean formula is tautology because the variable x is equal to itself in any valuation of the variable x.

Algorithm 6 Is Tautology

```
Input: Local Condition: $\varphi(t)$
Ensure: $\varphi(t)$ is in conjunctive normal form
Ensure: $\varphi(t)$ is Satisfiable

function TAUT($\varphi(t)$)
    $\varphi(t) = Calculate\_Transitivity(\varphi(t))$        ▷ Calculate Transitivity
    if root of $\varphi(t)$ is AND operator in expression tree then
        for all conjunct $\rho$ in $\varphi(t)$ do
            if TAUT($\rho$) is $False$ then
                return $False$
    else if root of $\varphi(t)$ is OR operator in expression tree then
        if $\varphi(t)$ contains both “$x = a$ and $x \neq a$” or $a = a$ or $x = x$ then
            return $True$
        else
            return G_TAUT(SUB($\varphi(t)$))
    else
        return G_TAUT(SUB($\varphi(t)$))
    return $True$
```

3. $a = a$

   In this case, Boolean formula is tautology because there is no variable and the constant value is equal to itself.

Example 5.7 *Assume that the local condition is $x = 1 \vee y = 5 \vee x \neq 1$. This local condition produces tautology because in the formula $x = 1 \vee x \neq 1$ is true in any valuation of the variable x and y.*

If the given Boolean formula does not yield a tautology, it needs to be checked with the global condition. In order to check the given Boolean formula with the global condition, the substitution of the given formula needs to be generated and with the generated substitutions the **G_TAUT** function needs to be called.

5.3 Select

In c-tables a tuple can contain variables or Boolean formulas. Since variables and the Boolean formula need to be checked with the valuations, the *Select* operation of the DBMS needs to be extended.

In the PossDB system, after the C-SQL statement converted to SQL statement and executed on the PostgreSQL, the output table is processed by a Java application. The algorithm explained below takes the tuples one by one from the output c-table and processes the tuples. The input c-table is the result of the converted C-SQL query result.

Algorithm 7 Select

Input: Select Condition: θ
Input: Conditional Table: T

```
1:  function SELECT($T, \theta$)
2:      for all tuples $t$ in $T$ do
3:          $\theta_t = \theta$
4:          for all attribute names in $\theta_t$ do
5:              Replace attribute name with attribute value from tuple $t$
6:          if SAT(DNF($\theta_t \wedge \phi(t)$)) then
7:              if TAUT(CNF($\theta_t \wedge \phi(t)$)) then
8:                  $\varphi(t) = True.$
9:              else
10:                 $\varphi(t) = \theta_t \wedge \varphi(t)$
11:         else
12:             Remove $t$ from $T$
13:     return $T$.
```

Note that, the $CNF()$ and $DNF()$ functions used in the algorithm above to convert the Boolean formula into form of DNF or CNF. The algorithm 7 extends the Select algorithm by adding variable checking and Boolean formula satisfiability control.

Example 5.8 `Select * From Emp Where Dept = 'IT'`

The result for the C-SQL query from the DBMS before applying the algorithms is given below:

Name	Gender	Marital Status	Dept	$\varphi(t)$
Alice	x_1	married	IT	$True$
David	M	married	x_3	$True$
Ella	F	single	x_3	$True$

Note that, before applying the select algorithm, the local conditions of the tuples were $True$. When the selected table above sent to the *Select* function, each tuple is going to be evaluated with the select condition (Dept = 'IT'). Since none of the tuples have a local condition, the conjunction of the where condition and the local condition is going to be the same. If the table had different local conditions each tuple would have a different local condition which is the conjunction of the local condition and the given where condition in the select statement. In the second and the third tuple *Dept* attribute contains a variable, hence the *Dept* will be replaced with that variable, after changing the attribute name, the local condition will be converted to x_3 ='IT' and it will be placed in the local condition of the processed tuple. After applying the select algorithm, the table above will be changed to the table given below:

Name	Gender	Marital Status	Dept	$\varphi(t)$
Alice	x_1	married	IT	$True$
David	M	married	x_3	x_3 ='IT'
Ella	F	single	x_3	x_3 ='IT'

5.4 Project

The projection operation is implemented, as expected, as an extension of the *Select* operation in the PossDB system. The algorithm explained in the *Select* section applies to the *Project* operation as well.

Example 5.9 `Select Name,Gender From Emp Where Gender = 'F'`

The result for the C-SQL query from the DBMS before applying the algorithms is given below:

Name	Gender	$\varphi(t)$
Alice	x_1	$True$
Bob	x_2	$True$
Ella	F	$True$

For the first tuple when we apply the algorithm, the local condition will be generated as $x_1 =$ 'F', as the same way, the second tuple will generate the local condition $x_2 =$ 'F'. Since $x_1 =$ 'F' and $x_2 =$ 'F' are both satisfiable but not tautology, the generated local conditions will be displayed in the final result. The final result is given below:

Name	Gender	$\varphi(t)$
Alice	x_1	$x_1 =$ 'F'
Bob	x_2	$x_2 =$ 'F'
Ella	F	$True$

Note that the *Project* operation only discards the unneeded columns, besides that, it works as the same way as the *Select* operation.

