Query Answering over Functional Dependency Repairs

by

Artur Galiullin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

© Artur Galiullin 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Inconsistency often arises in real-world databases and, as a result, critical queries over dirty data may lead users to make ill-informed decisions. Functional dependencies (FDs) can be used to specify intended semantics of the underlying data and aid with the cleaning task. Enumerating and evaluating all the possible repairs to FD violations is infeasible, while approaches that produce a single repair or attempt to isolate the dirty portion of data are often too destructive or constraining. In this thesis, we leverage a recent advance in data cleaning that allows sampling from a well defined space of reasonable repairs, and provide the user with a data management tool that gives uncertain query answers over this space. We propose a framework to compute probabilistic query answers as though each repair sample were a possible world. We show experimentally that queries over many possible repairs produce results that are more useful than other approaches and that our system can scale to large datasets.

Acknowledgements

I would like to extend my gratitude and appreciation to my supervisor, Dr. Ihab Ilyas, for his support and guidance throughout my whole graduate program. I would also like to thank Dr. Lukasz Golab, for his great help in the process of writing my thesis and for agreeing to be one of my readers. Finally, I would like to thank my second reader, Dr. Tamer Özsu, for his participation and feedback.

Special thanks to my colleague, Dr. George Beskales, for all his help.

Big thanks to all of my close friends, for all their support and motivation.

I would like to thank my parents and family for their neverending support during all these years. This work would never have been possible without them.

Dedication

This is dedicated to Capt. Jean-Luc Picard.

Table of Contents

List of Figures

Chapter 1

Introduction

Inconsistencies in data often arise in practice and may cause serious and costly problems. Faced with a dirty database, users still need to get query answers over their data. Manually repairing large and evolving databases is infeasible, while leaving data unchanged may result in incorrect answers to critical queries.

Functional dependencies (FDs) are integrity constraints that could be used to specify indented semantics underlying the data. In scenarios such as data integration or information extraction from the web, the FDs may no longer hold, and could be used to identify the resulting inconsistencies in data. There may be many other sources of error, as illustrated by a the following simple example.

Example 1 *Consider a customer database depicted in Figure 1.1. An FD that makes sense for this schema is* $Area \rightarrow City$*, signifying that an area code should uniquely determine a city. The current state of the database is dirty, since tuples* t_1, t_2, t_3 *all contain the same area code, but differ in city. One reason for this inconsistency may be that Patrick and Jane have previously lived in Manhattan and both recently moved to Queens. They have updated their telephone number with this business, but failed to update their address, a situation depicted as the ground truth.*

Suppose the business from Example 1 needs to assess the market potential in Queens by performing an analytic query over their database that calculates total sales in that city, and a join is required with the customer table to connect each customer with their city. If the query is performed without modifying the dirty data, it may incorrectly suggest that there are not enough customers in Queens, costing the business potential profit.

Dirty Database

	Name	City	Area
t_1	Patrick	Manhattan	347
t_2	Jane	Manhattan	347
t_3	Clare	Queens	347
t_4	Allen	Queens	718
t_5	Betty	Manhattan	212

Ground Truth

	Name	City	Area
t_1	Patrick	Queens	347
t_2	Jane	Queens	347
t_3	Clare	Queens	347
t_4	Allen	Queens	718
t_5	Betty	Manhattan	212

Repair 1

	Name	City	Area
t_1	Patrick	Manhattan	347
t_2	Jane	Manhattan	347
t_3	Clare	Queens	718
t_4	Allen	Queens	718
t_5	Betty	Manhattan	212

Repair 2

	Name	City	Area
t_1	Patrick	Manhattan	212
t_2	Jane	Queens	347
t_3	Clare	Queens	347
t_4	Allen	Queens	718
t_5	Betty	Manhattan	212

Figure 1.1: A database instance violating the FD $Area \rightarrow City$

Figure 1.1 shows two possible ways to repair the dirty database. Both ways rely on modifying the values so that the resulting data satisfies the FD. In general, the number of possible repairs of FD violations is intractable, which is caused by interaction of violations that span multiple FDs.

One way to tackle this challenge is to delete violating tuples. In Example 1, we would need to delete either tuples t_1, t_2 or t_3. Previous work have focused on finding the fewest number of tuples that need to be removed to reach a consistent state [11, 12]. In either case, we would not get closer to the ground truth. Worse, this will result in deletion of clean data that could still be very useful, i.e., the customer names and their telephone numbers.

Alternatively, we could rely on value modification and try to reach a consistent state by making the fewest number of changes or by using some information distance metric [9, 13, 19]. For example, in Figure 1.1, we may choose `Repair 1` over `Repair 2` because it made fewer changes. However, it should be evident that choosing a single repair over other possibilities creates a potential for making very bad mistakes, even if a strong minimality metric is used.

Finally, we could consider a number of possible repairs that satisfy some minimality constraint and return only the answers that appear in all of them, an approach called *Consistent Query Answering (CQA)* [3, 12, 16, 20, 23]. In our example, the query for customers in Queens would return Clare and Allen, since they appear in both repairs. This is no better than evaluating

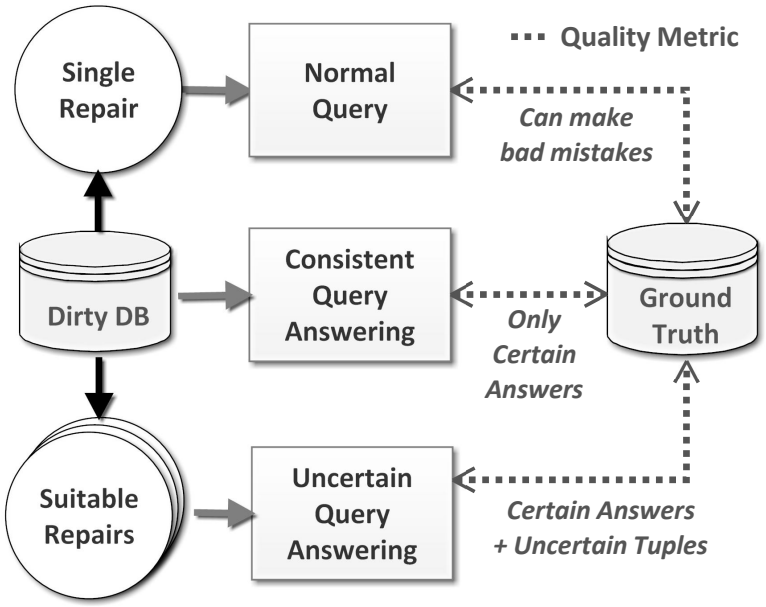


Figure 1.2: Data cleaning driven query evaluation

the query over the dirty database, and worse, if we consider additional repairs, e.g. a repair that changes the value of $City$ in t_3, CQA would return even less information. In general, CQA is often too constraining, especially in the case of FD violations where there is a very large number of ways to repair the data.

On the other hand, suppose we generalize the notion of CQA and consider answers that do not necessarily occur in all minimal repairs. If enough repairs are considered, some of them could potentially be very close to the ground truth giving the right answers an opportunity to be part of the result, instead of being completely discarded. In our example, considering both repairs 1 and 2 in Figure 1.1 will provide a more informative answer to the query for customers in Queens and suggest the Jane could *possibly* be from Queens. With additional repairs, Patrick may also be correctly considered. Thus, the business will be able to make a more informed decision using such results.

The approach that allows answers that may not occur is some instances of an uncertain database is called *uncertain query answering* and is an established way to manage data uncertainty. Figure 1.2 provides a summary for various ways to answer queries in the presence of FD violations. If we attempt to find a single good repair and execute queries over the resulting database, some answers are prone to be far from ground truth. If queries over the dirty database return only *certain* answers, the results will be limited and a lot of answers could be missed. On

the other hand, if a set of possible repairs is generated, uncertain query answering will provide many other possibilities in addition to the certain answers, giving it a better chance to be closer to the ground truth.

Our aim in this thesis is to investigate a relaxation of CQA using state of the art in probabilistic databases (PDBs). A number of previous works have proposed semantics for doing so in the context of the deduplication data cleaning task [1, 8]. The example we presented above motivates the application of this concept to FDs, which are able to encode business rules and assist in cleaning data that has become dirty for many reasons other than duplication.

It is often the case that the user does not know how to repair the dirty DB, and there are no legacy experts that can help, i.e., the user has little or no information about the ground truth. There are a number of ways of obtaining approximate answers in this situation.

One way is to set a probability distribution on attributes. Then a generative model with simple correlations can be used [18]. This only works in relatively simple scenarios, such as a measurement error, where correlations are reasonable enough to be expressed as conditional probabilities. Repairs to FD violations create complex correlations that cannot be efficiently modeled at attribute level.

Another way is to treat each possible repair as a possible world and use an existing PDB that is expressive enough to represent this space [2]. However, PDBs that can faithfully represent the correlations that arise in FD repairs are inherently inefficient in computing even the simplest probabilistic queries.

In both situations, we still face the problem of enumerating an intractable space of all the possible repairs for probabilistic representation. In the context of deduplication, this problem has been easier to solve because of the ability to cluster data into independent tuple blocks.

A number of previous works on FD cleaning have attempted to constrain this space by imposing minimality constrains, such as avoiding redundant repairs. However, even these subsets of all possible repairs have proven to be difficult to enumerate.

A recent work has proposed a way to sample from a space of possible repairs [7]. In this thesis, we leverage this advance in data cleaning, and provide the user with a data management tool that gives uncertain query answers over a set of possible FD repairs. We propose a framework to compute probabilistic query answers as though each repair sample were a possible world. Thus, the probability of a query answer is proportional to the number of repairs it appears in.

The way in which possible repairs are obtained and their probability distribution are independent of our proposal. We show empirically that query answers over a set of possible repairs can get closer to ground truth than other approaches.

The system we propose deals with the challenge of representing the complex correlations that occur within repairs of FD violations, while allowing efficient probabilistic query answering directly over repair samples.

The conceptual contribution of this thesis is as follows.

- We propose a framework for uncertain query answering over dirty data that violates FD constraints. This framework integrates and extends previous proposals in the areas of probabilistic databases and data cleaning, by leveraging recent work in repair sampling.

  Uncertain query answering over FD repairs has not been possible prior to the advent of repair sampling. Generative systems require a distribution model to be defined over the uncertain data, while probabilistic databases require materialization of the possible worlds and are inefficient in computing probabilistic queries in the presence of complex correlations.

The technical contributions are as follows.

- We propose a space efficient representation system for storing FD repair samples inside a conventional RDBMS. Our storage model avoids explicit storage of each repair and is specifically designed to be accessed by fast sample-based query evaluation.

  Further, we propose an optimization to a naive implementation of this idea that exploits the tuple-level commonalities that arise in FD repairs. This improvement allows for space efficient caching and we empirically show several orders of magnitude faster computation for certain queries.

- We show how to efficiently implement sample-based query semantics [18] over materialized samples stored in our representation system.

  This approach is more efficient than generating samples on-the-fly for each query, for several reasons. First, current state of the art generative systems do not efficiently support the complex correlations that arise in FD repairs. Second, precomputing and materializing samples enables traditional query optimization techniques such as indexing.

- We implement our proposed methods and build a prototype system.

- Using experimental results, we argue that probabilistic answers over a set of repairs provide intuitive semantics and can be computed efficiently. We show that uncertain query answering provides better quality results than consistent query answering and state of the art

single repair methods. Finally, we show that our approach scales well for large databases and a large number of repair samples.

Chapter 2

Problem Statement

In this section, we describe the general problem that our system attempts to solve.

Given a dirty database, we assume there is a black-box system that can produce a large number of possibly distinct repairs to this database. We further assume that the whole space of possible repairs produced by this system is either intractable or there is no known mechanism to efficiently exhaust the possibilities. Thus, we are interested in a sample of repairs of arbitrary size.

Since this sample is likely to contain repairs the differ from one another, this collection of repairs represents a probability distribution on the possible instantiations of the database. At this point, we should note that our proposed system is independent of the actual space of possible repairs that the block-box system samples from. Instead, we focus on handling the general characteristics that any set of possible repairs to FD violations would have.

The goal of our proposal is then to produce probabilistic answers to queries over this collection of possible repairs. This can be achieved with *possible worlds* semantics, which is traditionally used in probabilistic query answering. Each repair sample represents a possible world and the probability of the query answer is proportional to the number of repairs it appeared in.

Consider again the example in Figure 1.1. We were given a dirty database and a black-box system produced two repairs. Each repair represents a possible world, i.e. a possible instantiation of values to tuples of the original dirty database. The two repairs together represent a probability distribution of possible instantiations. For example, the *cell* $t_3[City]$ is instantiated to the value $Queens$ in both repairs, so a query asking for the city where the name is $Clare$ should return $Queens$ along with its probability of 1.0. A more interesting example would be a query that asks for the city for $Jane$. The value for $t_2[City]$ differs in the two repairs, so the query should output two results, $Manhattan$ with probability 0.5 and $Queens$ with probability 0.5.

