Unsupervised Bayesian Data Cleaning Techniques for Structured Data

by

Sushovan De

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved May 2014 by the
Graduate Supervisory Committee:

Dr. Subbarao Kambhampati, Chair
Dr. Yi Chen
Dr. Selçuk Candan
Dr. Huan Liu

ARIZONA STATE UNIVERSITY

August 2014

ABSTRACT

Recent efforts in data cleaning have focused mostly on problems like data deduplication, record matching, and data standardization; few of these focus on fixing incorrect attribute values in tuples. Correcting values in tuples is typically performed by a minimum cost repair of tuples that violate static constraints like CFDs (which have to be provided by domain experts, or learned from a clean sample of the database). In this thesis, I provide a method for correcting individual attribute values in a structured database using a Bayesian generative model and a statistical error model learned from the noisy database directly. I thus avoid the necessity for a domain expert or master data. I also show how to efficiently perform consistent query answering using this model over a dirty database, in case write permissions to the database are unavailable. A Map-Reduce architecture to perform this computation in a distributed manner is also shown. I evaluate these methods over both synthetic and real data.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the thoughtful, patient and knowledgable guidance of my advisor, Dr. Subbarao Kambhampati. To him I am eternally grateful.

I also could not have been here without the love and support of my family: my father, mother and little sister; to whom I've always looked when the going went rough, and for keeping me on the right track.

The bitter Arizona heat would have been unbearable without the constant companionship of my friends; to whom I owe much of what I have accomplished.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Although data cleaning has been a long standing problem, it has become critical again because of the increased interest in web data and big data. The need to efficiently handle structured data that is rife with inconsistency and incompleteness is now more important than ever. Indeed multiple studies (Computing Research Association, 2012) emphasize the importance of effective and efficient methods for handling "dirty data" at scale. Although this problem has received significant attention over the years in the traditional database literature, the state-of-the-art approaches fall far short of an effective solution for big data and web data.

1.1 A Motivating Example

Most of the current techniques are based on deterministic rules, which have a number of problems:

Suppose that the user is interested in finding 'Civic' cars from Table 1.1. Traditional data retrieval systems would return tuples t_1 and t_4 for the query, because they are the only ones that are a match for the query term. Thus, they completely miss the fact that t_4 is in fact a dirty tuple — A Ford Focus car mislabeled as a Civic. Additionally, tuple t_3 and t_5 would not be returned as a result tuples since they have a typos or missing values, although they represent desirable results. My objective is to provide the true result set (t_1, t_3, t_5) to the user. □

Table 1.1: A snapshot of car data extracted from cars.com using information extraction techniques

TID	Model	Make	Orig	Size	Engine	Condition
t_1	Civic	Honda	JPN	Mid-size	V4	NEW
t_2	Focus	Ford	USA	Compact	V4	USED
t_3	Civik	Honda	JPN	Mid-size	V4	USED
t_4	Civic	Ford	USA	Compact	V4	USED
t_5		Honda	JPN	Mid-size	V4	NEW
t_6	Accord	Honda	JPN	Full-size	V6	NEW

1.2 Limitations of Existing Techniques

A variety of data cleaning approaches have been proposed over the years, from traditional methods (e.g., outlier detection (Knorr *et al.*, 2000), noise removal (Xiong *et al.*, 2006), entity resolution (Xiong *et al.*, 2006; Fan *et al.*, 2013), and imputation(Fellegi and Holt, 1976)) to recent efforts on examining integrity constraints, Although these methods are efficient in their own scenarios, their dependence on clean master data is a significant drawback.

Specifically, state of the art approaches (e.g., (Bohannon *et al.*, 2005; Fan *et al.*, 2009; Bertossi *et al.*, 2011) attempt to clean data by exploiting patterns in the data, which they express in the form of conditional functional dependencies (or CFDs). In my motivating example, the fact that Honda cars have 'JPN' as the origin of the manufacturer would be an example of such a pattern. However, these approaches depend on the availability of a clean data corpus or an external reference table to learn data quality rules or patterns before fixing the errors in the dirty data. Systems such as ConQuer (Fuxman *et al.*, 2005) depend upon a set of clean constraints provided by the user. Such clean corpora or constraints may be easy to establish in a tightly controlled enterprise environment but are infeasible for web data and big data. One may attempt to learn data quality rules directly from the noisy data. Unfortunately however, my experimental evaluation shows that even small amounts of noise severely impairs the ability to learn useful constraints from the data.

1.3 BayesWipe Approach

To avoid dependence on clean master data, in this thesis, I propose a novel system called BayesWipe that assumes that a statistical process underlies the generation of clean data (which I call the *data source model*) as well as the corruption of data (which I call the data *error model*). The noisy data itself is used to learn the parameters of these the generative and error models, eliminating dependence on clean master data. Then, by treating the clean value as a latent random variable, BayesWipe leverages these two learned models and automatically infers its value through a Bayesian estimation.

I designed BayesWipe so that it can be used in two different modes: a traditional *offline cleaning* mode, and a novel *online query processing* mode. The offline cleaning mode of BayesWipe follows the classical data cleaning model, where the entire database is accessible and can be cleaned *in situ*. This mode is particularly useful for cleaning data crawled from the web, or aggregated from various noisy sources. The cleaned data can be stored either in a deterministic database, or in a probabilistic database. If a probabilistic database is chosen as the output mode, BayesWipe stores not only the clean version of the tuple it believes to be most likely correct one, but the entire distribution over possible clean tuples. This mode is most useful for those scenarios where recall is very important for further data processing on the cleaned tuples.