5.5 Join

The Join operation built based on the *Select* operation as the standard SQL. The keyword `INNER JOIN` distinguishes the *Select* operation from the *Join* operation. Since the join operation implemented on top of the *Select* operation, *Join* operation keeps all the features of the *Select* operation.

Algorithm 8 Join

```
Input: Select Condition: $\theta$
Input: Join Condition: $\omega$
Input: Joined Table : $T$

 1: function JOIN($T, \theta, \omega$)
 2:     for all tuples $t$ in $T$ do
 3:         $\theta_t = \theta \wedge \omega \wedge \Theta(t)$
 4:         for all attribute names in $\theta_t$ do
 5:             Replace attribute name with attribute value from $t$
 6:         if SAT(DNF($\theta_t$)) then
 7:             if TAUT(CNF($\theta_t$)) then
 8:                 $\varphi(t) = True$
 9:             else
10:                 $\varphi(t) = \theta_t$
11:         else
12:             Remove the tuple $t$ from $T$
13:     return $T$
```

The join algorithm works almost the same way as the select algorithm, the main difference is on the line 3, each local condition consist of with three items, which are:

1. θ: Represents the Select Condition, which is given after the `WHERE` keyword in C-SQL.

2. ω: Represents the Join Condition, which is given after the `INNER JOIN` ... `ON` keywords in C-SQL.

3. $\Theta(t)$: Represents the local condition which is appeared after executing the converted C-SQL statement. $\Theta(t)$ comes from the $Merge\_Condition()$ function, which is explained in the section 4.4. $Merge\_Condition()$ function returns the conjunction of the condition columns from the tables which are participated to join.

Example 5.10 *Assume that there is a relation Dept which stores the phone numbers and the locations of the departments. Consider joining the Dept relation and the Emp relation of our running example to show the phone numbers and the locations of the employees. The Dept relation is given below:*

Dept

Name	Location	Phone	Condition
AC	101	2090	$True$
HR	103	1010	$True$
PR	201	2450	$True$
IT	301	4270	$True$

To join the *Emp* and *Dept* relations, we need to run the C-SQL query given below:

```
Select E.Name Employee,
       D.Phone Phone,
       D.Location Location
  From Emp E
 Inner Join Dept D ON
       E.Dept = D.Name
```

After the C-SQL query converted to SQL and run on the DBMS, it returns the given table below:

E.Name	E.Dept	D.Name	D.Location	D.Phone	Condition
Alice	IT	IT	301	4270	$True$
Bob	HR	HR	103	1010	$True$
David	x_3	AC	101	2090	$True$
David	x_3	HR	103	1010	$True$
David	x_3	PR	201	2450	$True$
David	x_3	IT	301	4270	$True$
Ella	x_4	AC	101	2090	$True$
Ella	x_4	HR	103	1010	$True$
Ella	x_4	PR	201	2450	$True$
Ella	x_4	IT	301	4270	$True$

Note that $E.Gender$ and $E.Marital\ Status$ is also in the result set, but since they are not needed, we have not showed those columns in the result. After getting the table above from the DBMS, we need to apply the Join algorithm.

In this example we only have the join condition which is ($E.Dept = D.Name$). In this condition each attribute name will be replaced with the attribute value. For the first tuple $E.Dept$ will be replaced with IT and $D.Name$ will be replaced with IT, then the condition will turn into $IT = IT$, since it is a tautology the tuple will be in the result of the join operation with the local condition $True$. Same procedure applies to the second tuple.

For the third tuple, when we replace the condition with attribute values of the tuple, we get $x_3 = AC$, since it is not satisfiable because of the global condition, it will be eliminated from the result. Same procedure applies to the fourth tuple.

For the fifth tuple, when we replace the condition with attribute values of the tuple, we get $x_3 = PR$, since it is satisfiable but not tautology, it will be in the result with the local condition $x_3 = PR$.

After all the tuples has been processed, the column projection applies and projects selected columns and returns the final result. Final result of the example is given below:

E.Name	D.Location	D.Phone	Condition
Alice	301	4270	$True$
Bob	103	1010	$True$
David	201	2450	$x_3 = PR$
David	301	4270	$x_3 = IT$
Ella	201	2450	$x_4 = PR$
Ella	301	4270	$x_4 = IT$

5.6 Insert

Insert operation is almost the same operation as in the standard SQL. The only difference is that if there exist a local condition given, that local condition should be satisfiable otherwise, the insertion should be aborted by the PossDB system. After the local condition passes the satisfiability check, it also checked against the tautology and if it is tautology the local condition will be converted to $True$ automatically, otherwise the DNF form of the local condition will be inserted.

Algorithm 9 Insert

Input: Tuple: t

Input: Local Condition: $\varphi(t)$

```
1: function INSERT($t, \varphi(t)$)
2:     if SAT($\varphi(t)$) then
3:         if TAUT($\varphi(t)$) then
4:             $\varphi(t) = True$
5:         else
6:             $\varphi(t)$ =DNF($\varphi(t)$)
7:     else
8:         return
9:     Insert $t$ with the local condition $\varphi(t)$
```

5.7 Is Possible

Is Possible function is a unique function which has no counterpart in standard SQL. The Is Possible function returns *True* if the given tuple exist in some valuation of the nulls and it returns *False* if the given tuple does not exist in any of the valuations of the nulls. The C-SQL query returns the answer where all the nulls treated as satisfiable conditions, after that the *Is Possible* function checks the variables in the tuple if there exist a valuation with constant values.