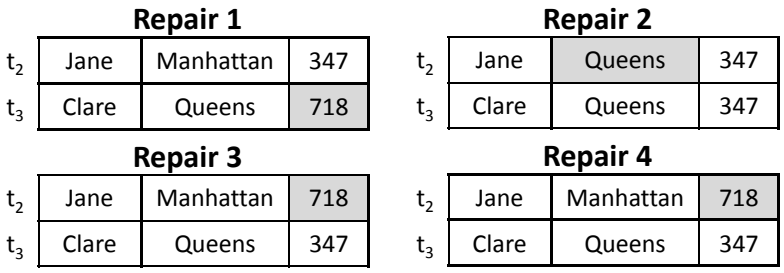


Figure 2.1: A set of possible repairs to t_2 and t_3

These general semantics of probabilistic query evaluations are further described in Section 4.3 by defining a brute-force implementation. In this thesis, our proposal is a realization of these semantics that is more efficient than state of the art systems used off-the-shelf, specifically in the context of FD repairs. We next discuss the particular challenges that arise in this context.

2.1 FD Repair Space Characteristics

The example in Figure 1.1 demonstrates a characteristic of the probability space that is fundamental to any set of "reasonable" FD repairs, namely, the existence of *cell-level correlations*, i.e. correlations that span both the tuple and attribute dimension. By reasonable repairs, we assume some minimality constraint on the the value changes - a repair that sets all values in the database to 0 would be considered unreasonable, while a repair that attempts to minimize the number of redundant changes is reasonable. For example, in Figure 1.1, the probability of $t_3[Area] = 212$ depends on the the value chosen for $t_2[City]$ and cannot be calculated independently.

The existence of cell-level correlations should be expected as the general case for FD repairs. Any violation of an FD involves two tuples, in which the values for the left hand side (LHS) attributes are equal while the values for the right hand side (RHS) attributes differ, as is the case for t_2 and t_3 and the FD $Area \rightarrow City$. In order to repair this violation, assuming we are not allowed to delete tuples, we can do one of the following. We can choose one of the two tuples and either change the RHS values such that they differ, or the LHS values such that they become equal. Consider a set of possible repairs shown in Figure 2.1.

Repairs 1 and 3 represent a space where the choice of tuples is free, but the choice of attributes is restricted to LHS, i.e. $Area$ in our example. In this space, the choice of $Area$ for t_3 is dependent on the choice of $Area$ for t_2. Thus, the uncertainty can be modeled on attribute level with tuple correlations.

Repairs 2 and 4 represent a space where the choice of attribute is free, but the choice of tuple is restricted to t_2. In this space, the choice of $Area$ for t_2 is dependent on the choice of its $City$. Thus, the uncertainty can be modeled on tuple level with attribute correlations.

However, if we consider the space of all four repairs, the correlations span both dimensions. One example is the probability of $t_3[Area] = 347$, which is dependent among other things on the value for $t_2[City]$. In this case, the uncertainty must be modeled on cell level, with cell value correlations.

Another characteristic of FD repairs is the presence of arbitrary *certain cells*, i.e. cells that are assigned the same value across all repairs in the sample. In Figure 2.1, cells $t_2[Name]$, $t_3[Name]$, $t_2[City]$ are certain. In general, cells are certain for attributes not part of any FDs and tuples not part of any violations. So while $t_2[City]$ is not expected to be certain given enough repair samples, cells for tuples t_4 and t_5 in Figure 1.1 are not part of any violation and thus do not need to be modified. Thus, we should usually expect a large proportion of certain cells, as the normal assumption in data cleaning is that the majority of the database is clean. Further, this means that an attribute level uncertainty model is not suited to represent a space of FD repairs, as attributes are likely to be only partially uncertain.

Chapter 3

Related Work

As discussed in Section 1, there are a number of ways to deal with FD violations. One way is to produce a single repair with some minimality constraint [9, 13, 19]. Another approach is to provide query answers that are consistent across a set of repairs. Since these approaches essentially provide a single possibility, they cannot be used in conjunction with a probabilistic database to support uncertain query answering over dirty data. A recent proposal allows an arbitrarily large set of possible repairs to be sampled [7], which provides an opportunity to use a PDB system in this context.

Traditional PDBs use a simple representation system where each tuple is an independent probabilistic event [4, 6, 14, 15, 21], which allows a large class of queries to be evaluated efficiently, but does not faithfully model correlations that arise in repairing functional dependency violations. As shown in Section 2, FD repairs give rise to correlations that span both dimensions, tuples and attributes. In the presence of many FDs whose attributes overlap, the dependencies can form a large and complex network of correlations that is difficult to model.

A recently proposed system, called MayBMS [2], decomposes the possible worlds based on the independence between sets of fields and allows complex correlations to be represented. However, the flexibility of its representation system makes computation of most probabilistic queries inefficient.

Another PDB system, called MCDB [18], focuses on incorporating statistical methods directly into the database and provides probabilistic query answering by generating samples on-the-fly. However, while MCDB supports both tuple and attribute correlations, it is designed to model uncertainty at the level of attributes. As shown in Figure 1.1, an FD repair may require only a small portion of attribute values to change. In typical data cleaning scenarios where the

t_2.Name	Pr
Jane	1.0

X

t_3.Name	Pr
Clare	1.0

X

t_3.City	Pr
Queens	1.0

X

t_2.City	t_2.Area	t_3.Area	Pr
Manhattan	347	718	0.25
Queens	347	347	0.25
Manhattan	718	347	0.50

Figure 3.1: Independent components in MayBMS

majority of the database is assumed to be clean, most attribute values remain unchanged across possible repairs. In such cases, MCDB is ill-suited to efficiently compute probabilistic queries.

Previous work includes proposals on probabilistic query answering in the context of other data cleaning tasks, such as deduplication [1, 8]. To the best of our knowledge, no such work exists for FD repairs. In deduplication, the ability to cluster the dirty database into independent blocks of mutually exclusive tuples makes it possible to apply existing PDB technology directly, whereas FD violations give rise to a much more complex uncertainty model which makes direct application challenging.

3.1 State of the Art in PDBs

In this section, we show how the MayBMS and MCDB systems can be used to solve our problem, the inefficiencies that arise in each case individually, and provide intuition on how an integration of these two approaches may result in an efficient solution.

3.1.1 MayBMS

The main focus of MayBMS is space-efficient materialization of an exponential number of possible worlds, based on the independence between arbitrary sets of cells. The reason we chose MayBMS for case study is because it is the state of the art in PDBs with cell level uncertainty model. As will be demonstrated below, the main reason against using MayBMS to solve our problem is the fundamental inefficiency of its query evaluation.

Figure 3.1 shows how MayBMS would split the possible worlds represented by four repairs in Figure 2.1 into independent components. The values for certain cells are stored each in a separate component, with probability 1.0. And since the values for the three uncertain cells are correlated, they get stored in one component. Each row in this component represents a distinct value combination that appeared in the possible worlds, and its probability as a proportion. For example, the value combination in the third row of this component has appeared in Repairs 3 and 4, out of 4 repairs total, and thus is assigned a probability of 0.5.

This set of independent components represents a factorization of all the possible worlds. Composing the components, i.e. taking their cartesian product, produces all the distinct possible worlds along with their probabilities.

In query evaluation, MayBMS focuses on transforming the components, such that the resulting components represent the query answer. The major problem with query evaluation comes when computing the probability values of the results. In order to compute the probability of a tuple in the query answer, all the components that contain the cells from this tuple need to be composed. Thus, returning all tuples in the query answer along with their probabilities requires that we compose all of the components that resulted from running the query.

Suppose we produce N repair samples for each dirty relation in the database. Since FD repairs to different relations are independent, each relation in the best case will consist of one component. Further suppose that each repair is distinct, which will result in N rows for each component. Thus, a join between M such relations could result in a query answer that represents N^M possible worlds, i.e. in general, computing query results is exponential in the number of independent components.

3.1.2 MCDB

MCDB focuses on incorporating statistical methods into the database and on using sampling to efficiently estimate probabilistic query answers. Efficient query processing over database samples is precisely our goal and thus the techniques proposed MCDB are useful for our system. However, since MCDB only supports attribute level uncertainty and generates samples on-the-fly for each query, off-the-shelf, it cannot solve our problem efficiently.

A natural example of MCDB application is illustrated in Figure 3.2. The relation `Patients` holds information on blood pressure of patients and the `systolic blood pressure (SBP)` attribute is uncertain. The SBP attribute is distributed normally, with the mean and standard deviation parameters conditioned on the patient's gender are stored in the `SBP_Params` relation.

For each probabilistic query that involves $Patients$ and SBP, MCDB generates N samples of SBP data for each patient from the normal distribution. This collection then represents

Patients

	Name	Gender	SBP
t_1	Patrick	Male	?
t_2	Jane	Female	?
t_3	Clare	Female	?
t_4	Allen	Male	?
t_5	Betty	Female	?

SBP_Params

Gender	Mean	StD
Male	118.3	5.2
Female	110.5	3.1

Figure 3.2: Uncertain Relation Example in MCDB

N possible worlds and the semantics for computing probabilities of answers are illustrated in Section 4.3.

Such uncertain relation is defined in MCDB using the following schema semantics:

```
CREATE TABLE Patients(Name, Gender, SBP) AS
FOR EACH p in PATIENTS
    WITH SBP AS Normal (
        SELECT s.Mean, s.StD
        FROM SBP_Params s
        WHERE s.Gender = p.Gender)
    SELECT p.Name, p.Gender, b.Value
    FROM SBP b
```

In this case, $Normal$ is a *value generating function (VGF)*, which takes as parameters the mean and standard deviation and returns a table with one row and one column $Value$, which contains a random sample from the normal distribution with the given parameters. When a relation has several uncertain attributes, several VGFs can be used in the definition, values from which are then joined with the certain attributes of the relation, as the SBP is joined with the gender of the patient in the example above. These VGFs can also be user-defined, as long as they follow the interface rules.

When there are correlations between attributes, a VGF can return a table with multiple columns, one for each attribute, which represents a sample from the correlated space. For correlations between tuples, a VGF returns a table with multiple rows, one for each tuple in the correlated group. In this case, the outer $foreach$ loop would have to iterate over groups of rows from the original relation, instead of one row at a time as in the above example.

It may then seem possible to use these mechanisms for representing correlations in the context of FD repairs, which has both tuple and attribute correlations. However, the cell value correlations that arise in from many overlapping FDs can form large and complex networks. Thus, defining a space of FD repairs using such simple generative model would be extremely difficult, especially with a primitive correlation model provided by MCDB.

It is still possible to use MCDB to solve our problem. This is achieved by defining, for each dirty relation, a VGF that returns a random repair sample of the whole relation, i.e. there is one correlated tuple group, which includes all tuples in the relation, and one correlated attribute group, which include all attributes of the relation. This may seem like a naive approach, but it is difficult to do better for our problem in MCDB. The problem comes from the fact that MCDB fundamentally only supports attribute level uncertainty, meaning that if an attribute is uncertain, the assumption is that all of cells under this attribute are uncertain. As discussed in Section 2.1, FD repairs tend to have *partially* uncertain attributes, where the majority of values for cells are the same across all repairs.

In general, the VGF may not need to return the whole table for each sample. The *certain* attributes, i.e. those for which all values are the same across the repairs, do not need to be included, as would be the case with the $Name$ attribute in Figure 1.1. Further efforts, such as splitting the table into independent tuple groups, would not improve efficiency. This is because, if an attribute has at least one uncertain cell, values for all of its cells will need to be generated for each sample during query evaluation, by definition of attribute level uncertainty. Thus, this method of using MCDB would approach the naive query evaluation method described in Section 4.3.

It is still possible for MCDB to beat the naive method, if the repair process itself takes significant time. This can be achieved by materialization. If we pre-sample the repairs and store them in a MayBMS-like schema, it would be possible to define a VGF that samples from this space, using the marginal probabilities in for each value combination. In this case, while the redundancy of regenerating certain cell values remains, the repairs themselves don't have to be performed for each query.

In fact, in the presence of complex cell value correlations and partially uncertain attributes, it is difficult to imagine efficiently using a purely generative query evaluation approach of MCDB, instead of materialization. Our approach uses the sample-based query evaluation semantics defined in MCDB, directly over materialized repair samples, removing the need to inefficiently generate a second layer of samples for each query. Materialization brings a secondary benefit, enabling traditional query optimization techniques such as indexing. Ideas from MayBMS provide a way for space-efficient materialization, stemming from its cell level uncertainty model.

3.1.3 Global Samples

In this section, we show how the query evaluation semantics of MCDB avoid the exponential complexity of MayBMS, with the concept of *global samples*. This idea was not discussed in the original proposal of MCDB [18], however the understanding of this concept is crucial for making the distinction between query evaluation semantics of MCDB and MayBMS.