The online query processing mode of BayesWipe (De *et al.*, 2014) is motivated by web data scenarios where it is impractical to create a local copy of the data and clean it offline, either due to large size, high frequency of change, or access restrictions. In such cases, the best way to obtain clean answers is to clean the resultset as I retrieve it, which also provides me the opportunity of improving the efficiency of the system, since I can now ignore entire portions of the database which are likely to be unclean or irrelevant to the top-k. BayesWipe uses a query rewriting system that enables it to efficiently retrieve only those tuples that are important to the top-k result set. This rewriting approach is inspired by, and is a significant extension of our earlier work on QPIAD system for handling data incompleteness (Wolf *et al.*, 2009a).

1.4 Probabilistic Database Dependencies

One of the features of the offline mode of BayesWipeis that a probabilistic database (PDB) can be generated as a result of the data cleaning. Probabilistic databases are complex and unintuitive, because each single input tuple is mapped into a distribution over resulting clean alternatives. This is further exacerbated by the fact that one of the key components generating this PDB was a Bayes network. While the structure of a Bayes network can be visualized easily, the parameters are a set of numbers that is hard to get an intuitive grasp on.

In order to understand the underlying model of the data, and to discover relationships between attributes, I show novel algorithms to mine variations of functional dependencies (approximate, conditional and regular) over PDBs. This helps both experts who wish to verify that the data cleaning system is working based on sound reasoning, and naïve users looking for an explanation for a surprising output.

In order to find dependencies among attributes, I first provide the definitions of the equivalent of functional dependencies on PDB (which I call pFD, pAFD, CpFD and CpAFD, for the regular, approximate, conditional and conditional approximate variations). Then I show both an exact algorithm as well as a fast approximate algorithm to find the confidence of a dependency. Using this, I show how to mine these dependencies efficiently.

There are two ways in which probabilistic dependency mining ties into data cleaning and BayesWipe.

Cleaning of PDBs: In the first instance, notice that BayesWipe was built for deterministic databases. It can operate on a deterministic database and produce a probabilistic cleaned database as an output, but it cannot clean a database that is probabilistic to begin with. Indeed, cleaning of probabilistic databases is a largely unexplored area. Using pFDs, I can begin to create algorithms that clean probabilistic data. Indeed, just like CFDs can be used to perform data cleaning of deterministic data, CpFDs can be used to perform data cleaning of certain kinds of probabilistic data.

In order to clean this data, I need a set of dependencies. As I shall show for both CFDs in deterministic data (Chapter 7.2) and probabilistic data (Chapter 10.6), conditional dependencies (that are not approximate) cannot directly be learned from the dirty data, since even a small amount of noise makes their confidence zero. I can ask a domain expert to provide a set of dependencies, or alternatively, I can mine high confidence CpAFDs from the data to perform the cleaning.

PDBs come in various forms; the most general being a collection of possible worlds, and the most simple being tuple-independent databases. This form of data cleaning is able to handle both block-disjoint independent and tuple-independent databases, since in both of these cases, the database can be broken down into a set values that are mutually independent. Existing min-cost repair algorithms can then be modified to perform the smallest change to the database that makes it consistent with the learned dependencies.

Supporting and explaining BayesWipe output: Secondly, probabilistic dependencies can also be used to further analyze a PDB generated as a result of running BayesWipe on a dataset. While BayesWipe uses Bayesian methods to perform the cleaning, the choice of the probabilities of the generated PDBs is made as a product of the generative model and error model's probabilities. In order to gain insight into why a certain option of a certain tuple is given a higher probability, one may find the set of pAFDs from the PDB and use the learned dependencies as an explanation. Additionally, the set of dependencies learned from the cleaned data can be matched against highly correlated attributes in the Bayes network in order to ensure the data cleaning was performed correctly. For example, had the overcorrection parameter been set incorrectly, I will find a discrepancy between Bayes Network and the learned pAFDs.

1.5 Organization

The rest of the thesis is organized as follows. I describe related work in the next chapter, followed by an overview of the architecture in Chapter 3. Chapter 4 describes the learning phase of BayesWipe, where I find the data and error models. Chapter 5 describes the offline cleaning mode, and the next chapter details the query rewriting and online data processing. I describe the results of my empirical evaluation in Chapter 7. The next two chapters describe the BayesWipe Application and the optimizations to run on Map-Reduce framework. In Chapter 10, I describe how to find functional dependencies in probabilistic databases.

Chapter 2

RELATED WORK

2.1 Data Cleaning

The current state of the art in data cleaning focuses on deterministic dependency relations such as FD, CFD, and INDs.

CFDs: Bohannon *et al.* proposed (Bohannon *et al.*, 2007; Fan *et al.*, 2008) using Conditional Functional Dependencies (CFD) to clean data. Indeed, CFDs are very effective in cleaning data. However, the precision and recall of cleaning data with CFDs completely depends on the quality of the set of dependencies used for the cleaning. As my experiments show, learning CFDs from dirty data produces very unsatisfactory results. In order for CFD-based methods to perform well, they need to be learned from a clean sample of the database (Fan *et al.*, 2009). Learning CFDs are more difficult than learning plain FDs. For FDs, the search space is of the order of the number of all possible combinations of the attributes. In the case of CFDs, each such dependency is further adorned with a pattern tableau, that determines specific patterns in tuples to which the dependency applies. Thus, the search space for mining CFDs is extended by all combinations of all possible *values* that can appear in the attributes. Not only does this make learning CFDs from dirty data more infeasible — this also shows that the clean sample of the database from which CFDs are learn must be large enough to be representative of all the patterns in the data. Finding such a large corpus of clean master data is a non-trivial problem, and is infeasible in all but the most controlled of environments (like a corporation with high quality data).