Algorithm 10 Is Possible

Input: Conditional Table: T

Input: Tuple: t

```
1: function IS_POSS($T, t$)
2:     $\theta = True$
3:     for all tuple element $e_n$ in $t$ do
4:         $\theta = \theta \wedge$ (Name of $e_n$ = Value of $e_n$)
5:     if |SELECT(T,$\theta$)| > 0 then          ▷ If there exist a tuple in the result
6:         return $True$
7:     return $False$
```

In order to check the valuation, the given tuple is converted to a Boolean formula and the c-table is selected with the generated Boolean formula. The line 3 generates the Boolean formula. When we call the *Select* function, which is explained in the algorithm 7, with the generated Boolean formula, it returns us a c-table. Returned c-table shows us if the tuple is possible in the c-table or not. Note that the Select operation checks all the variable to constant mappings in the PossDB system. Select operation returns a non empty c-table in two cases:

1. The given tuple exists in the c-table without containing any varaibles. Note that in this case, the given tuple is also a certain tuple in the c-table.
2. There exist a tuple in the c-table which contains variables and those variables can have the constant valuation that has been queried.

Let us consider the example below:

Example 5.11 `Is Possible(Name,Bob,Gender,M,Dept,HR) In Emp`

The tuple generates the Boolean formula given below.

$$True \wedge (Name = \text{'Bob'}) \wedge (Gender = \text{'M'}) \wedge (Dept = \text{'HR'})$$

the equivalent normalized version of the given formula is:

$$(Name = \text{'Bob'}) \wedge (Gender = \text{'M'}) \wedge (Dept = \text{'HR'})$$

Now we need to Select *Emp* relation with the given condition above.

$$\textbf{Select}\boldsymbol{(}Emp\textbf{,}(Name = \text{'Bob'}) \wedge (Gender = \text{'M'}) \wedge (Dept = \text{'HR'})\boldsymbol{)}$$

The *Select* function above returns us a c-table. The result of the *Select* function is given below:

Name	Gender	Marital Status	Condition
Bob	x_2	married	$x_2 =$ 'M'

Since the c-table is not empty, it means that the given tuple is possible in the given *Emp* relation.

5.8 Is Certain

Is Certain function is also an unique function as the Is Possible function. The Is Certain function needs more steps than the Is Possible function needs because each possible world needs to be checked. In practice in c-tables, whenever there exist a local condition with $True$, it means the the tuple is certain. Unfortunately sometimes the local condition does not seem to be $True$ to the user or the tuple can occur in a different forms in the c-table which is difficult to catch by the user that they are not mutually exclusive. The Is Certain function solves the above mentioned unseen tautologies.

Let us consider the example below.

Example 5.12 *Is there exist a certain tuple which has Comedian Jerry in the given c-table below:*

Name	Job	$\varphi(t)$
Jerry	Comedian	$x \neq$ 'Comedian'
Jerry	x	$x =$ 'Comedian'

Algorithm 11 Is Certain

Ensure: Given tuple is possible in the c-table
Input: Conditional Table: T
Input: Tuple: t

```
1: function IS_CERT($T, t$)
2:     $\theta = True$
3:     for all tuple element $e$ in $t$ do
4:         $\theta = \theta \wedge$ (Name of $e$ = Value of $e$)
5:     $T' =$ SELECT(T,$\theta$)
6:     $\theta' = False$
7:     for all tuple $t'$ in $T'$ do
8:         $\theta' = \theta' \vee \varphi(t')$
9:     return TAUT($\theta'$)
```

In the c-table above, there exist two people who are names as Jerry. Both are depend on a local condition, which states that they are possible tuples in the c-table but not certain. On the other hand, the other representation of the given tuple c-table is given below:

Name	Job	$\varphi(t)$
Jerry	Comedian	~~$x =$ 'Comedian' $\vee x \neq$ 'Comedian'~~ $True$

The local condition of the only tuple of the above c-table is a tautology, which states that it is certain, the tuple exists in any valuations of the null values.

Chapter 6

Experimental Results

This chapter describes the experimental results of the PossDB system. The main idea behind the experimental results is to demonstrate the scalability of the system. Our experiments benchmark the system with the most similar system MayBMS, which is implemented for the same purpose as the PossDB system. MayBMS also returns the exact answer to queries as PossDB does, and the scalability of MayBMS has been proven [9]. Furthermore, both PossDB and MayBMS are built on top of PostgeSQL. Note that MayBMS system has two accessible versions, we conduct our experiments with the first available version of MayBMS [9] system, which does not have the probabilistic features.

6.1 Data Set

The experiments are based on the queries and data which were used for the MayBMS experimental evaluation [9]. Their experiment used a large census database encoded as integers [28]. The large census database contains the real

United States of America (USA) census data from 1990. The database contains the 5 percent of the real census data, which amounts to 10 million tuples. The large census database has only one relation which is named as R and the attributes of the relation R are given in the table 6.1 with their descriptions. Since the census data is complete data, 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 every experiment which is compared with the MayBMS, the same noised data set have been used. The MayBMS system and the noise generator has been obtained from [1].