We illustrate the concept of global samples using an example. Figure 3.3 shows results of a coin flip experiment simulation, where 10 coins were flipped, 10 times each, with independent $P(Heads) = 0.9$.

A MayBMS-like approach would be to store the results in 10 independent components, one for each coin, and compute the marginal probabilities for possible outcomes, as shown in the $Components$ relation. On the other hand, the $GlobalSamples$ relation stores the set of sample IDs (SIDs) for each possibility, for each coin. Note that the SIDs are *global* to all the coins. For example, on the tenth coin flip ($SID = 9$), coin C_5 produced Heads, and coin C_6 produced Tails, so 9 is added to the Heads set for C_5 and to the Tails set of C_6. So even though the coin flips were independent, the global SIDs enforces dependence.

The advantage of enforcing dependence lies in its bounding of the number of possible worlds, and thus making probabilistic query evaluation computationally efficient. Consider a general case, with M independent components, N samples, and K possible values for each component. The number of possible worlds under the independence assumption would then be K^M, while in the global sample approach, it would be $M \cdot N$. In our experiments with FD repairs, we show empirically that K often grows linearly with N.

Under the global sample semantics, the probability values for some query answers may differ from the ones obtained under independence assumption. For example, computing $P(AllHeads)$ under global sample semantics would involve intersecting Heads sets of all ten coins. The resulting set is $\{6\}$ - also column 6 in the coin flips data of Figure 3.3 - leading to probability $1/10 = 0.1$. On the other hand, using the components, we multiply out the $P(Heads)$ for all coins, leading to probability 0.108. Note that, in general, accuracy of the result using the correct independence assumption is expected to be only marginally better than of results using global sample semantics. Thus, it is always possible to simply take a linear number of extra samples to achieve the same standard error as with the independence assumption. Since the difference in the size of the possible worlds of the two methods is exponential, this trade-off makes sense. In Section 10.5, we show empirically that the difference in quality results between the two approaches, given a fixed number of samples, is negligible.

Observe that the presence of samples, in form of repairs, is what allows us to use this method in our problem. In the scenarios where only marginal probabilities are given - what MayBMS is

		Sample IDs									
		0	1	2	3	4	5	6	7	8	9
Coins	C_0	H	H	H	H	H	H	H	H	H	H
	C_1	H	H	H	T	H	H	H	T	H	H
	C_2	T	H	T	H	H	H	H	H	H	H
	C_3	H	T	H	H	H	H	H	H	H	H
	C_4	T	H	H	H	T	H	H	H	T	H
	C_5	H	H	H	H	H	H	H	H	H	H
	C_6	H	H	T	H	H	H	H	T	T	T
	C_7	H	H	H	H	T	T	H	H	H	H
	C_8	T	H	H	H	H	H	H	T	H	H
	C_9	H	T	H	H	H	T	H	T	H	H

Components

	H	T
C_0	1.0	0.0
C_1	0.8	0.2
C_2	0.8	0.2
C_3	0.9	0.1
C_4	0.7	0.3
C_5	1.0	0.0
C_6	0.6	0.4
C_7	0.8	0.2
C_8	0.8	0.2
C_9	0.7	0.3

Global Samples

	Sample IDs
C_0	H: (0,1,2,3,4,5,6,7,8,9) T:()
C_1	H: (0,1,2,4,5,6,8,9) T:(3,7)
C_2	H: (1,3,4,5,6,7,8,9) T:(0,2)
C_3	H: (0,2,3,4,5,6,7,8,9) T:(1)
C_4	H: (1,2,3,5,6,7,9) T:(0,4,8)
C_5	H: (0,1,2,3,4,5,6,7,8,9) T:()
C_6	H: (0,1,3,4,5,6) T:(2,7,8,9)
C_7	H: (0,1,2,3,6,7,8,9) T:(4,5)
C_8	H: (1,2,3,4,5,6,8,9) T:(0,7)
C_9	H: (0,2,3,4,6,8,9) T:(1,5,7)

Figure 3.3: Coin flip experiment

designed for - the use of global samples is not possible, since the needed information is absent.

In other words, FD repair samples provide a unique scenario. The complex cell value correlations and partially uncertain attributes can be materialized with space-efficiency of MayBMS,

but using its query semantics is inefficient. The same characteristics of the FD repair space prevent us from being able to use MCDB directly, because of its attribute level uncertainty, but the presence of repair samples enables the use of its efficient query evaluation semantics. The focus of our thesis is then on efficiently materializing the repair samples without the loss of information required for the global sample semantics and on the implementation of these semantics over the materialized repairs for efficient query processing.

Chapter 4

System Architecture

In this section we overview our proposed probabilistic repair database system (PRepDB). Figure 4.1 shows the system architecture. Conceptually, the user runs a query over the schema of the dirty database and gets probabilistic results based on the set of repairs of this dirty database.

4.1 Repair Sampling

The majority of work on repairing FD violations has focused on producing a single repair that satisfies some minimality constraint. This approach cannot be used for uncertain query answering, which requires a set of possible worlds.

In [7], Beskales et. al. propose a space of *cardinality-set-minimal* (CSM) repairs, which

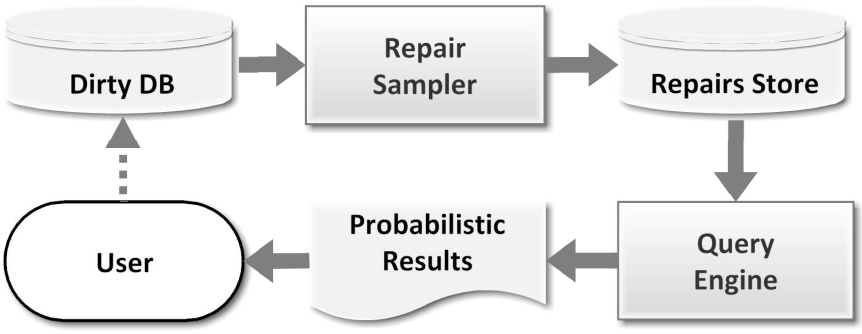


Figure 4.1: PRepDB Architecure

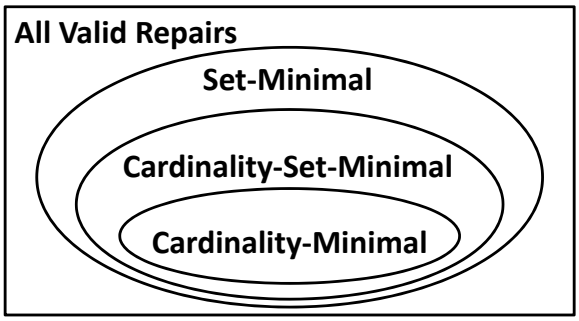


Figure 4.2: Relationship Between Repair Spaces

aims to strike a balance between the fewest changes metric of cardinality-minimality and the necessary changes criterion of set-minimality [7]. A repair is defined to be CSM if no subset of changed cells can be reverted back to their original values without violating the set of FDs, even if the values of the other changed cells are allowed to change.

In contrast, in a *set-minimal* repair, no subset of changed cells can be reverted back to original values, while values for other changed cells are unchanged. All CSM repairs are also set-minimal. Finally, a repair is *cardinality-minimal* if the number of changed cells is minimal with respect to all valid repairs. All cardinality-minimal repairs are also CSM and set-minimal. Figure 4.2 shows the relationship between the repair spaces.

For example in Figure 1.1, `Repair 2` has cells $t_1[Area]$ and $t_2[City]$ modified. If we revert $t_1[Area]$ from 212 to the original value 347, it will cause a violation. In this case, there is no value we can set for $t_2[City]$ that will resolve the violation now present between t_1 and t_3. Same could be said for $t_2[City]$, as well as the case where values for both cells are reverted. Thus, this repair is cardinality-set-minimal.

`Repair 1` in Figure 1.1 is cardinality-minimal, as it has one changed cell and at least one change is needed to repair the database. Finally, if we disregard t_1, a repair where the value of $City$ for t_2 and t_3 are changed to $Bronx$ will be set-minimal. For this repair, reverting either cell or them both to their original value will cause a violation. However, when the value of $t_2[City]$ is reverted to $Manhattan$, changing $t_3[City]$ to $Manhattan$ will resolve the violation. Thus, this repair is not CSM.

Critically, the proposal in [7] provides a way to sample from the CSM repair space, which in turn enables the application of uncertain query answering. PRepDB uses the CSM sampling algorithm to generate a set of repairs for each dirty relation. As shown in Figure 4.1, the *Repair Sampler* takes as input the dirty database, which includes the FDs defined over its schema, and outputs a set of CSM repairs.

4.2 Representation System

The ultimate goal of the system is to give the user an ability to query the repaired database. PRepDB does not require generating repair samples for each query, this would be too inefficient and often unnecessary. Instead, the repairs are sampled as a preprocessing step. This removes the need to sample repairs on-the-fly and also allows traditional query optimization techniques such as indexing to be used.

Once a set of repair samples of desired size is collected, the data can be encoded and stored inside a traditional RDBMS. As shown in Figure 4.1, the repairs from the *Repair Sampler* get stored in the *Repairs Store*.

We propose two representation systems that allow storing large sets of repair samples. First, a baseline system is proposed in Section 5, which stores possible values at the level of individual cells. In Section 6 we propose an improved representation system, which exploits commonalities that occur at tuple level. The improved system enables caching and greatly speeds up the relational join computation.

4.3 Query Evaluation Semantics

PRepDB uses the sample-based query evaluation semantics proposed in [18], which in turn follows the conventional *possible worlds* semantics. That is, we treat each repair sample as a possible world and the probability of a query answer is proportional to the number of worlds it appears in.

In this section, we describe the naive algorithm for query evaluation over a sample of repairs. The purpose is to illustrate the semantics of probabilistic queries, provide intuition for the design of our proposed system and provide a baseline system to compare against.

Algorithm 1 defines the naive method to execute a query Q over a dirty relation D using N repair samples. For each unique sample ID SID, the algorithm generates a random repair sample of D and materializes it into a temporary table T. Then, query Q is executed using the clean relation T. If Q involves more than one relation, a repair has to be produced for each one, so that Q can run over this set of clean relations. The results of this query are inserted into the answer relation A and each tuple is appended with the current sample ID SID. Note that relation A has the schema representing the result of Q, with an additional SID column.

After N samples, the answer tuples in A are sorted and a linear scan is performed to identify unique tuples. For each unique tuple, its probability is calculated as the proportion of samples it has appeared in out of N and the result is written to output.

Algorithm 1 Naive Query Algorithm Processing

1: **procedure** NAIVEQUERY(D, Q, N)
2: Load dirty table D into memory
3: **for** $SID = 1$ to N **do**
4: Generate a repair sample R_{SID} for D
5: Insert R_{SID} into a temp table T_{SID}
6: Execute the query Q over table T_{SID}
7: Insert results into answer table A and append with sample ID SID
8: Sort A, and for every unique tuple, count the number of sample IDs and divide by N to get its probability
9: Output each unique tuple in A along with its probability

This algorithm implements the *possible worlds* semantics over repair samples in a literal way and has a number of drawbacks, which our proposed system addresses. First, the repairs have to be generated on-the-fly for each query. Although repairs can be sampled efficiently, a disproportionate amount of latency is incurred over query evaluation, where the majority of time spent on regenerating repairs to the same relations. This may be unacceptable in data warehousing scenario where the size of relations is very large and many different analytic queries are run over the same relations. PRepDB generates all the repair samples as a preprocessing step, so that the query is executed once over the relations involved. Materialization also allows PRepDB to use optimizations techniques such as indexing to be used.

Second, when the size of dirty relations and the number of samples is large, the answer relation (table A in Algorithm 1) can grow very large. This not only takes a lot of space, but the query processing efficiency also suffers. The major problem is redundancy, as in typical data cleaning scenarios the majority of data is expected to be clean, which will be unnecessarily replicated over each sample. PRepDB exploits this commonality by storing each unique possible only once, improving both the space efficiency and query processing speed.

4.4 Query Engine Operators

In this section, we describe the operators that are necessary to process probabilistic queries over repair samples. In Figure 4.1, these operators are part of the *Query Engine*, which uses the repair samples stored in the *Repairs Store* to produce probabilistic results to the user's query.

4.4.1 P-Tuples

In contrast to traditional query processing, the PRepDB query pipeline contains probabilistic tuples. We introduce *p-tuples*, which extend the traditional tuples to allow probabilistic processing over materialized repair samples. P-tuples may contain uncertain values for some of its attributes, which we denote as *u-cells*. They also contain an additional $SIDS$ field that stores a set of sample IDs, representing the worlds in which the current *value assignments* to this p-tuple are valid.

Conceptually, p-tuples are similar to *tuple bundles* in MCDB. However, tuple bundles are designed for the on-the-fly sample-based query processing. The u-cells in tuple bundles are represented as an array of possible values, with the value at index i representing the value that was generated for sample ID i. In contrast, p-tuples are designed for processing over materialized samples, where the possible values for u-cells are mapped to the set of sample IDs they have appeared in. Thus, the u-cells in p-tuples can initially be represented with a '?' placeholder, and the specific possible values along with their sets of sample IDs retrieved from storage when necessary.