Indeed, all these deterministic dependency based solutions were focused towards the business data problem, where it is well known that the error rate lies between 1% – 5% (Redman, 1998). BayesWipe can handle much higher rates of error, which makes this technique applicable for web data and user-generated data scenarios, which are much more relevant today. This is because BayesWipe learns both the generative and error model from the dirty data itself using Bayes networks and not deterministic rules; the system is a lot more forgiving of dirtiness in the training sample.

Even if a curated set of integrity constraints are provided, existing methods do not use a probabilistically principled method of choosing a candidate correction. They resort to either heuristic based methods, finding an approximate algorithm for the least-cost repair of the database (Arenas *et al.*, 1999; Bohannon *et al.*, 2005; Cong *et al.*, 2007); using a human-guided repair (Yakout *et al.*, 2011), or sampling from a space of possible repairs (Beskales *et al.*, 2013b). There has been work that attempts to guarantee a correct repair of the database (Fan *et al.*, 2010), but they can only provide guarantees for corrections of those tuples that are supported by data from a perfectly clean master database. Recently, Beskales *et al.* (2013a) have shown how the relative trust one places on the constraints and the data itself plays into the choice of cleaning tuples. A Bayesian source model of data was used by Dong *et al.* (2009), but was limited in scope to figuring out the evolution over time of the data value.

Most of the existing work has been focused on deterministic rules, and as a result, the repairs to the database they perform do not have any probabilistic semantics. On the other hand, BayesWipe provides confidence numbers to each of the repairs it performs, which is the posterior probability (in a Bayesian sense) of the corrected tuple given the input database and error models. Recent work (Beskales, 2012) shows the use of a principled probabilistic method for two scenarios: (1) using a probabilistic database to perform data de-duplication, and (2) to fix violations of functional dependencies. Similar to (1), I allow the use of a probabilistic database, however, I use it to store the outcome of my cleaning of corrupted data. As for (2), it has been shown that CFDs are far more effective in data cleaning that FDs (Bohannon *et al.*, 2007), and I show in this thesis that my approach is superior to CFDs as well.

Kubica and Moore also use a probabilistic model that attempts to learn the generative and error model (Kubica and Moore, 2003), and apply it to a image processing domain. However, this thesis separates the noise model into two parts, the noise itself, and the corruption given the noise. Additionally, Kubica and Moore do not specify how the generative and error models were learned.

Recent work has also focused on the metrics to use to evaluate data cleaning techniques (Dasu and Loh, 2012). In this thesis, I focus on evaluating my method against ground truth (when the ground truth is known), and user studies (when the ground truth is not known).

2.2 Query Rewriting

The classic problem of query rewriting (Papakonstantinou and Vassalos, 1999) is to take a SQL query Q that was written against the full database D, and reformulate it to work on a set of views $\mathcal{V}$ so that it produces the same output. I use a similar approach in this thesis — when it is not possible to clean the entire database in place, I use query rewriting to efficiently obtain those tuples that are most likely to be relevant to the user by exploiting all the views that the database *does* expose.

The query rewriting part of this thesis is inspired by the QPIAD system (Wolf *et al.*, 2009a), but significantly improves upon it. QPIAD performed query rewriting over incomplete databases using approximate functional dependencies (AFD). Unlike QPIAD, BayesWipe supports cleaning databases that have both null values as well as *wrong* values. The problem I are attempting to solve in this thesis would not be solvable by QPIAD, since it needs to know the exact attribute that is dirty (QPIAD assumed any non-null value in a tuple was correct). The inference problem I solve is much harder, since I have to infer both the location as well as the value of the error in the tuples.

Researchers have also suggested query rewriting techniques to get clean answers over inconsistent databases. However, there are significant differences between the problem that I solve and the problem typically solved by query rewriting techniques. Arenas *et al.* show (Arenas *et al.*, 1999) a method to generate rewritten queries to obtain clean tuples from an inconsistent database. However, the query rewriting algorithm in that paper is driven by the deterministic integrity dependencies, and not the generative or error model. Since this system requires a set of curated deterministic dependencies, it is not directly applicable to the problem solved in this thesis. Further, due to the use of Bayes networks to model the generative model, BayesWipe is able to incorporate richer types of dependencies.

Recently, performing cleaning of the top-k results of a query has gained interest. Mo *et al.* propose a system (Mo *et al.*, 2013) that cleans just the top-k returned tuples — similar to what I do in this thesis. However, their definition of cleaning the data is very different from ours; while I algorithmically find the best correction for a given tuple, they query the real world for a cleaner sample of any tuple that the system flags as ambiguous.