Attribute Name	**Description**
YEAR	Census year
DATANUM	Data set number
SERIAL	Household serial number
HHWT	Household weight
STATEFIP	State (FIPS code)
GQ	Group quarters status
PERNUM	Person number in sample unit
PERWT	Person weight
STEPPOP	Probable step/adopted father
SUBFAM	Subfamily membership
CBSFTYPE	Subfamily type (original Census Bureau classification)
SEX	Sex
MARST	Marital status
CITIZEN	Citizenship status
SPEAKENG	Speaks English
OCC1990	Occupation, 1990 basis
MIGPUMA	PUMA of residence 5 years ago
VETSTAT (general)	Veteran status [general version]
VETSTATD (detailed)	Veteran status [detailed version]
VET75X80	Veteran, served 1975 to 1980
VET55X64	Veteran, served 1955 to 1964
VETOTHER	Veteran of other period

Table 6.1: Census Database Attributes

6.2 System Settings

The experiments were conducted on Intel®Core^{TM}i5-760 processor machine with 8 GB RAM, running Windows 7 Enterprise, PostgreSQL 9.0, and Java SE Runtime Environment build 1.6.027.

6.3 MayBMS & PossDB Benchmark

This section provides the benchmark results of the PossDB system with the MayBMS system. Both systems were tested with the same data set, in the same PostgreSQL system with an empty database cache. Since there is no special query language in MayBMS system, both systems are tested with the SQL queries, the query conversion time does not included. Three queries were chosen to show the behaviour of different operation characteristics. Each query has a different character from the others, as a result, we expect to see different execution times.

Experiment 1 *Selection over the relation R. The given query below used to test both systems.*

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

The results for the given query is given below.