Figure 4.3 extends the example in Figure 1.1 with four additional repair samples, with their respective sample IDs (SIDs). For brevity, only the first two tuples are considered. The tuple

$$t_1 = [Patrick, ?, ?]$$

is a p-tuple that is valid in all possible worlds, since this particular value assignment appears in all samples. That is, since the attributes $City$ and $Area$ are uncertain in t_1, they are free to take any value. Instead of storing the sample ID set $\{1, 2, 3, 4, 5, 6\}$ under $t_1.SIDS$, we simply set the value to $NULL$, which will signify the validity in all the samples.

Now consider the same tuple after we fix a certain value for the $City$ attribute:

$$t_1 = [Patrick, Queens, ?]$$

This particular value combination only appears in samples $3, 4, 5, 6$. Thus, $t_1.SIDS = \{3, 4, 5, 6\}$. Finally, we fix the value for the remaining $Area$ attribute:

$$t_1 = [Patrick, Queens, 347]$$

This value assignment is observed in samples 4 and 5. When this tuple is written to output, its probability can be calculated as $2/6$, since it appears in 2 out of 6 possible samples.

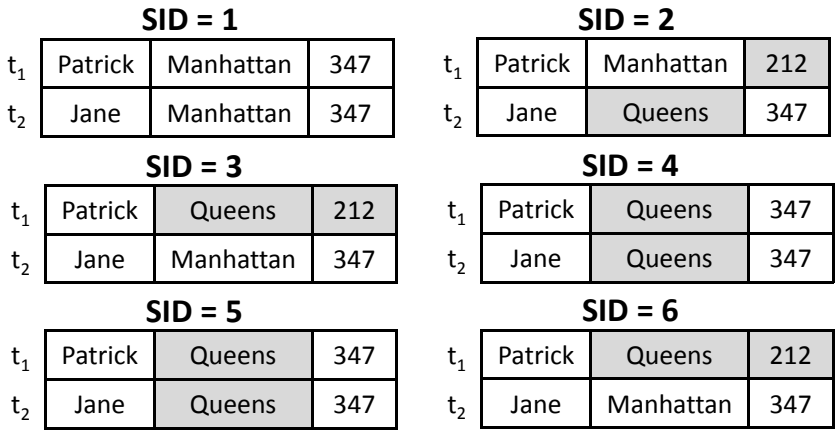


Figure 4.3: Example of generated repair samples

4.4.2 P-Tuple Expansion

In general, we call the process of assigning values to uncertain cells of a p-tuple *tuple expansion*. The presence of u-cells in a p-tuple means that multiple combinations of value assignments exist, each producing a different p-tuple with its corresponding probability.

This operation is conceptually similar to the *split* operator on tuple bundles in MCDB. However, the split operator acts on the generated array of possible values, while p-tuple expansion uses the materialized mapping of values to sample IDs.

In the example above, the original p-tuple could be expanded to four p-tuples, which represent the distinct value assignments in the sampled repairs to the uncertain attributes: $[Manhattan, 347]$, $[Manhattan, 212]$, $[Queens, 212]$ and $[Queens, 347]$. In Section 8 we show how to perform this expansion and assign the corresponding $SIDS$ to each valid value assignment in an efficient way.

4.4.3 Relational Selection

The traditional selection operator takes as input one tuple t and filters it according to the predicate condition c. In our case, if c refers only to constant attributes in t, classical selection is used. Otherwise, the uncertain attributes in t that are required by c need to be expanded. Only p-tuples with value assignments that satisfy c pass the filter.

For tuple bundles, the possible value arrays of u-cells required by c need to be scanned in

order to find which SIDs, if any, satisfy c. In contrast to this top-down approach, materialized repair samples enable much faster bottom-up filtering of p-tuples, as we describe in Section 7.3.

4.4.4 Projection

The projection operator is PRepDB is almost identical to its traditional counterpart. Both certain and uncertain attributes can be simply removed, while the $SIDS$ field is unaffected.

4.4.5 Cartesian Product and Join

The cartesian product between two relations R and S is performed as follows.

$$R \times S = \{t = r \cup s, t.SIDS = r.SIDS \cap s.SIDS | r \in R, s \in S\}$$

Each pair of p-tuples results in a concatenated p-tuple and their sets of sample IDs are intersected.

The join operation with a predicate condition c is equivalent to applying the cartesian product, followed by a relation selection with c.

Chapter 5

Baseline Representation System

In this section we present a baseline representation system for PRepDB. The main goals of this system are: the ability to store a large set of repair samples with cell level value correlations, avoid redundantly storing possible values, and enable efficient relational query evaluation.

Our baseline is a decomposition representation system (*DECOR*), inspired by the storage model of MayBMS [2]. MayBMS decomposes all possible worlds into sets of independent cells, each assigned some probability value. In contrast, DECOR does not assume commonality for arbitrary sets of cells and decomposes the possible worlds at the level of single cells. Further, DECOR is designed to store repair samples in a way that enables sample-based query evaluation, instead of the event model semantics used in MayBMS.

For each dirty relation D in the database, DECOR adds two additional relations with the following schema. *Template* relation D^T has the same schema as D, with additional metadata to signify uncertain cells (*u-cells*). U-cells are those cells for which the possible values differ across the sampled repairs. In contrast, the cells that contain the same value across all repairs store this certain value in D^T. Certain values could mean that either the cell is clean or it has a unique repair across all samples. The idea of a template relation has been previously proposed by MayBMS.

Values relation D^V stores the possible values for u-cells of D. It has a fixed schema with columns `TID, AID, VAL, SIDS`. `TID` and `AID` identify the cell using the tuple and attribute IDs. `VAL` stores a possible value for the specified cell. `SIDS` stores the set of sample IDs that the possible value stored in `VAL` appears in. For efficient access by the query engine operators, a multicolumn index on the `TID,AID,VAL` attributes is created.

Figure 5.1 illustrates how the six repairs shown in Figure 4.3 get stored using the DECOR system. Observe that the values for the `Name` column as well as the `Area` column for t_2 remain

Customers^T

	Name	City	Area
t_1	Patrick	?	?
t_2	Jane	?	347

Customers^V

TID	AID	VAL	SIDS
t_1	City	Manhattan	1, 2
t_1	City	Queens	3, 4, 5, 6
t_1	Area	347	1, 4, 5
t_1	Area	212	2, 3, 6
t_2	City	Manhattan	1, 3, 6
t_2	City	Queens	2, 4, 5

Figure 5.1: Baseline Representation System for Repair Samples

the same across all the repairs. Thus, the $Customers^T$ relation contains certain values for these cells, while the u-cells are specified using a '?' placeholder. The possible values for each of the u-cells are stored in the $Customers^V$ relation. For example, the value of $City$ for t_1 is $Manhattan$ in repairs with sample IDs 1 and 2, so a tuple

$$[t_1, City, Manhattan, \{1, 2\}]$$

is stored in the $Customers^V$ relation.

Traditional RDBMS do not allow storing different data types under one column, so in practice the values under `VAL` attribute can either be stored as text and then cast to the original data type at application level, or a separate *Values* relation could be used for each data type. The set of sample IDs for the `SIDS` column can be represented as a set of integers at the application level and stored in a serialized form inside the relation.

The DECOR system stores each possible value for a cell exactly once. Since the repair samples usually contain a high degree of commonality and overlap, this allows a large number of possible repairs to be stored in a very efficient manner. Explicitly storing the sample IDs allows DECOR to preserve the value correlations that may occur at the cell level.

Chapter 6

Improved Representation System

In this section we present an improved representation system that builds on the ideas in DECOR. The VA table representation (VATAR) system exploits the tuple level commonalities that occur in the repairs of FD violations, as we demonstrate empirically in our experiments. This greatly improves the efficiency of joins and enables space efficient in-memory caching.

For each dirty relation D in the database, VATAR uses four relations to represent the repair samples. The $Template$ relation D^T is the same as in DECOR. A VA relation D^VA stores the value assignments on tuple level and has a schema with four attributes `TID,VAID,VA,SIDS`. The `TID` column stores the tuple identifier, the `VAID` column stores the unique ID of the VA stored in the `VA` column. The `SIDS` column stores the sample IDs the value assignment stored in `VA` occurs in.

The $Values$ relation D^V is similar to DECOR, with schema `TID,AID,VAL,VAIDS`, where the `VAIDS` column stores the VA IDs in which the value stored in `VAL` occurs. The purpose of this relation is to provide an index into VA table that could be used by the Filter operator which could require values for specific uncertain attributes.

Finally, the $VAAID$ relation D^VAAID stores the attribute structure of the VAs for each tuple stored in the VA relation. The schema has attributes `TID,AIDMAP`, where `AIDMAP` stores the description of the attribute structure for its corresponding tuple. Each tuple in a $Template$ relation may have a different number of u-cells, for different attributes, so the $VAAID$ allows the VAs for these u-cells to be stored in a dynamic form inside the VA relation.

Figure 6.1 shows how the repair samples from Figure 5.1 are stored using the VATAR system. Note that even though VATAR does not present space savings over DECOR, the query efficiency is greatly improved. This comes from the fact the the root-to-leaf paths are essentially materialized in the *VA* table and can be efficiently accessed with the index represented by the *Values*

Customers^{T}

	Name	City	Area
t_1	Patrick	?	?
t_2	Jane	?	347

Customers^{V}

TID	AID	VAL	VAIDS
t_1	City	Manhattan	0, 1
t_1	City	Queens	2, 3
t_1	Area	347	0, 3
t_1	Area	212	1, 2
t_2	City	Manhattan	0
t_2	City	Queens	1

Customers^{VAAID}

TID	AID
t_1	[City, Area]
t_2	[City]

Customers^{VA}

TID	VAID	VA	SIDS
t_1	0	[Manhattan, 347]	1
t_1	1	[Manhattan, 212]	2
t_1	2	[Queens, 212]	3, 6
t_1	3	[Queens, 347]	4, 5
t_2	0	[Manhattan]	1, 3, 6
t_2	1	[Queens]	2, 4, 5

Figure 6.1: VA Table Representatoin System for Repair Samples

table. Since there is a high degree of commonality on tuple level for repairs of FD violations, the number of root-to-leafs stays manageable.

Chapter 7

Query Processing over DECOR

In this section we introduce new operators, as well as modifications to existing operators, that are necessary for efficient query processing over the DECOR system.

7.1 Value Mapper

The *Value Mapper* module acts as an interface with the $Values$ relation and is used by a number of other modules in the system to process p-tuples.

The module accepts as input a relation identified (RID), a tuple identifier (TID) and a set of attribute identifiers $(Attrs)$, and outputs a set of maps $\mathcal{M}$, with $\mathcal{M}_A$ for each $A \in Attrs$. The possible values are obtained from the RID^V values relation.

An output map $\mathcal{M}_A$ contains key-value pairs (ν, ψ), associating a possible value ν of attribute A with its corresponding set of sample IDs ψ (stored as $SIDS$ in $Values$ relation).

The Value Mapper also supports calls that additionally specify the value (VAL), in which case only $SIDS$ need to be returned.

7.2 Unfold Operator

As a p-tuple goes through the query pipeline, it makes sense to expand its u-cells *lazily*, i.e. only when a value for a certain attribute is required by an operator. Expanding uncertain attributes could be expensive, as they could represent a large number of possible combinations, and if

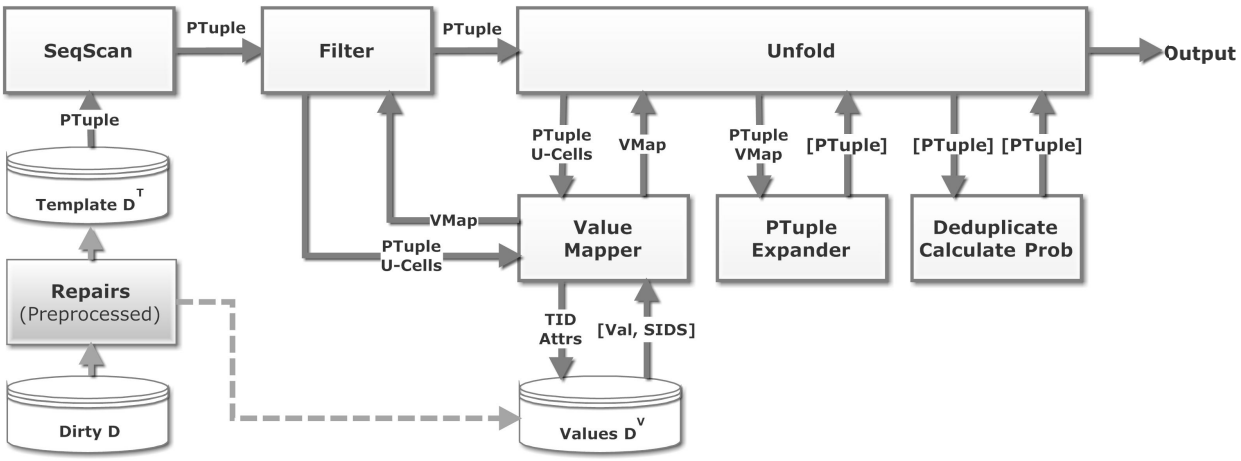


Figure 7.1: PRepDB Query Engine diagram for Select with Condition Operation

the attribute ends up projected away or the tuple is filtered out, the expansion could have been unnecessary. However, as the p-tuple reaches the end of the pipe, the remaining u-cells need to be expanded before it is ready for output.