2.3 Probabilistic Dependencies

Monte Carlo methods have been used in probabilistic databases before, for example, (Dalvi and Suciu, 2007b) uses Monte Carlo methods to give top-k results for queries on probabilistic databases. A more general framework for probabilistic databases, MCDB, is proposed by Jampani et. al. in (Jampani *et al.*, 2008) where the uncertainty is represented by parameters instead of probabilities, so that a more generalized model of uncertainty can be represented. In MCDB, the authors consider the creation and querying of uncertain databases in great detail, and provide algorithms that improve on the time taken to perform the Monte Carlo simulations by considering groups of tuples at a time, which they call tuple bundles. However, neither the problem of finding the confidence of dependencies, nor the problem of mining dependencies is considered by the authors. Sarma et al. extended FDs to probabilistic data in (Sarma *et al.*, 2009), how- ever, in that paper the dependencies that were proposed were appropriate for schema normalization, but were inappropriate for discovering hidden relationships in data. Specifically, the horizontal dependencies specified can detect databases where the FD holds either in the union of all probabilistic tuples, each tuple individually, or within a specific tuple. The first two of these cases are intolerant to any noise in the data. The last one needs a single tuple to be specified, which is not holistic enough to discover any patterns. Such dependencies are ideal for schema normalization, since they allow the tables to be decomposed and simpler schema to be built, but it is not appropriate for discovering data patterns which needs to be fault tolerant.

There is a large body of research that talks about association rules (Agrawal and Srikant, 1994) and itemsets (Brin *et al.*, 1997), more commonly known as the market-basket analysis problem. This work on association rules was recently improved by Kalavagattu (Kalavagattu, 2008) to include pruning based on specificity and to roll them up into approximate functional dependencies. AFDs have also been used to mine attribute correlations on autonomous web databases by Wolf *et al.* (Wolf *et al.*, 2009b). They have also been used by Wang *et al.* (Wang *et al.*, 2009) to find dirty data sources and normalize large mediated schemas. FDs have also been generalized into conditional functional dependencies. Their role in data cleaning was shown by

Bohannon *et al.* (Bohannon *et al.*, 2007).

In (Gupta and Sarawagi, 2006), Gupta and Sarawagi demonstrate how to create probabilistic databases that are an approximation of an information extraction model and find that using the appropriate model of uncertainty in a database is important. If the model of uncertainty is too simple then interactions between elements of the generating model cannot be represented; if it is too complex then querying becomes inefficient. Similarly, in this chapter, I am proposing the right level of uncertainty, but for functional dependencies. I show that using probabilistic semantics does cause a significant change in the confidence of the dependencies, and I show efficient algorithms that find these dependencies.

Chapter 3

BAYESWIPE OVERVIEW

BayesWipe views the data cleaning problem as a statistical inference problem over the structured data. Let $\mathcal{D} = \{T_1, ..., T_n\}$ be the input structured data which contains a certain number of corruptions. $T_i \in \mathcal{D}$ is a tuple with m attributes $\{A_1, ..., A_m\}$ which may have one or more corruptions in its attribute values. So given a correction candidate set $\mathcal{C}$ for a possibly corrupted tuple T in $\mathcal{D}$, I can clean the database by replacing T with the candidate clean tuple $T^* \in \mathcal{C}$ that has the maximum $P(T^*|T)$. Using Bayes rule (and dropping the common denominator), I can rewrite this to

$$T^*_{best} = \arg\max[P(T|T^*)P(T^*)] \tag{3.1}$$

If I wish to create a probabilistic database (PDB), I don't take an $\arg\max$ over the $P(T^*|T)$, instead I store the entire distribution over the T^* in the resulting PDB.

For online query processing I take the user query Q^*, and find the relevance score of a tuple T as

$$Score(T) = \sum_{T^* \in \mathcal{C}} \underbrace{P(T^*)}_{\text{source model}} \underbrace{P(T|T^*)}_{\text{error model}} \underbrace{R(T^*|Q^*)}_{\text{relevance}} \tag{3.2}$$

In this thesis, I used a binary relevance model, where R is 1 if T^* is relevant to the user's query, and 0 otherwise. Note that R is the relevance of the query Q^* to the candidate clean tuple T^* and not the observed tuple T. This allows the query rewriting phase of BayesWipe, which aims to retrieve tuples with highest $Score(.)$ to achieve the non-lossy effect of using a PDB without explicitly rectifying the entire database.

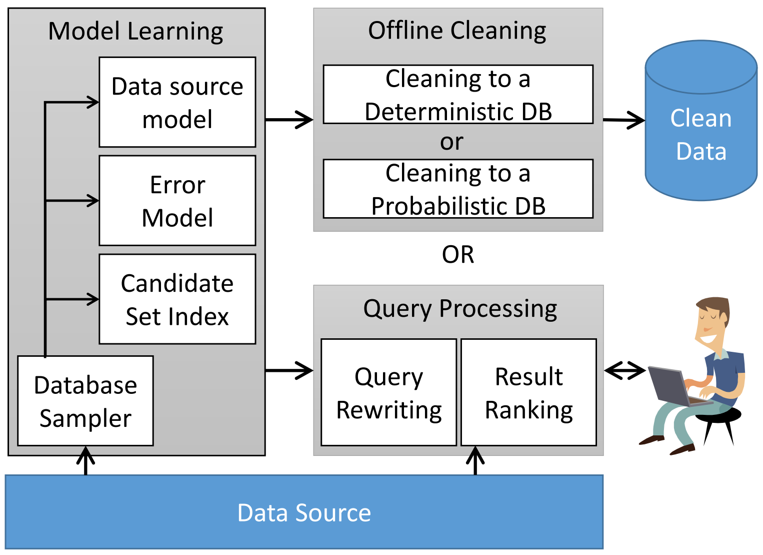


Figure 3.1: The architecture of BayesWipe. My framework learns both data source model and error model from the raw data during the model learning phase. It can perform offline cleaning or query processing to provide clean data.

3.1 Architecture

Figure 3.1 shows the system architecture for BayesWipe. During the model learning phase (Section 4), I first obtain a sample database by sending some queries to the database. On this sample data, I learn the generative model of the data as a Bayes network (Section 4.1). In parallel, I define and learn an error model which incorporates three types of errors (Section 4.2). I also create an index to quickly propose candidate T^*s.