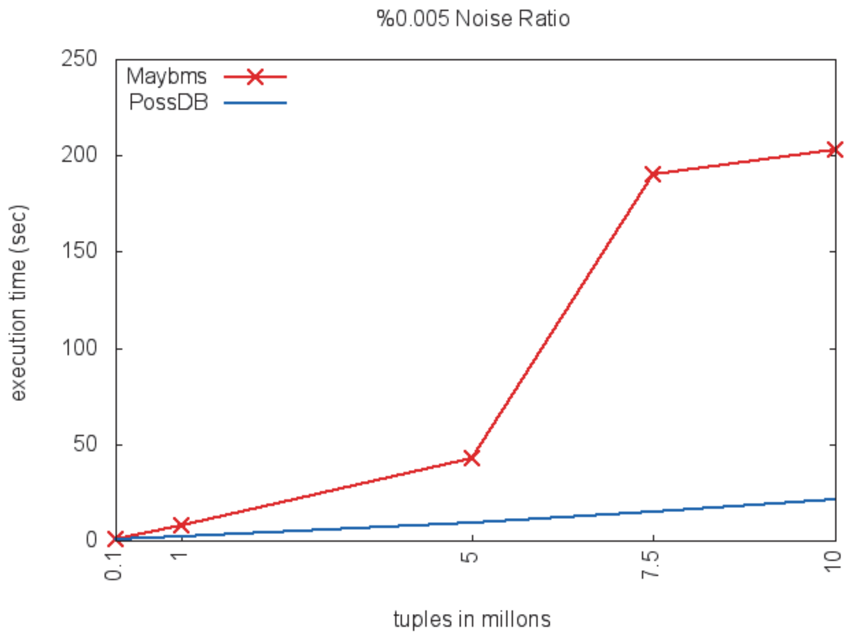

Figure 6.1: Selection with %0.005 Noise

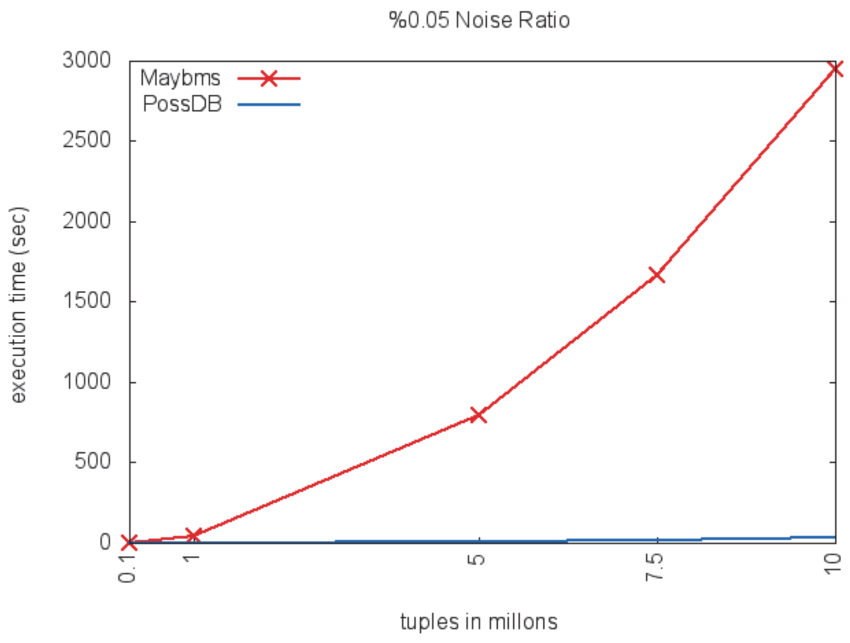

Figure 6.2: Selection with %0.05 Noise

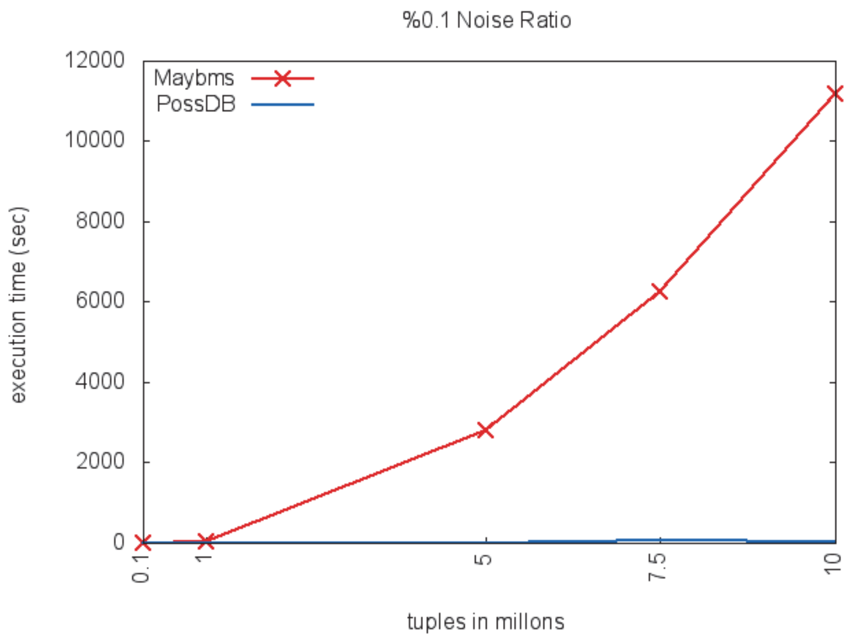

Figure 6.3: Selection with %0.1 Noise

In the experiment 1, we wanted to check our systems scalability with an easy query. The query returns the USA citizens who is military veteran. As expected in a simple query, the PossDB system processes the data much faster than the MayBMS, it is because of the MayBMS system needs to perform the join operation even though the query is just a simple query. In the PossDB system the only process during the query execution time is tuple by tuple variable valuation check if the tuple contains variable in the *VETSTAT* or *CITIZEN* attribute.

Experiment 2 *Selection and Projection experiment, where there are few columns projected. The given query below has been used to test both systems.*

```
SELECT STATEFIP, OCC1990 FROM R
 WHERE SPEAKENG = 3
```

The results for the given query is given below.