The *Unfold* operator accepts p-tuples as input and outputs tuples with complete value assignments and their corresponding probabilities. For each input p-tuple, the possible values for the remaining u-cells are obtained using the Value Mapper. We rely on the query executor to keep track of linage of each cell, i.e., the attribute ID and relation it came from. The obtained value maps are then used to expand all of the remaining uncertain attributes.

Finally, the operator performs duplicate removal by merging p-tuples with identical value assignments and taking the union of their sets of sample IDs stored in the $SIDS$ field. After this step, the probability of each tuple is calculated by dividing the size of its $SIDS$ by the total number number of samples in the database, and the result is written to output.

7.3 Selection with Condition

For selection with conjunctive conditions, the following optimizations are enabled by DECOR storage system. First, the filtering is performed over certain attributes. If the tuple passes the first test, the u-cells are tested before proceeding with the expansion. That is, for each u-cell, we make sure that the values required by the condition exist as a possible value. However, this test is not enough to pass the filter, as it does not guarantee that the required *combination* of possible values

ever occurs in one of the samples. For this, we need to fully expand the required u-cells and test that the resulting $SIDS$ is not empty. Nevertheless, this strategy allows multiple opportunities for early termination before performing the expensive expansion operation.

Observe that the filter relies heavily on the fast indexed access to the $Values$ tables. These tables can be much larger than the original dirty database, so performing a linear scan for each input tuple would be unacceptable.

Figure 7.1 shows the diagram of the query engine executing a selection with condition query. Conceptually, the user is running the query over the dirty relation D. However, invisible to the user, PRepDB has generated a number of possible repairs for D and stored them into a *template* and *values* relations of the DECOR system. This is done as a preprocessing step, before any queries are actually executed. So when a query actually runs, it uses only the data stored in DECOR.

First, a sequential scan is performed over the template relation. Each p-tuple is then filtered according to the predicate condition, while accessing the Value Mapper when required for u-cells. Then, the remaining u-cells are expanded in the Unfold operator. Finally, duplicates are removed and the resulting tuple with its probability is written to the output.

We now show how a query based on Example 1 is evaluated over the database in Figure 5.1. Consider the query `SELECT * FROM Customers WHERE City = 'Queens'`. We start with a sequential scan over the template relation of the $Customers$ table, $Customers^T$. The first p-tuple is:

$$[t_1, Patrick, ?, ?] \;\; t_1.SIDS = NULL$$

Attributes $City$ and $Area$ for t_1 contain u-cells and the $SIDS$ field is initially set to $NULL$.

Next, t_1 is passed to the filter. The predicate requires a value for the u-cell under $City$, so the filter calls $ValueMapper(t_1, City, Queens)$, which returns the SIDS $\{3, 4, 5, 6\}$. Since the return was not empty, this p-tuple passes the filtering, and is updated to:

$$[t_1, Patrick, Queens, ?] \;\; t_1.SIDS = \{3, 4, 5, 6\}$$

The Unfold operator then identifies the remaining u-cells, in this case just the $Area$ attribute. It calls $ValueMapper(t_1, Area)$, which returns a value map with key-value pairs $(347, \{1, 4, 5\})$ and $(212, \{2, 3, 6\}$. The tuple is expanded using this value map into two p-tuples:

$$[t_1, Patrick, Queens, 347] \;\; t_1.SIDS = \{4, 5\}$$

$$[t_1, Patrick, Queens, 212] \;\; t_1.SIDS = \{3, 6\}$$

These tuples can then be written to output, both assigned with probability of $2/6$.

7.4 Equijoin

The equijoin operator concatenates two tuples r and s, from two relations R and S respectively, only if their values match on a given set of attributes $Attrs$. We implement this operator as a hash join. First, each tuple $r \in R$ is hashed w.r.t. its values in $Attrs$ into a map $\mathcal{H}$, s.t. $\mathcal{H}[r[Attrs]] \cup = r$. If $r[Attrs] \notin \mathcal{H}$, we first insert an empty list into $\mathcal{H}[r[Attrs]]$. Once all of R is processed, each tuple in $s \in S$ is concatenated with each tuple stored in $\mathcal{H}[s[Attrs]]$.

When we encounter a tuple t with at least one u-cell in $t[Attrs]$, in either R or S, t is first expanded w.r.t. its u-cells in $t[Attrs]$. Then, each tuple that resulted from the expansion is processed normally.

For example, suppose we are processing tuples $r_0, r_1, ..., r_{|R|}$ in R. For each r_i with no u-cells in $r_i[Attrs]$, we append r_i to the list stored at $\mathcal{H}[r_i[Attrs]]$. If $'?' \in r_i[Attrs]$, we first expand r_i w.r.t. u-cells in $Attrs$. Suppose the expansion of r_i produces p-tuples $r_i^1, r_i^2, ..., r_i^n$. Then for each r_i^j, we set $\mathcal{H}[r_i^j[Attrs]] \cup = r_i^j$.

Chapter 8

P-Tuple Expansion

Expanding a set of attributes for a p-tuple is a critical operation in PRepDB query processing. In this section, we show how to efficiently perform expansion of DECOR p-tuples. In Section 9, we extend the concepts introduced here to the VATAR system.

Expanding a single uncertain attribute of a p-tuple is easy. Consider a p-tuple t with a u-cell at $t[A]$ that has k possible values. This means that the call $ValueMapper(t, A)$ would return the value map $\mathcal{M}$ with k key-value pairs$(\nu, \psi) \in \mathcal{M}_A$) Thus, t would expand into k new p-tuples. The value $t'.SIDS$ for each resulting tuple with $t'[A] = \nu$ is obtained by intersecting the corresponding ψ with $t.SIDS$.

On the other hand, expanding multiple attributes for one p-tuple results in exponential number of possibilities. A p-tuple with m uncertain attributes, each having k possible values, corresponds to m^k possible value combinations.

8.1 Naive Expansion

A naive algorithm would enumerate each one of these possibilities, intersecting the sample sets along the way. Algorithm 2 illustrates this method.

The runtime complexity of main loop is m^k. However, we also need to account for the intersection operation on line 8. At each level of recursion (representing an attribute), there can be at most k non-empty sets of sample IDs (SIDS), of average size N/k, where N is the total number of repair samples. The rest will contain empty SID sets.

Algorithm 2 Naive P-Tuple Expansion

```
1:  procedure EXPANDTUPLE(t, Attrs, M)
2:      return RecurseExpand(t, Attrs, M, 0, {})

3:  procedure RECURSEEXPAND(t, Attrs, M, i, T))
4:      if i = |Attrs| then
          return {t}
5:      else
6:          for each (ν, ψ) ∈ M_Attrs[i] do
7:              t[Attrs[i]] = ν
8:              t.SIDS ← t.SIDS ∩ ψ
9:              T∪ = RecurseExpand(t, Attrs, M, i + 1, T)
10:         return T
```

Each of these sets will need to intersect with every SID set of the attribute at the next level, which also has at most k non-empty SID sets of size N/k each. Thus, the intersection will take $N \cdot k$ operations at each attribute level in the worst case. The total runtime complexity is then $O(m^k + N \cdot k \cdot m)$.

It is clear from this analysis that if k is of the same order as N, most of the SIDS at deeper levels will be empty. A p-tuple with an empty SIDS represents a value assignment that never occurred in any of the repair samples, and thus must be discarded.

An obvious optimization to the naive algorithm is then to avoid propagating a p-tuple with empty SIDS. This can be done easily by incorporating a check that executes line 9 only when $|t.SS| > 0$. This optimization prunes the recursion tree, such that each level will have at most k sets of SIDS of size N/k, all of them non-empty. Following the preceding analysis, the overall complexity becomes $O(N \cdot k \cdot m)$.

In practice, the average cardinality (k) of u-cells grows linearly with the number of repair samples (N), for reasonables size of N. Thus, even with the pruning optimization, the naive algorithm will be roughly quadratic in N.

8.2 Cardinality-Invariant Expansion

In the sections that follow, we propose a cardinality-invariant expansion algorithm that runs linear in N, with an overall complexity $O(N \cdot m)$. This new approach uses the idea of partition products

to intersect the SID sets in a bottom-up fashion.

An important observation is that the SID sets of a u-cell with cardinality k represent a *partition* of the set $\{1, 2, ..., N\}$, which we denote as $\mathcal{S}$, into k non-empty parts. This is because each repair sample produces exactly one value for each u-cell, assigning its sample ID to one of k possible values.

Consider the p-tuple t_1 in Figure 5.1. The SID sets of each of the uncertain attributes partition $\mathcal{S}$ into two parts:

$$\pi_{City} = \{\{1, 2\}, \{3, 4, 5, 6\}\} \;\; \pi_{Area} = \{\{1, 4, 5\}, \{2, 3, 6\}\}$$

The product of these two partitions is calculated by taking the union of the intersections in the cartesian product of the parts between the two partitions:

$$\pi_{City,Area} = \{\{1\}, \{2\}, \{3, 6\}, \{4, 5\}\}$$

When we expand the u-cells of a p-tuple, we are logically performing the same operation, i.e., taking the cartesian product of possible values and intersecting the corresponding SID sets. Thus, the parts of $\pi_{City,Area}$ represent the SID sets of the expanded p-tuples w.r.t. $City$ and $Area$. In our example, we can easily verify that these SID set correspond to the following value assignments to t_1: $[Manhattan, 347]$, $[Manhattan, 212]$, $[Queens, 212]$ and $[Queens, 347]$.

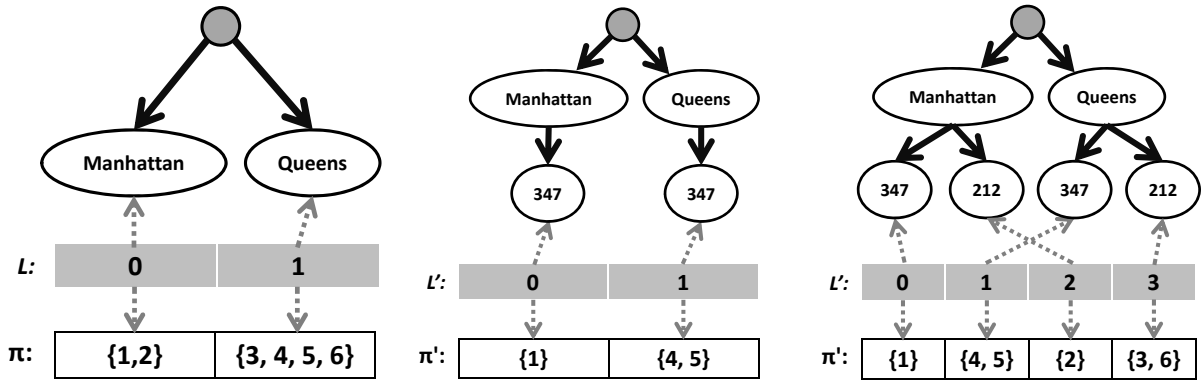
It is possible to compute the product of two partitions of the set $\{1, 2, ..., N\}$ in $O(N)$ time, regardless of the number of parts within each of these partitions. Thus, the cumulative product of m such partitions will take $O(N \cdot m)$ time and produce the SID sets of the expanded p-tuples with respect to m attributes.

However, as we show in the next section, it not trivial to use the linear time partition product algorithm to also keep track of the value assignment to each of the expanded p-tuples without incurring a non-constant overhead.

8.3 Naive Value Assignment Tracking

One way to track the value assignments (VA for brevity) is to keep a separate list in which each entry corresponds to the VA for a part in the current cumulative partition.

In our running example, we start with partition π_{City} represented as a list: $[\{1, 2\}, \{3, 4, 5, 6\}]$, and the VA list $\nu_{City} = [[Manhattan], [Queens]]$.



(a) $City$ initialized (b) $Area = 347$ processed (c) $Area = 212$ processed

Figure 8.1: Example of constructing a VA Tree for p-tuple expansion

When performing the partition product with π_{Area}, as we add each new part to $\pi_{City,Area}$, we also add the corresponding VA to $\nu_{City,Area}$. Thus, we get two resulting lists: $[\{1\}, \{2\}, \{3, 6\}, \{4, 5\}]$ and $[[Manhattan, 347], [Manhattan, 212], [Queens, 212], [Queens, 347]]$.