I can then choose to do either offline cleaning (Section 5) or online query processing (Section 6), as per the scenario. In the offline cleaning mode, I can choose whether to store the resulting cleaned tuple in a deterministic database (where I store only the T^* with the maximum posterior probability) or probabilistic database (where I store the entire distribution over the T^*). In the online query processing mode, I obtain a query from the user, and do query rewriting in order to find a set of queries that are likely to retrieve a set of highly relevant tuples. I execute these queries and re-rank the results, and then display them to the user.

Algorithm 1: The algorithm for offline data cleaning

```
Input: $D$, the dirty dataset.
$BN \leftarrow$ Learn Bayes Network ($D$)
foreach Tuple $T \in D$ do
    $\mathcal{C} \leftarrow$ Find Candidate Replacements ($T$)
    foreach Candidate $T^* \in \mathcal{C}$ do
        $P(T^*) \leftarrow$ Find Joint Probability ($T^*, BN$)
        $P(T|T^*) \leftarrow$ Error Model ($T, T^*$)
    end
    $T \leftarrow \arg\max_{T^* \in \mathcal{C}} P(T^*)P(T|T^*)$
end
```

In Algorithms 1 and 2, we present the overall algorithm for BayesWipe. In the offline mode, we show how we iterate over all the tuples in the dirty database, D and replace them with cleaned tuples. In the query processing mode, the first three operations are performed offline, and the remaining operations show how the tuples are efficiently retrieved from the database, ranked and displayed to the user.

3.2 BayesWipe Application, Map-Reduce and Probabilistic Dependencies extensions

I made publicly available a version of BayesWipe that users can download and clean any dataset of their choice. The architecture for this application matches the architecture described in the previous section.

Making a complex algorithm like BayesWipe truly parallel is not simple: the pieces

Algorithm 2: Algorithm for online query processing.

```
Input: $D$, the dirty dataset
Input: $Q$, the user's query
$S \leftarrow$ Sample the source dataset $D$
$BN \leftarrow$ Learn Bayes Network $(S)$
$ES \leftarrow$ Learn Error Statistics $(S)$
$R \leftarrow$ Query and score results $(Q, D, BN)$
$ESQ \leftarrow$ Get expanded queries $(Q)$
foreach Expanded query $E \in ESQ$ do
    $R \leftarrow R \cup$ Query and score results $(E, D, BN)$
    $RQ \leftarrow RQ \cup$ Get all relaxed queries $(E)$
end
$Sort(RQ)$ by expected relevance, using $ES$
while top-k confidence not attained do
    $B \leftarrow$ Pick and remove top $RQ$
    $R \leftarrow R \cup$ Query and score results $(B, D, BN)$
end
$Sort(R)$ by score
return $R$
```

are coupled because the index needs to be generated, and the size of the index quickly gets out of hand. I show a way to decouple the index, by dividing it based on the hash of the common attribute. This lets me route the input to the correct node so that the index on the node is minimized. The architecture for this decoupling is explained more fully in Chapter 9. In short, there is a two-stage map-reduce architecture, where in the first stage, the tuples are routed to a set of reducer nodes which hold the relevant candidate tuples for them. In the second stage, the resulting candidate tuples along with their scores are collated, and the best one is selected from them.

In order to better express the dependencies among attributes to humans, I extend the definition of functional dependencies to incorporate cases for PDBs. There are two existing dimensions along which functional dependencies have been generalized: approximate and conditional. I introduce each of these to PDBs, creating four new types of dependencies: pFD, pAFD, CpFD, CpAFD. In order to mine these dependencies, I take a two-stage approach. First, I present an algorithm to determine the confidence of a dependency. For pFD, I show an exact algorithm that efficiently trims down the number of computation it has to perform. I also show a Monte-Carlo based approximation method to efficiently estimate the confidence of all these dependencies.

Finally, similar to the algorithm for mining AFDs from data, I show how to reduce the search space when trying to mine these dependencies directly from the probabilistic data.

Chapter 4

MODEL LEARNING

This chapter details the process by which I estimate the components of Equation 3.2: the data source model $P(T^*)$ and the error model $P(T|T^*)$

4.1 Data Source Model

The data that I work with can have dependencies among various attributes (e.g., a car's *engine* depends on its *make*). Therefore, I represent the data source model as a Bayes network, since it naturally captures relationships between the attributes via structure learning and infers probability distributions over values of the input tuple instances.

Constructing a Bayes network over $\mathcal{D}$ requires two steps: first, the induction of the graph structure of the network, which encodes the conditional independences between the m attributes of $\mathcal{D}$'s schema; and second, the estimation of the parameters of the resulting network. The resulting model allows me to compute probability distributions over an arbitrary input tuple T.

I observe that the structure of a Bayes network of a given dataset remains constant with small perturbations, but the parameters (CPTs) change more frequently. As a result, I spend a larger amount of time learning the structure of the network with a slower, but more accurate tool, Banjo (Hartemink., ????). Figures 4.1 and 4.2 show automatically learned structures for two data domains.