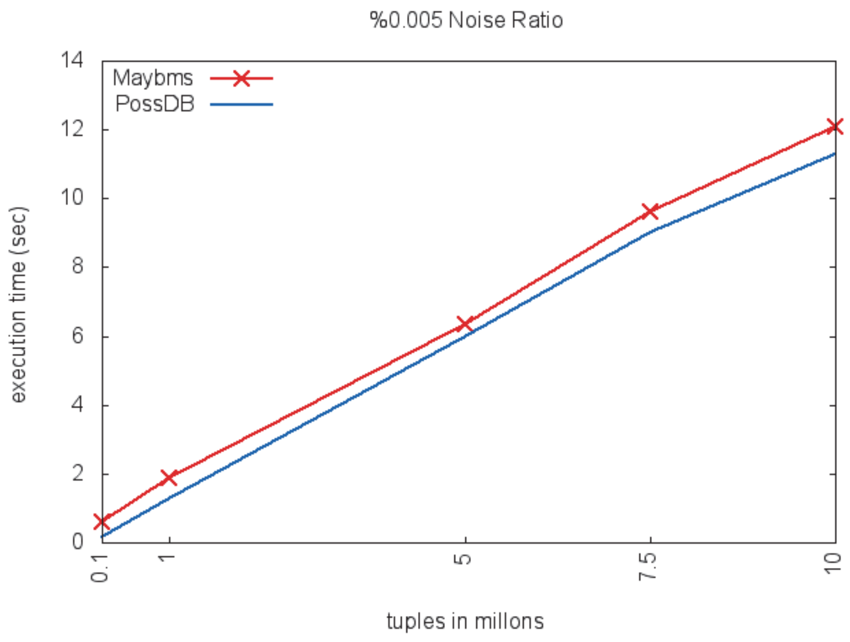

Figure 6.4: Projection with two columns and %0.005 Noise

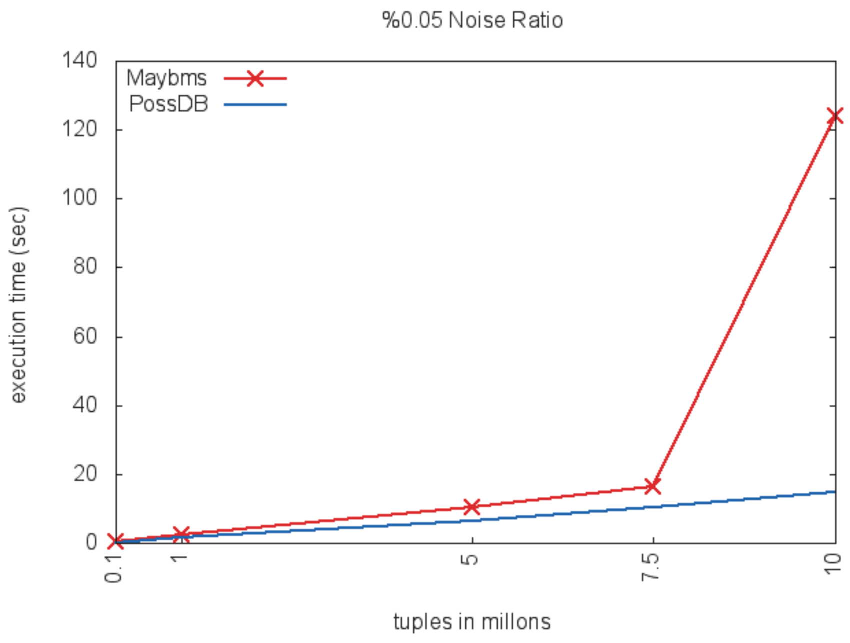

Figure 6.5: Projection with two columns and %0.05 Noise

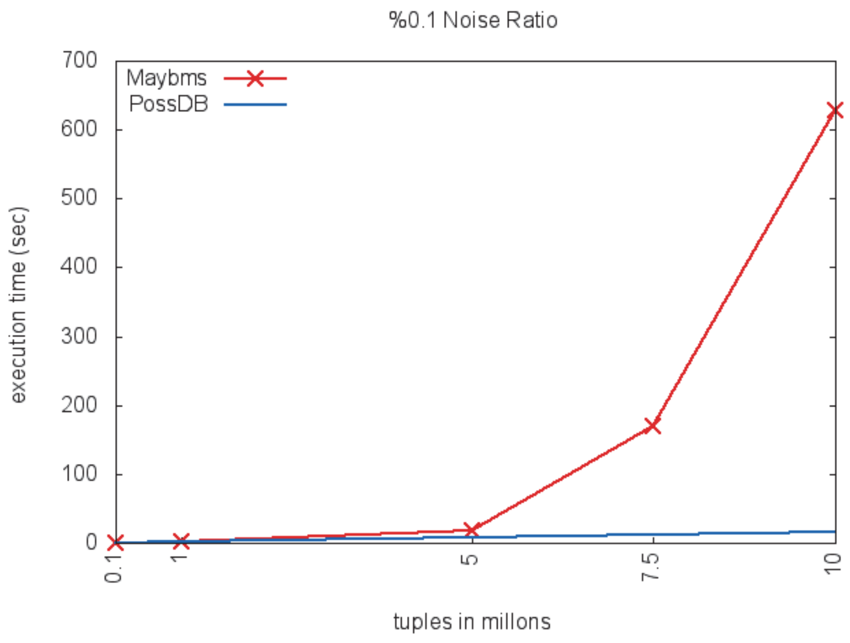

Figure 6.6: Projection with two columns and %0.1 Noise

In the experiment 2, we wanted to benchmark our system with MayBMS with the most similar case. In the MayBMS theory, the less column you select the less decomposed world you get. This brings us almost the same amount of data in both systems. The query returns the state of residency and occupations of all the USA citizens who can only speak English.

As it can bee seen from the graphs, when there exist only a few variables in the database, the both systems works with almost with the same speed. Unfortunately when the number of variables increases, the decomposed world sets gets bigger and the MayBMS starts to loose its fast execution speed.

Experiment 3 *Selection and Projection experiment, where there is a column - column equality in the condition. The given query below has been used to test both systems.*

```
SELECT STATEFIP, OCC1990, CITIZEN, SUBFAM FROM R
 WHERE (SUBFAM > 4)
   AND (CITIZEN = 1)
   AND (STATEFIP = OCC1990)
```

The results for the given query is given below.