The problem with this approach comes from the overhead of maintaining the VA list. For each resulting part in the product of two partitions, it is necessary to copy over the corresponding entry in the initial VA list and append it with the overlapping value in the second partition.

Suppose that in our example, t_1 has an additional uncertain attribute $Gender$, with value-map $\{Male : \{1, 2, 3, 4, 5\}, Female : \{6\}\}$. One of the resulting parts in $\pi_{City,Area,Gender}$ is $\{3\}$, for which the corresponding entry $[Queens, 212, Male]$ is added to $\nu_{City,Area,Gender}$. The $[Queens, 212]$ prefix needs to be copied and not simply appended to, because the SID set for $Female$ also intersects with that part, i.e. one-to-many relationships are possible.

In the worst case, when there are N distinct repairs and we expand a tuple with m uncertain attributes, $O(N \cdot m^2)$ copy operations are performed. This can cause serious performance issues in cases involving joins of many uncertain relations, which could result in a large number of uncertain attributes.

8.4 Value Assignment Trees

In this section, we propose a way to track VAs with only a constant time overhead. We dynamically build a VA Tree as we perform partition products, adding a level for each consecutive attribute, while maintaining an index to all the leaves (bottom level of the tree).

The VA Tree is an empty-rooted ordered tree, where each node stores a possible attribute value, and all valid VAs to the uncertain attributes are represented by root-to-leaf paths in the tree. The index to the bottom level links each SID set in the cumulative partition product to its corresponding VA.

Algorithm 3 Cardinality-Invariant P-Tuple Expansion

procedure EXPANDTUPLE($t, Attrs, \mathcal{M}$)
    Initialize cumulative partition π, leaf index $\mathcal{L}$, VA Tree $\mathcal{T}$ with an empty root
    $i \leftarrow 0$
    for each $(\nu, \psi) \in \mathcal{M}_{Attrs[0]}$ **do**
        $\eta \leftarrow$ New tree node with value ν
        Add η as child at $\mathcal{T}$
        $\mathcal{L}[i] \leftarrow$ pointer to η
        $\pi[i] \leftarrow \psi \cap t.SIDS$
        $i \leftarrow i + 1$
    for $m = 1$ to $|Attrs| - 1$ **do**
        Initialize/reset inverted index array $\mathcal{S}$
        for $i = 0$ to $|\pi|$ **do**
            for each $j \in \pi[i]$ **do**
                $\mathcal{S}[j] \leftarrow i$
        $i \leftarrow 0,\ \pi' \leftarrow \varnothing,\ \mathcal{L}' \leftarrow \varnothing$
        for each $(\nu, \psi) \in \mathcal{M}_{Attrs[m]}$ **do**
            Initialize/reset intersection index hashmap $\mathcal{R}$
            for each $j \in \psi$ **do**
                $\mathcal{R}[\mathcal{S}[j]] \leftarrow \mathcal{R}[\mathcal{S}[j]] \cup j$
            for each $(j, \rho) \in \mathcal{R}$ **do**
                $\eta \leftarrow$ New tree node with value ν
                Add η as child to node at $\mathcal{L}[j]$
                if $m = |Attrs|$ **then**
                    Store ρ at η
                else
                    $\mathcal{L}'[i] \leftarrow$ pointer to η
                    $\pi'[i] \leftarrow \rho$
                    $i \leftarrow i + 1$
        $\pi \leftarrow \pi',\ \mathcal{L} \leftarrow \mathcal{L}'$
    Traverse $\mathcal{T}$ and return the resulting expanded tuples and their SID sets

Algorithm 3 shows how to expand a p-tuples by constructing a VA tree. The input is a p-tuple t, a set of uncertain attributes $Attrs$ and a set $\mathcal{M}$ containing a value map $\mathcal{M}_A$ for each $A \in Attrs$. Figure 8.1 follows the example of expanding the tuple t_1 from Figure 5.1 w.r.t attributes $City$ and $Area$.

Lines 4-9 initialize the first attribute. A new tree node is added for each possible value ν of attribute $Attrs[0]$, while the cumulative partition π and the leaf index $\mathcal{L}$ are initialized to the SID set ψ of the current value being processed. Thus, π gets initialized to the SID set of the first attribute and the leaf index connects tree leaf containing the value ν with its corresponding SID set in π. Figure 8.1a shows the resulting tree, leaf index $\mathcal{L}$ and partition π after initializing the $City$ attribute.

The loop in lines 10-29 processes the rest of the uncertain attributes. In lines 12-14, we build the inverted index array for the current partition π, which maps each sample ID to the part in π it belongs to. The for loop in lines 16-28 intersects the SID sets of the current attribute $Attrs$ with the partition π. Partition π' and leaf index $\mathcal{L}'$ represent the next level of the tree. We iterate over each sample ID in the current SID set ψ and intersect it with the part in π with the same SID, accessed by the inverted index we have built. In our example, suppose we process the $(347, \{1, 4, 5\})$ pair for the $Area$ first. SID 1 intersects with part $\{1, 2\}$ in π, which is at index $\mathcal{S}[1] = 0$, so we set $\mathcal{R}[0] = \{1\}$. Both SIDs 4 and 5 intersect with $\pi[1]$, so $\mathcal{R}[1] = \{4, 5\}$.

Each intersection results in a tree node for the current attribute, processed in lines 20-28. The leaf index of the previous level $\mathcal{L}$ is used to determine the parent node of the current intersection. If the $Attrs[m]$ is the last attribute to be processed, we store the resulting SID set in the leaf node, so that it can be easily accessed when the root-to-leaf paths are enumerated. Figure 8.1b shows the resulting VA tree after processing $Area = 347$. For illustration purposes, we assume that $Area$ is not the last attribute, so $\mathcal{L}'$ and π' still get updated. Figure 8.1c shows the final tree for the attributes $City$ and $Area$.

To expand tuple t_1, we traverse all root-to-leaf paths, each one resulting in a new p-tuple with its $SIDS$ stored at the leaf of the path. In our example, expansion of t_1 results in the following four p-tuples:

Name	City	Area	SIDS
Patrick	Manhattan	347	$\{1\}$
Patrick	Manhattan	212	$\{2\}$
Patrick	Queens	347	$\{4, 5\}$
Patrick	Queens	212	$\{3, 6\}$

If there were more uncertain attributes, we could continue the process, computing partition products and adding further levels to $\mathcal{T}$ and updating π.

8.5 Complexity Analysis of Algorithm 3

Computing the inverted index $\mathcal{S}$ (lines 12-14) takes $O(N)$ per attribute, since there are exactly N sample IDs for each uncertain cell. Computing the intersection index $\mathcal{R}$ (lines 18-19) also takes $O(N)$ per attribute. In the worst case, $\mathcal{R}$ will contain N intersections, but processing each one (lines 20-28) takes constant time, so the overall runtime is $O(N)$ per attribute. Thus, processing all attributes (the full loop at lines 10-29) takes $O(N \cdot m)$.

In the worst case, the number of VAs is at most N, when each repair is distinct, and for each distinct VA, the root-to-leaf path visits exactly m nodes (height of $\mathcal{T}$). Thus, iterating $\mathcal{T}$ to produce all valid VAs and their sample sets (line 30) takes $O(N \cdot m)$.

Chapter 9

Query Processing over VATAR

In this section, we show how the VATAR system can enable more efficient processing of certain queries.

9.1 VATAR P-Tuples

In contrast to DECOR, there is no longer needs to explicitly store the sample IDs (SIDS) for each p-tuple as it moves through the query plan. Instead, each p-tuple t stores a $Relation \rightarrow VAIDS$ map in the $t.VAIDS$ field, which defines which value assignments are valid for t at the relation level.

Suppose that tuple t_1 from Figure 6.1 is conditioned on $City = Queens$. The $t_1.VAIDS$ would then contain $Customers : \{2, 3\}$, since those are the value assignment IDs that match the condition. If this tuple is conditioned further to $Area = 212$, the corresponding VAIDS would be updated to $\{2\}$. This is obtained by interesting the previous VAID set $\{2, 3\}$ with the VAID set where $t_1[Area] = 212$, namely $\{1, 2\}$.

The sample IDs (SIDS) stored in the *VA* table are only accessed when p-tuple probability is calculated before output and when expanding across multiple uncertain relations.

9.2 Joins, Expansion and Relation-level VA Trees

When performing a join between two relations R and S in DECOR, a p-tuple r from R can join with a large number of tuples $s_1, s_2, ..., s_n$ from S. During the Unfold operation, the VA tree

Orders^{T}

	Name	Item	Price
s_1	Patrick	?	?

Orders^{VAAID}

TID	AID
s_1	[Item, Price]

Orders^{VA}

TID	VAID	VA	SIDS
s_1	0	[iPhone 5, $900]	1, 3, 4, 5
s_1	1	[Galaxy S4, $700]	2, 6

Figure 9.1: An Orders relation stored using VATAR

for u-cells in r have to be rebuilt for each intermediate tuple $r \cdot s_i$. Caching the whole VA tree for each p-tuple is too expensive, and it is not obvious how to dynamically update the tree in the presence of differences in intermediate tuples of r that stem from different u-cells from S or conditions on their values.

In contrast, the VA tree for the intermediate tuples can be built at relation level in VATAR, using a relation-level VA tree (RVAT). Each level in RVAT represents a relation and each node stores unique value assignments for that relations that are attached to the p-tuple being expanded. The SIDS attached to each VA still represent a partition of the complete sample ID set, so taking the partition product correctly computes the SIDS that result from expanding across multiple relations. The paths of RVAT are greatly compressed versions of the paths of the equivalent VA trees, so the expansion can be computed much faster. This also allows the VAs to be cached for each p-tuple, so that the VA tree does not have to be rebuilt for each intermediate tuple.

We illustrate this concept using an example of a join between two relations. Consider an additional uncertain relation $Orders$ that stores the customer orders. Figure 9.1 shows an instance of such relation, with some VATAR tables omitted for brevity.

Now suppose we run the query described in Example 1 that joins the customers with their orders in order to get the total amount of orders in Queens. The resulting RVAT when expanding the intermediate tuple $t_1 \cdot s_1$, from $Customers$ and $Orders$ respectively, is shown in Figure 9.2.

The first level of the tree represents the VAs for attributes from the $Customers$ relation. Since we conditioned the city on Queens, only the corresponding VAs are considered for this p-tuple. The partition for this level is as follows (using Figure 5.1):

$$\pi_{Customers} = \{\{3, 6\}, \{4, 5\}\}$$

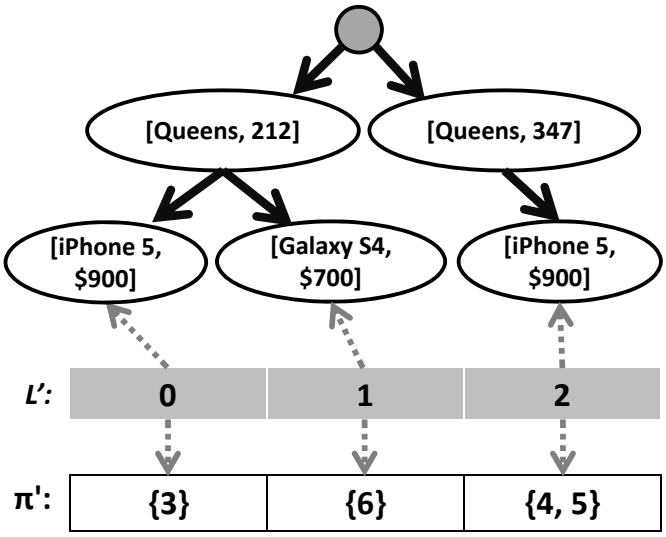


Figure 9.2: Example of a Relation-level VA Tree

The second level represents the VAs for attributes from the $Orders$ relation, with partition

$$\pi_{Orders} = \{\{1, 3, 4, 5\}, \{2, 6\}\}$$

The sample IDs that resulted from the partition product $\pi_{Customers,Orders}$ are stored in π'.

Chapter 10

Experimental Study

In this section, we evaluate our system. First, we show that the system is able to efficiently scale to large datasets. Second, we perform a qualitative study that compares quality results obtained from our system to those of competing approaches.

10.1 Setup

All experiments were performed using a system with an Intel Core i5 3.3GHz processor and 8GB of RAM. The system was implemented in Python 3.2, and PostgreSQL 9.2 was used for the underlying storage. We use the UIS Database Generator to generate synthetic data that imitates a real personal contact relation with the following schema: `SSN, FirstName, MiddleInit, LastName, StNum, StAddr, Apt, City, State, ZIP`. The following FDs are defined over this schema:

- `SSN` $\rightarrow$ `FirstName, MiddleInit, LastName, StNum, StAddr, Apt, City, State, ZIP`
- `FirstName, MiddleInit, LastName` $\rightarrow$ `SSN, StNum, StAddr, Apt, City, State, ZIP`
- `ZIP` $\rightarrow$ `City, State`

For the performance experiment involving an equijoin query, we normalize the relation into two tables using the FD `ZIP` $\rightarrow$ `City, State` and make repairs to the two relations separately.