Then, given a learned graphical structure $\mathcal{G}$ of $\mathcal{D}$, I can estimate the conditional probability tables (CPTs) that parameterize each node in $\mathcal{G}$ using a faster package called Infer.NET (Minka *et al.*, 2010). This process of inferring the parameters is run

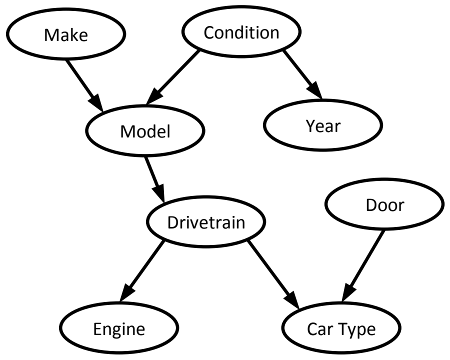


Figure 4.1: Learned Bayes Network: Auto dataset

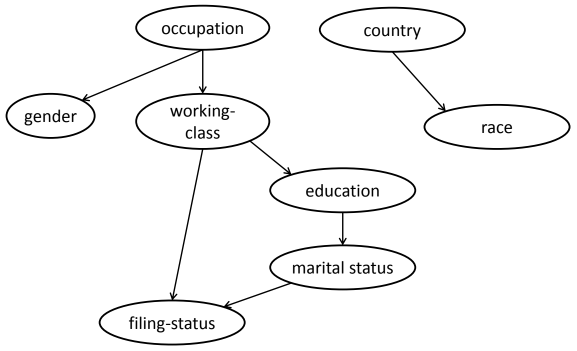


Figure 4.2: Learned Bayes Network: Census dataset

offline, but more frequently than the structure learning.

Once the Bayesian network is constructed, I can infer the joint distributions for arbitrary tuple T, which can be decomposed to the multiplication of several marginal distributions of the sets of random variables, conditioned on their parent nodes depending on $\mathcal{G}$.

4.2 Error Model

Having described the data source model, I now turn to the estimation of the error model $P(T|T^*)$ from noisy data. Given a set of clean candidate tuples $\mathcal{C}$ where $T^* \in \mathcal{C}$, my error model $P(T|T^*)$ essentially measures how clean T is, or in other words, how similar T is to T^*. Unlike traditional record linkage measures (Koudas *et al.*, 2006), my similarity functions have to take into account dependencies among attributes.

I now build an error model that can estimate some of the most common kinds of errors in real data: a combination of spelling, incompletion and substitution errors.

4.2.1 Edit Distance Similarity:

This similarity measure is used to detect spelling errors. Edit distance between two strings T_{A_i} and $T^*_{A_i}$ is defined as the minimum cost of edit operations applied to dirty tuple T_{A_i} transform it to clean $T^*_{A_i}$. Edit operations include character-level copy, insert, delete and substitute. The cost for each operation can be modified as required; in this thesis I use the Levenshtein distance, which uses a uniform cost function. This gives me a distance, which I then convert to a probability using (Ristad and Yianilos, 1998):

$$f_{ed}(T_{A_i}, T^*_{A_i}) = \exp\{-cost_{ed}(T_{A_i}, T^*_{A_i})\} \tag{4.1}$$

4.2.2 Distributional Similarity Feature:

This similarity measure is used to detect both substitution and omission errors. Looking at each attribute in isolation is not enough to fix these errors. I propose a context-based similarity measure called Distributional similarity (f_{ds}), which is based on the probability of replacing one value with another under a similar context (Li *et al.*, 2006). Formally, for each string T_{A_i} and $T^*_{A_i}$, I have:

$$f_{ds}(T_{A_i}, T^*_{A_i}) = \sum_{c \in C(T_{A_i}, T^*_{A_i})} \frac{\mathbf{Pr}(c|T^*_{A_i})\mathbf{Pr}(c|T_{A_i})\mathbf{Pr}(T_{A_i})}{\mathbf{Pr}(c)} \tag{4.2}$$

where $C(T_{A_i}, T^*_{A_i})$ is the context of a tuple attribute value, which is a set of attribute values that co-occur with both T_{A_i} and $T^*_{A_i}$. $\mathbf{Pr}(c|T^*_{A_i}) = (\#(c, T^*_{A_i}) + \mu)/\#(T^*_{A_i})$ is the probability that a context value c appears given the clean attribute $T^*_{A_i}$ in the sample database. Similarly, $P(T_{A_i}) = \#(T_{A_i})/\#tuples$ is the probability that a dirty attribute value appears in the sample database. I calculate $\mathbf{Pr}(c|T_{A_i})$ and $\mathbf{Pr}(T_{A_i})$ in the same way. To avoid zero estimates for attribute values that do not appear in the database sample, I use Laplace smoothing factor μ.

4.2.3 Unified Error Model:

In practice, I do not know beforehand which kind of error has occurred for a particular attribute; I need a unified error model which can accommodate all three types of errors (and be flexible enough to accommodate more errors when necessary). For this purpose, I use the well-known maximum entropy framework (Berger *et al.*, 1996) to leverage all available similarity measures, including Edit distance f_{ed} and distributional similarity f_{ds}. So for the input tuple T and T^*, I have my unified error model defined on this attribute as follows:

$$\mathbf{Pr}(T|T^*) = \frac{1}{Z}\exp\left\{\alpha\sum_{i=1}^{m} f_{ed}(T_{A_i}, T^*_{A_i}) + \beta\sum_{i=1}^{m} f_{ds}(T_{A_i}, T^*_{A_i})\right\} \qquad (4.3)$$

where α and β are the weight of each similarity measure, m is the number of attributes in the tuple. The normalization factor is $Z = \sum_{T^*}\exp\{\sum_i \lambda_i f_i(T^*, T)\}$.