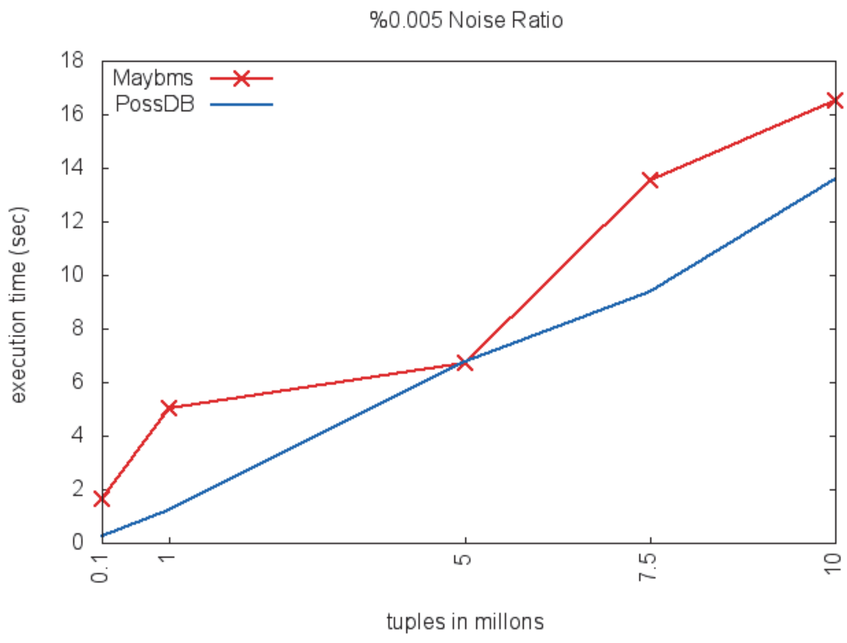

Figure 6.7: Projection with column-column equality and %0.005 Noise

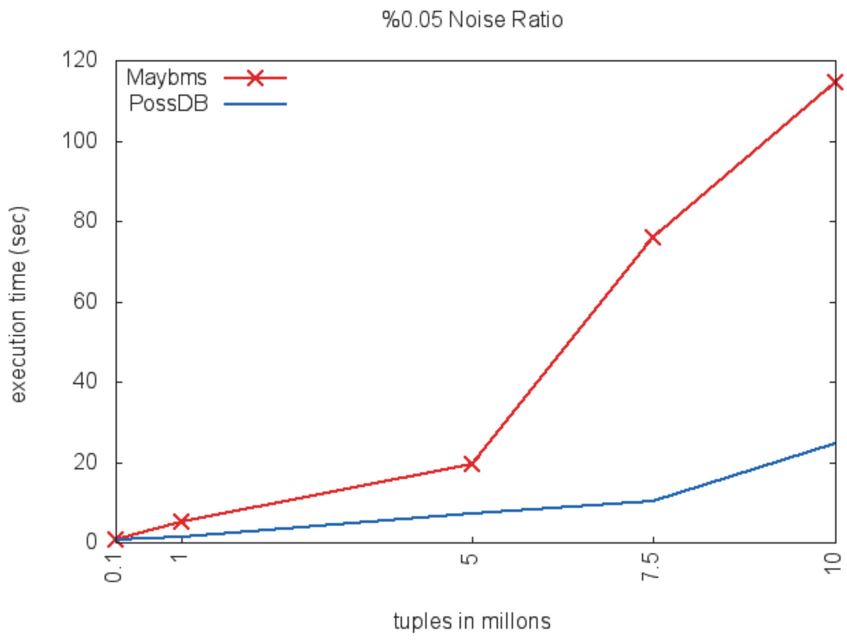

Figure 6.8: Projection with column-column equality and %0.05 Noise

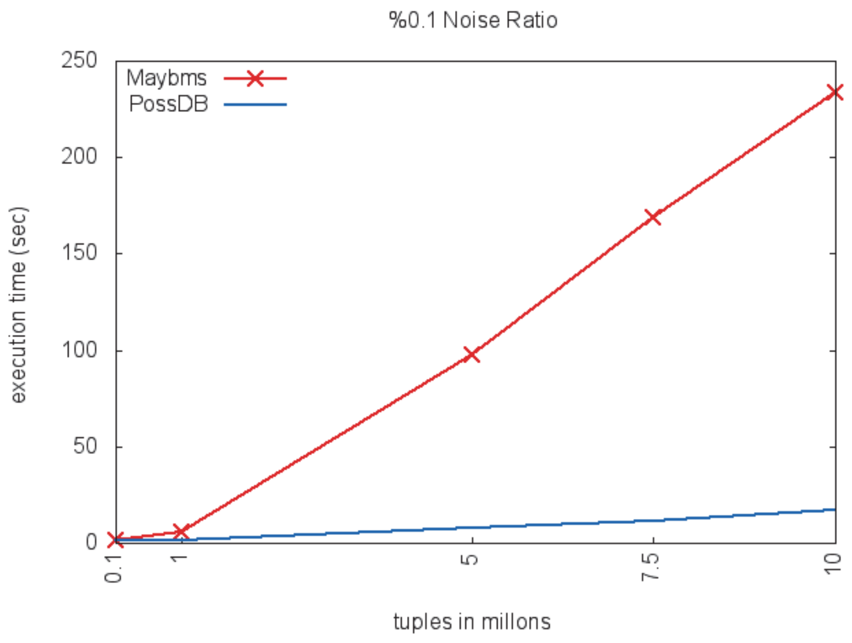

Figure 6.9: Projection with column-column equality and %0.1 Noise

In the experiment 3, the main goal is to check the PossDB system with a more complex query. We achieved the complex query by adding two columns equality condition. The column to column equality brings us a variable to variable equality in some cases and it is more complex valuation check rather than variable to constant equality. This query returns the USA citizens who was born outside of the USA, lives with more then 4 people in the house, and works in the field which is specific for their state of residence.

As we expected the variable to variable equality takes more time to check variable to constant equation, because in the variable to variable equality cases, the intersection needs to be done to both variable valuation set, and finally the intersect set should be considered to find the answer.

6.4 Summary

In this chapter, we tested our system with MayBMS-1 and analyzed the results. The results show that PossDB clearly outperforms MayBMS. This is mostly because of the MayBMS needs to use two more extra relations to express possible valuations for the nulls, but in the PossDB, there is no extra relation needed to store possible valuations. In the PossDB system each valuation for the variables stored in the memory with the hashed structure. This helps the PossDB system for checking the valuations in less time than MayBMS. The MayBMS system needs to perform joins in order to check valuations against the constant values.