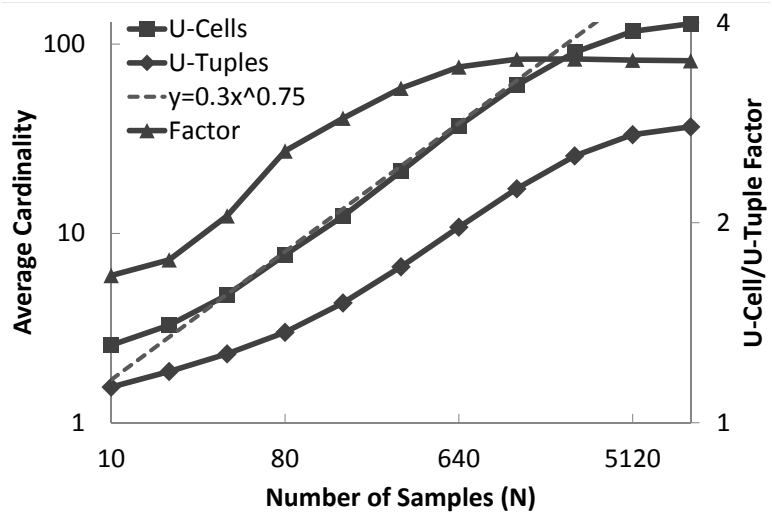


Figure 10.1: Dataset Characteristics

We add inconsistency to the database as follows. First, we generate a clean dataset using UIS. After that, we make value modifications to random cells until we reach the target perturbation rate, expressed as a percentage of all the cells. Our perturbation model makes sure that each cell modification causes an FD violation. For each cell change, first an FD is chosen at random, then we randomly choose to modify either the LHS or the RHS. For LHS perturbations, we select two random tuples that differ in the chosen FD's LHS and RHS attributes, and make modify the values such that the LHS values become equal. For RHS perturbations, we select two tuples with equal LHS values and make the RHS attribute different.

10.2 Performance Evaluation

We evaluate the performance of our system on a 5000 tuple UIS dataset with a 5% perturbation rate. We choose to vary the number of repair samples parameter, as it gives direct control over the size of the uncertain database, i.e. the number of distinct repairs, which ultimately impacts the system performance.

The log-log plot in Figure 10.1 shows the characteristics of the dataset with varying number of samples. The average number of possible values (cardinality) of uncertain cells (u-cells) grows exponentially for low number of samples, as each repair likely to be distinct. Between 40 and 1000 samples, the grows is sublinear, after which the space begins to saturate and each new sample is less likely to be distinct. On the other hand, the average cardinality of u-tuples, i.e. the

average number of possible value assignments to the u-cells of the u-tuple, grows slower, with the factor against u-cells growing from 2 to 4 as the number of samples increase. This supports the motivation behind VATAR, which attempts to exploit the tuple level commonality for efficiency.

We compare the performance of the following approaches:

- `Naive`: This approach executes the query over each repair sample explicitly and then aggregates the results.
- `DECOR`: This approach uses the baseline representation system described in Section 5.
- `VATAR`: This approach uses the improved representation system that utilizes tuple compression, as described in Section 6.
- `VATAR+C`: This approach enables in-memory caching of the VA and $VAAID$ tables.
- `VATAR+C(2nd)`: This is the second run with caching enabled, i.e. minimal DB IO required.

Only the runtime for `Naive` method includes repair generation time, since for other methods repairs are generated as a preprocessing step.

We use the following queries:

- Q_1: `SELECT * FROM UIS`
- Q_2: `SELECT * FROM UIS WHERE FirstName='Demby' AND LastName='Rowsso'`
- Q_3: `SELECT * FROM UIS NATURAL JOIN ZIPCODES`

The log-log plots in Figures 10.2-10.4 show the runtime performance for the three queries. For Q_1, shown in Figure 10.2, even the baseline representation system (DECOR) is several orders more efficient than the naive method. Moreover, the caching enabled VATAR system runs another order of magnitude faster than DECOR.

Query Q_2 is much more selective and takes advantage of indexing. As a result, the gap with the Naive method is even wider in this case. Since the result set is already very small for this query, tuple compression in VATAR does not bring as much advantage over DECOR. However, the ability to cache on tuple level does improve performance by an order of magnitude.

The join evaluation in query Q_3 brings out the redundancy in DECOR discussed in Section 9.2, as shown in Figure 10.4 Thus, the tuple-level compression in VATAR provides up to two orders of magnitude improvement over DECOR.

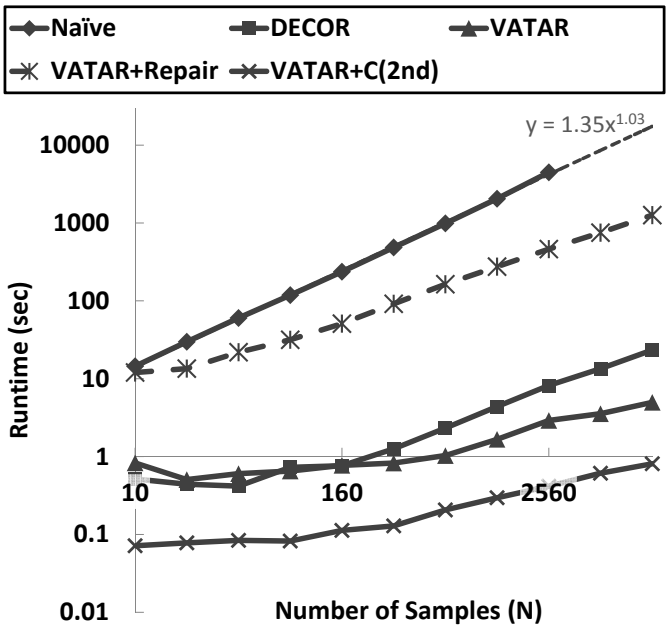


Figure 10.2: PRepDB Runtime Performance for Q_1

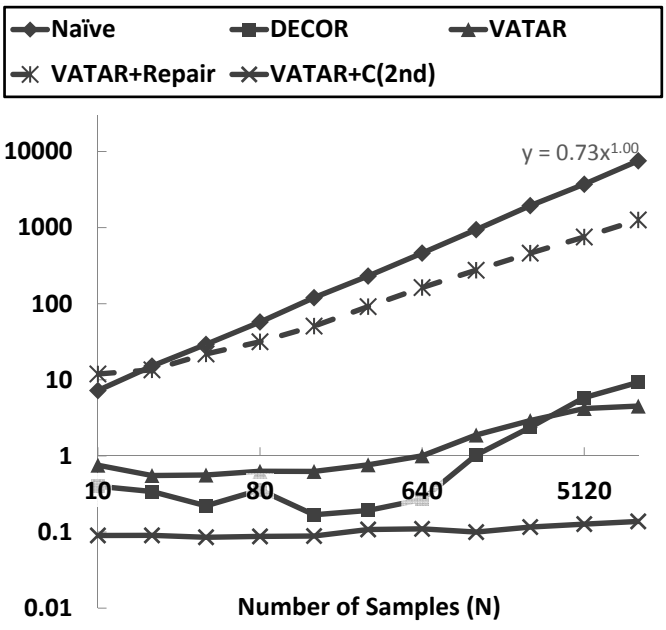


Figure 10.3: PRepDB Runtime Performance for Q_2

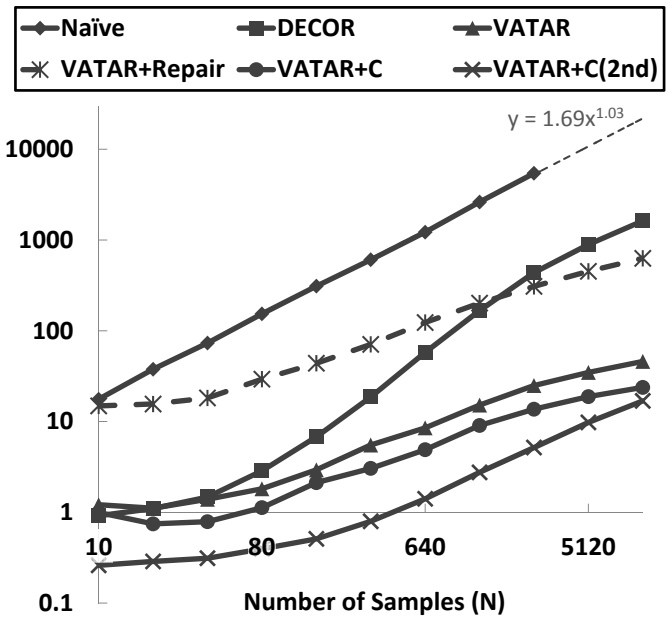


Figure 10.4: PRepDB Runtime Performance for Q_3

10.3 PDB Performance Comparison

In this section, we compare the performance of PRepDB to state of the art in Probabilistic Databases (PDBs) when used in context of FD repairs. For this purpose, we choose two systems to implement, MayBMS [2] and MCDB [18], both of which were discussed in Section 3.1. In order to make a fair comparison to our PRepDB prototype, both of the competing systems are implemented using Python, and PostgreSQL is used for the relational store component. For PRepDB, the `VATAR` configuration described in Section 10.2 is used, with no caching enabled.

We use the same UIS 5000 tuple dataset described in Section 10.2, with a sample of 1024 repairs. The runtime performance is measured of each system for a self-join query with an increasing number of joins. The UIS table is self-joined on the tuple ID TID, which is a certain attribute with an unique value for each row. We choose not to use a non-unique attribute instead, as it would cause an exponential growth in output size, overshadowing the performance difference between the three systems.

Figure 10.5 shows a log-plot of runtime performance for the three systems. MCDB starts off with a high initial overhead, but scales linearly with the number of relations. As expected, the overhead comes from the the need to generate Monte-Carlo samples, each with a large number of certain attributes. This redundancy causes MCDB to be over two orders of magnitude slower

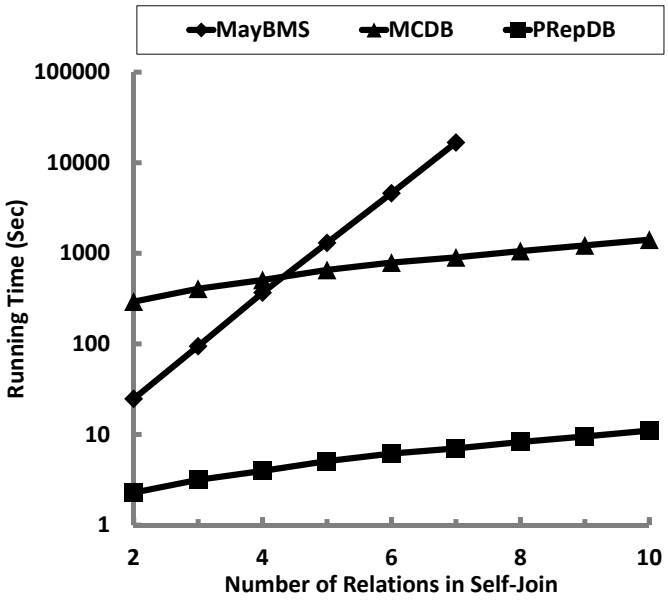


Figure 10.5: Runtime Performance Comparison

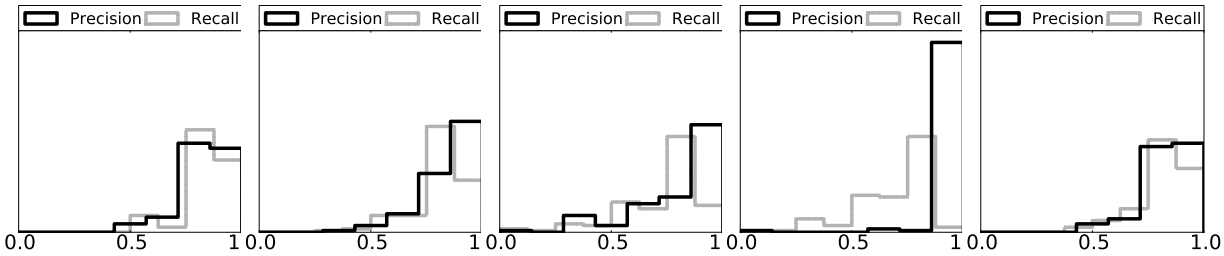
than PRepDB for this query.

On the other hand, MayBMS does not have this redundancy, as it stores and processes each certain cell only once. However, its probability computation requires taking the cartesian product of all relations involved, which results in exponential complexity. At two relations, MayBMS is about an order of magnitude slower than PRepDB, as its possible world size is already twice as large. When the number of relations grows to seven, MayBMS is almost four orders of magnitude slower than PRepDB.