4.3 Finding the Candidate Set

I need to find the set of candidate clean tuples, $\mathcal{C}$, that comprises all the tuples in the sample database that differ from T in not more than j attributes. Even with $j = 3$, the naïve approach of constructing $\mathcal{C}$ from the sample database directly is too time consuming, since it requires one to go through the sample database in its entirety once for every result tuple encountered. To make this process faster, I create indices over $(j+1)$ attributes. If any candidate tuple T^* differs from T in less than or equal to j attributes, then it will be present in at least one of the indices, since

I created $j + 1$ of them. These $j + 1$ indices are created over those attributes that have the highest cardinalities, such as Make and Model (as opposed to attributes like Condition and Doors which can take only a few values). For every possibly dirty tuple T in the database, I go over each such index and find all the tuples that match the corresponding attribute. The union of all these tuples is then examined and the candidate set $\mathcal{C}$ is constructed by keeping only those tuples from this union set that do not differ from T in more than j attributes.

Thus I can be sure that by using this method, I have obtained the entire set $\mathcal{C}$. By using those attributes that have high cardinality, I ensure that the size of the set of tuples returned from the index would be small.

Chapter 5

OFFLINE CLEANING

5.1 Cleaning to a Deterministic Database

In order to clean the data *in situ*, I first use the techniques of the previous section to learn the data generative model, the error model and create the index. Then, I iterate over all the tuples in the database and use Equation 3.1 to find the T^* with the best score. I then replace the tuple with that T^*, thus creating a deterministic database using the offline mode of BayesWipe.

5.2 Cleaning to a Probabilistic Database

I note that many data cleaning approaches — including the one I described in the previous sections — come up with multiple alternatives for the clean version for any given tuple, and evaluate their confidence in each of the alternatives. For example, if a tuple is observed as 'Honda, Corolla', two correct alternatives for that tuple might be 'Honda, Civic' and 'Toyota, Corolla'. In such cases, where the choice of the clean tuple is not an obvious one, picking the most-likely option may lead to the wrong answer. Additionally, if one intends to do further processing on the results, such as perform aggregate queries, join with other tables, or transfer the data to someone else for processing, then storing the most likely outcome is lossy.

A better approach (also suggested by others (Computing Research Association, 2012)) is to store all of the alternative clean tuples along with their confidence values. Doing this, however, means that the resulting database will be a probabilistic database (PDB), even when the source database is deterministic.

It is not clear upfront whether PDB-based cleaning will have advantages over cleaning to a deterministic database. On the positive side, using a PDB helps reduce loss of information arising from discarding all alternatives to tuples that did not have the maximum confidence. On the negative side, PDB-based cleaning increases the query processing cost (as querying PDBs are harder than querying deterministic databases (Dalvi and Suciu, 2004)). Another challenge is that of presentation: users usually assume that they are dealing with a deterministic source of data, and presenting all alternatives to them can be overwhelming to them.

In this section, and in the associated experiments, I investigate the potential advantages to using the BayesWipe system and storing the resulting cleaned data in a probabilistic database. For my experiments, I used Mystiq (Boulos *et al.*, 2005), a prototype probabilistic database system from University of Washington, as the substrate.

In order to create a probabilistic database from the corrections of the input data, I follow the offline cleaning procedure described previously in Section 4. Instead of storing the most likely T^*, I store all the T^*s along with their $P(T^*|T)$ values.

When evaluating the performance of the probabilistic database, I used simple select queries on the resulting database. Since representing the results of a probabilistic database to the user is a complex task, in this thesis I focus on showing just the tuple ID to the user. The rationale for my decision is that in a used car scenario, the user will be satisfied if the system provides a link to the car the user intended to purchase — the exact reasoning the system used to come up with the answer is not relevant to the user. As a result, the form of my output is a tuple-independent database. This can be better explained with an example:

Example: Suppose I clean my running example of Table 1.1. I will obtain

Table 5.1: Cleaned Probabilistic Database

TID	Model	Make	Orig.	Size	Eng.	Cond.	P
t_1	Civic	Honda	JPN	Mid-size	V4	NEW	0.6
	Civic	Honda	JPN	Compact	V6	NEW	0.4
...							
t_3	Civic	Honda	JPN	Mid-size	V4	USED	0.9
	Civik	Honda	JPN	Mid-size	V4	USED	0.1

a tuple-disjoint independent [1] probabilistic database (Suciu and Dalvi, 2005); a fragment of which is shown in Table 5.1. Each original input tuple (t_1, t_3), has been cleaned, and their alternatives are stored along with the computed confidence values for the alternatives (0.6 and 0.4 for t_1, in this example). Suppose the user issues a query $\mathsf{Model} = \mathsf{Civic}$. Both options of tuple t_1 of the probabilistic database satisfy the constraints of the query. Since I are only interested in the tuple ID, I project out every other attribute, resulting in returning tuple t_1 in the result with a probability 0.6 + 0.4 = 1. Only the first option in tuple t_3 matches the query. Thus the result will contain the tuple t_3 with probability 0.9. The experimental results use only the tuple ids when computing the recall of the method. The output probabilistic relation is shown in Table 5.2.

Table 5.2: Result Probabilistic Database

TID	P
t_1	1
t_3	0.9

[1] A tuple-disjoint independent probabilistic database is one where every tuple, identified by its primary key, is independent of all other tuples. Each tuple is, however, allowed to have multiple alternatives with associated probabilities. In a tuple-independent database, each tuple has a single probability, which is the probability of that tuple existing.