As it can be seen from the graphics, the PossDB system works more efficiently than the MayBMS system with large number of data. This can be explained as the same way as the first experiment, since MayBMS needs to join tables, it needs more space and it needs to use secondary storage [16, 21], but the PossDB needs to go over tuple by tuple, and it does not require to do join.

Chapter 7

Conclusion and Future Work

The PossDB system presented in this thesis is capable of storing incomplete data using c-tables. Even if the c-tables data model is well known, it has not been implemented before, although many probabilistic systems essentially use probabilistic versions of c-tables.

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

Possibility and Certainty checking functionalities 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 [3].

The future work should be 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. [10] or [11]. The SAT-solver would then handle the satisfiability and tautology tests, which is likely to further improve the performance of the system.

Finally, the system can be extended by implementing the chase based procedure on conditional tables [18] 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 system and the noise generator. http://pdbench.sourceforge.net/. 77

[2] Zql: a java sql parser, 2002. 35

[3] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. Theor. Comput. Sci., 78(1):158–187, 1991. 88

[4] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In 32nd International Conference on Very Large Data Bases. VLDB 2006 (demonstration description), September 2006. 12, 18

[5] E M Airoldi. Getting started in probabilistic graphical models. PLoS Comput Biol, 3(12), December 2007. 12

[6] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In Proceedings of the

2008 IEEE 24th International Conference on Data Engineering, ICDE '08, pages 983–992, Washington, DC, USA, 2008. IEEE Computer Society. 18

[7] Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In ICDE, pages 1479–1480, 2007. 14

[8] Lyublena Antova, Christoph Koch, and Dan Olteanu. Query language support for incomplete information in the maybms system. In Proceedings of the 33rd international conference on Very large data bases, VLDB '07, pages 1422–1425. VLDB Endowment, 2007. 20

[9] Lyublena Antova, Christoph Koch, and Dan Olteanu. 10ˆ10ˆ6 worlds and beyond: efficient representation and processing of incomplete information. The VLDB Journal, 18(5):1021–1040, October 2009. 76

[10] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In IJCAI, pages 399–404, 2009. 88

[11] Armin Biere. Preprocessing and inprocessing techniques in sat. In Haifa Verification Conference, page 1, 2011. 88

[12] Jihad Boulos, Nilesh Dalvi, Bhushan Mandhani, Shobhit Mathur, Chris Re, and Dan Suciu. Mystiq: a system for finding more answers by using probabilities. In Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, pages 891–893, New York, NY, USA, 2005. ACM. 11

[13] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. U-dbms: a database

system for managing constantly-evolving data. In In VLDB 05: Proceedings of the 31st international conference on Very large data bases, pages 1271–1274, 2005. 11

[14] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04, pages 864–875. VLDB Endowment, 2004. 11

[15] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06, pages 73–84, New York, NY, USA, 2006. ACM. 20

[16] Goetz Graefe. Query evaluation techniques for large databases. ACM COMPUTING SURVEYS, 25:73–170, 1993. 86

[17] Gösta Grahne. Conditional tables. In LING LIU and M.TAMER ÖZSU, editors, Encyclopedia of Database Systems, pages 446–447. Springer US, 2009. 22

[18] Gösta Grahne and Adrian Onet. Closed world chasing. In LID, pages 7–14, 2011. 88

[19] Todd J. Green, Grigoris Karvounarakis, and Val 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. 4

[20] Levent Gürgen, Claudia Roncancio, Cyril Labbé, and Vincent Olive. Transactional issues in sensor data management. In Proceedings of the 3rd workshop on Data management for sensor networks: in conjunction with VLDB 2006, DMSN '06, pages 27–32, New York, NY, USA, 2006. ACM. 20

[21] Evan P. Harris and Kotagiri Ramamohanarao. Join algorithm costs revisited. The VLDB Journal, 5(1):064–084, January 1996. 86

[22] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: a probabilistic database management system. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, SIGMOD Conference, pages 1071–1074. ACM, 2009. 12, 20

[23] T. Imielinski and K. Vadaparty. Complexity of query processing in databases with or-objects. In Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, PODS '89, pages 51–65, New York, NY, USA, 1989. ACM. 16

[24] Tomasz Imielinski and Witold Lipski. Incomplete information in relational databases. J.ACM, 31(4):761–791, September 1984. 3, 22

[25] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 687–700, New York, NY, USA, 2008. ACM. 20

[26] Dan Olteanu, Christoph Koch, and Lyublena Antova. World-set decom-

positions: Expressiveness and efficient algorithms. Theor. Comput. Sci., 403(2-3):265–284, August 2008. 12

[27] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator, 1995. 35

[28] 2004 Ruggles, S. Integrated public use microdata series: Version 3.0. 76

[29] Anish Das Sarma, Omar Benjelloun, Alon Halevy, and Jennifer Widom. Working models for uncertain data. In Proceedings of the 22nd International Conference on Data Engineering, ICDE '06, pages 7–, Washington, DC, USA, 2006. IEEE Computer Society. 17

[30] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. The VLDB Journal, 18(5):1065–1090, October 2009. 12

[31] Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. Proc. VLDB Endow., 1(1):340–351, August 2008. 12

[32] Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In Managing and Mining Uncertain Data. Springer, 2008. 18