10.4 Quality Evaluation

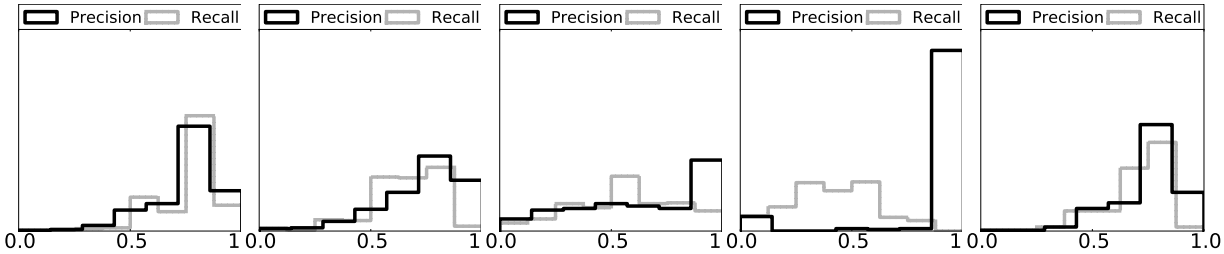
For quality evaluation, we generated a 10000 tuples UIS dataset and applied perturbation at 2%, 6% and 10% rates. We use the following query for each possible city in the database: `SELECT FirstName, LastName FROM UIS WHERE CITY = ?`. We compare the query result of each approach to the query result over the clean relation and analyze the distribution of precision and recall. When measuring precision and recall for the probabilistic results returned by PRepDB, each true positive and false negative is weighted according to its assigned probability.

We use two state of the art single repair algorithms and simulate consistent query answering



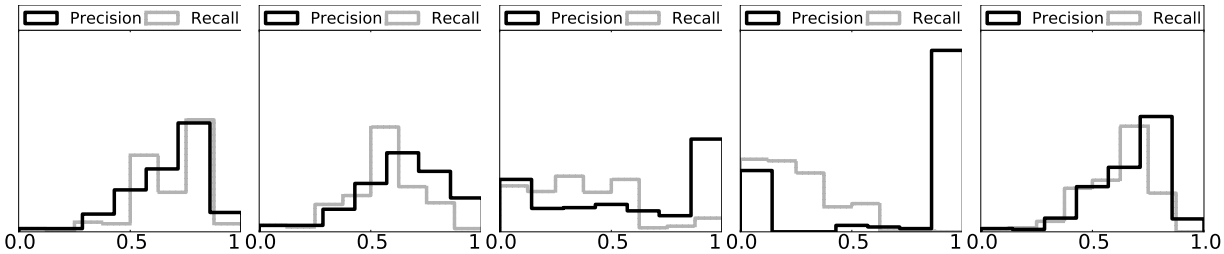
(a) Dirty (b) Vertex Cover (c) Greedy (d) CQA (e) PRepDB

Figure 10.6: Distribution of Quality Results for 2% Perturbation Rate



(a) Dirty (b) Vertex Cover (c) Greedy (d) CQA (e) PRepDB

Figure 10.7: Distribution of Quality Results for 6% Perturbation Rate



(a) Dirty (b) Vertex Cover (c) Greedy (d) CQA (e) PRepDB

Figure 10.8: Distribution of Quality Results for 10% Perturbation Rate

(CQA) by considering only those results obtained from PRepDB that have probability 1.0, i.e. which occur in all the sampled repairs. The following approaches were used in our evaluation:

- `Dirty`: Query result over the dirty relation.
- `PRepDB`: Query result over a sample of 80 CSM repairs.

- `CQA`: Simulation of CQA, that uses results from PRepDB with probability 1.0.
- `Vertex Cover`: This approach models the FD violations as hyper-edges, such that an approximate vertex cover produces a repair with a small number of changes [19].
- `Greedy`: This is a cost-based approach that makes a greedy decision for each subsequent value modification [13].

Figures 10.6, 10.7 and 10.8 are histogram plots that show the number of queries (one for each city) on the y-axis.

When the perturbation rate is low, all approaches do well, as illustrated in Figure 10.6. CQA is able to isolate the clean data and achieve higher precision than other methods. On the other hand, PrepDB provides more consistent recall than CQA and other methods, as it is able to explore a number of possibilities for repairs.

At a higher 6% perturbation rate, the quality distribution shifts to the left for most approaches, as shown in Figure 10.7. However, at this level, the precision results for PrepDB are comparable to other methods, while recall is noticeably better than most other methods.

At 10% perturbation rate the gap widens even more, as shown in Figure 10.8. The precision results get very inconsistent for single-repair methods, as they are prone to make bad mistakes. PrepDB is able to maintain consistently high precision rates, with results often higher than queries over the dirty database. The main drawback of competing approaches becomes apparent with recall results at this perturbation rate. In contrast, PrepDB is noticeably better at providing the user with enough repair possibilities to cover a high proportion of clean answers.

10.5 Quality Comparison of Query Semantics

In this section, we compare the quality results between the explicit query evaluation semantics of MayBMS and the global samples semantics used in PRepDB. We use the same 10000 tuple dataset described in Section 10.4, with 6% perturbation rate. Precision and recall is evaluated using the equijoin query (Q_3) described in Section 10.2.

Figure 10.9 shows the quality results for the two systems. As the number of samples increases, the recall also increases, logarithmically. The additional repair samples increase the probability that correct answers are included in the query result. The explicit query semantics of MayBMS give a slight advantage in the recall result for low number of samples, but this advantage decreases with more repair samples, as the sampling error plays a less significant role in the quality result.

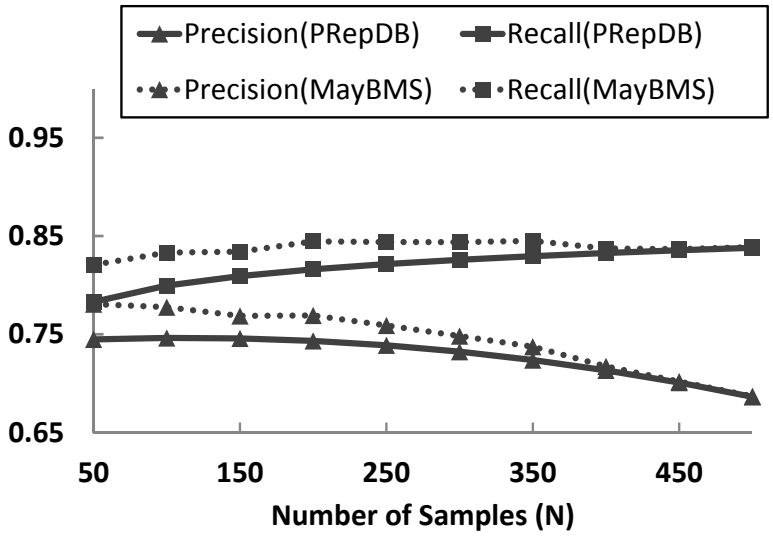


Figure 10.9: Quality Comparison of Query Semantics

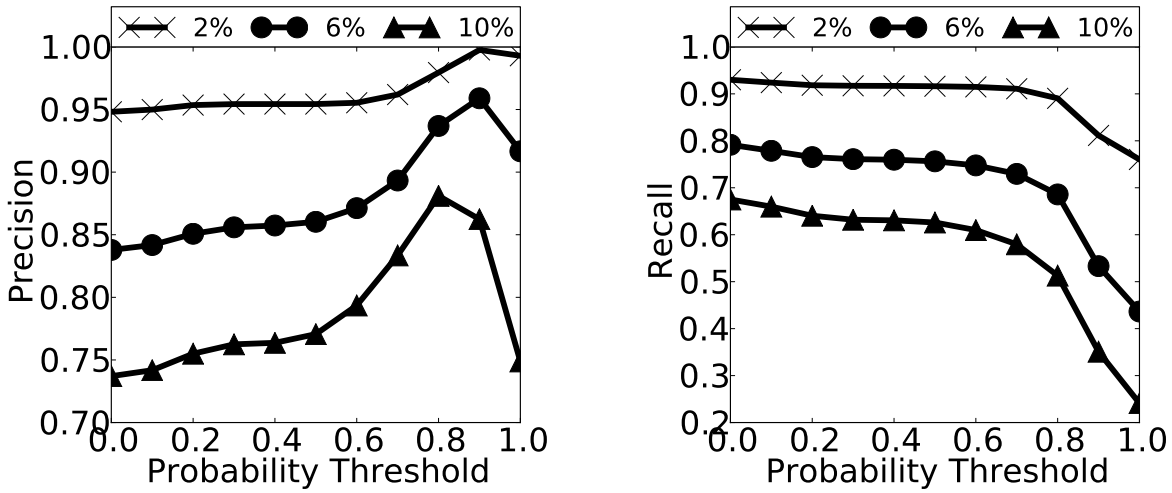
On the other hand, precision slowly decreases with more repair samples, as the additional samples provide more possibilities for each truly correct answer. Similarly to the recall results, MayBMS has a slight advantage for small sample sizes, which decreases as the sample size grows.

In both cases, the difference in quality results is negligible (less that 5%) and the difference effectively disappears with larger sample sizes.

10.6 Result Probability Threshold

In some situations PRepDB may return a very large number of results, many of which may have very low probability. In practice, the user could choose to only consider the results with probability above a set threshold τ. For example, setting the $\tau = 1.0$ will imitate CQA, while at $\tau = 0.0$ all results are considered.

In general, the user can expect τ to control the tradeoff between precision and recall. If obtaining all the potentially clean answers is of higher priority than the false positive rate, then τ should be set relatively low. On the other hand, if the goal is to obtain a small result set with the least number of false positives, then τ should be set higher. Although we operate under the assumption that nothing is known about the ground truth, in cases where a training set is



(a) Precision

(b) Recall

Figure 10.10: Threshold Parameter Quality Results

available, the user is encouraged to experiment with different values for τ in order to find the best parameter value for the desired tradeoff.

Depending on the chosen value for τ, PRepDB may still return results with varying probability values. In this case, the user may choose to rank the results according to their probability and, in critical cases, manually choose results that make the most sense. Another possibility is to feed the results into an analytical business procedure for which the probabilities may provide useful statistical information.

We evaluate the quality results as we vary this threshold parameter. We run the same query from Section 10.4 for each city, for perturbations 2%,6%,10% and average the precision and recall results across all queries. As expected, the recall drops with higher threshold values and approaches CQA level at 1.0, shown in Figure 10.10b. This is because more clean results get discarded. On the other hand, as shown in Figure 10.10a, the precision rises with probability threshold. This is because as more results are discarded, wrong results have a higher chance to get discarded as well. However, after probability threshold of 0.8, the precision drops steeply, as the total number of results that are considered shrinks substantially and so any errors are amplified in the measurement.

Chapter 11

Conclusion and Future Work

In this thesis, we proposed a framework for probabilistic queries over a set of possible repairs to a dirty database. We described a realization of this framework as a system, with methods of representing and storing large samples of FD repairs, as well as efficient query processing that produces probabilistic results over the materialized repair samples. We showed empirically how this approach can produce better quality results than alternative methods for certain queries.

An important direction for future work is to discover and define new spaces of FD repairs that allow direct sampling and thus can be used by our proposed system. Further, it is also important to study more deeply the probabilistic aspects of the FD repair spaces, in order to better understand how the probabilities may relate to the ground truth.

References

[1] Periklis Andritsos, Ariel Fuxman, and Renée J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, page 30, 2006.

[2] Lyublena Antova, Christoph Koch, and Dan Olteanu. $10^{(10^6)}$ worlds and beyond: efficient representation and processing of incomplete information. *VLDB J.*, 18(5):1021–1040, 2009.

[3] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[4] Daniel Barbará, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, 1992.

[5] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[6] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

[7] George Beskales, Ihab F Ilyas, Lukasz Golab, and Artur Galiullin. Sampling from repairs of conditional functional dependency violations. *The VLDB Journal*, pages 1–26, 2013.

[8] George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, and Shai Ben-David. Modeling and querying possible repairs in duplicate detection. *PVLDB*, 2(1):598–609, 2009.

[9] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD Conference*, pages 143–154, 2005.

[10] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. U-dbms: A database system for managing constantly-evolving data. In *VLDB*, pages 1271–1274, 2005.

[11] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2):90–121, 2005.

[12] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Computing consistent query answers using conflict hypergraphs. In *CIKM*, pages 417–426, 2004.

[13] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[14] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

[15] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[16] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, 2007.

[17] Sergio Greco and Cristian Molinaro. Approximate probabilistic query answering over inconsistent databases. In *ER*, pages 311–325, 2008.

[18] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher M. Jermaine, and Peter J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD Conference*, pages 687–700, 2008.

[19] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009.

[20] Andrei Lopatenko and Leopoldo E. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, pages 179–193, 2007.

[21] Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, and Jennifer Widom. Working models for uncertain data. In *ICDE*, page 7, 2006.

[22] Prithviraj Sen and Amol Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, pages 596–605, 2007.

[23] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, pages 375–390, 2003.