The interesting fact here is that the result of any query will always be a tuple-independent database. This is because I projected out every attribute except for the tuple-ID, and the tuple-IDs are independent of each other. □

When showing the results of my experiments, I evaluate the precision and recall of the system. Since precision and recall are deterministic concepts, I have to convert the probabilistic database into a deterministic database (that will be shown to the user) prior to computing these values. I can do this conversion in two ways: (1) by picking only those tuples whose probability is higher than some threshold. I call this method the *threshold based determinization*. (2) by picking the top-k tuples and discarding the probability values (*top-k determinization*). The experiment section (Section 7.2) shows results with both determinizations.

Chapter 6

ONLINE QUERY REWRITING

In this chapter I develop an online query processing method where the result tuples are cleaned at query time. Two challenges need to be addressed to do this effectively. First, certain tuples that do not satisfy the query constraints, but are relevant to the user, need to be retrieved, ranked and shown to the user. Second, the process needs to be efficient, since the time that the users are willing to wait before results are shown to them is very small. I show my query rewriting mechanisms aimed at addressing both these challenges.

I begin by executing the user's query (Q^*) on the database. I store the retrieved results, but do not show them to the user immediately. I then find rewritten queries that are most likely to retrieve clean tuples. I do that in a two-stage process: I first expand the query to increase the precision, and then relax the query by deleting some constraints (to increase the recall).

6.1 Increasing the Precision of Rewritten Queries

Since my data sources are inherently noisy, it is important that I do not retrieve tuples that are obviously incorrect. Doing so will improve not only the quality of the result tuples, but also the efficiency of the system. I can improve precision by adding relevant constraints to the query Q^* given by the user. For example, when a user issues the query Model = Civic, I can expand the query to add relevant constraints Make = Honda, Country = Japan, Size = Mid-Size. These additions capture the essence of the query — because they limit the results to the specific kind of car the user is probably looking for. These expanded structured queries I generate from the user's

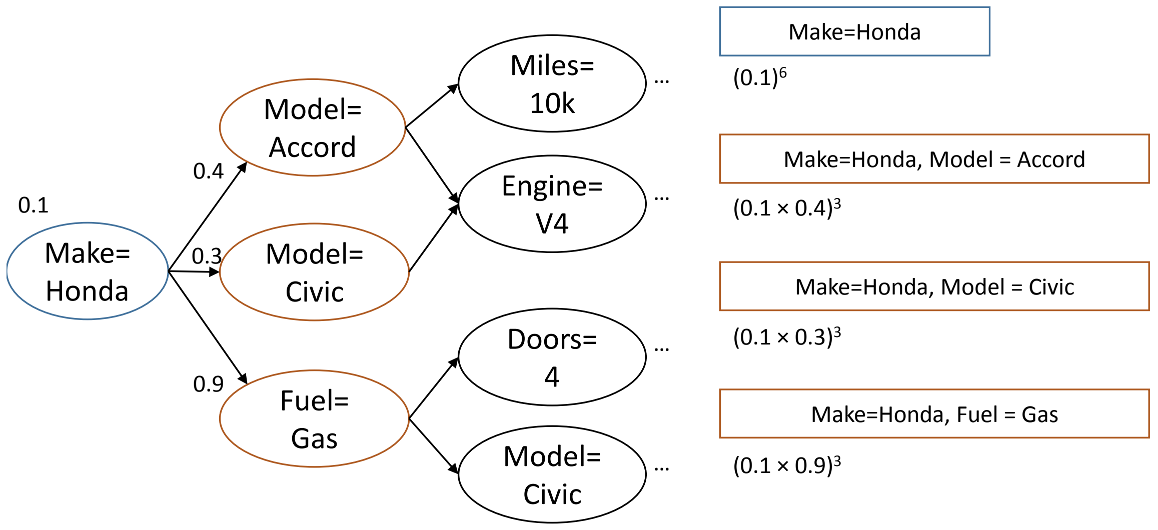


Figure 6.1: Query Expansion Example. The tree shows the candidate constraints that can be added to a query, and the rectangles show the expanded queries with the computed probability values.

query are called ESQs.

Each user query Q^* is a select query with one or more attribute-value pairs as constraints. In order to create an ESQ, I will have to add highly correlated constraints to Q^*.

Searching for correlated constraints to add requires Bayesian inference, which is an expensive operation. Therefore, when searching for constraints to add to Q^*, I restrict the search to the union of all the attributes in the Markov blanket (Pearl, 1988). The Markov blanket of an attribute comprises its children, its parents, and its children's other parents. It is the set of attributes whose value being given, the node becomes independent of all other nodes in the network. Thus, it makes sense to consider these nodes when finding correlated attributes. This correlation is computed using the Bayes Network that was learned offline on a sample database (recall the architecture of BayesWipe in Figure 3.1.) ‘’

Given a Q^*, I attempt to generate multiple ESQs that maximizes both the relevance of the results and the coverage of the queries of the solution space.

Note that if there are m attributes, each of which can take n values, then the total number of possible ESQs is n^m. Searching for the ESQ that globally maximizes the objectives in this space is infeasible; I therefore approximately search for it by performing a heuristic-informed search. My objective is to create an ESQ with m attribute-value pairs as constraints. I begin with the constraints specified by the user query Q^*. I set these as evidence in the Bayes network, and then query the Markov blanket of these attributes for the attribute-value pairs with the highest posterior probability given this evidence. I take the top-k attribute-value pairs and append them to Q^* to produce k search nodes, each search node being a query fragment. If Q has p constraints in it, then the heuristic value of Q is given by $P(Q)^{m/p}$. This represents the expected joint probability of Q when expanded to m attributes, assuming that all the constraints will have the same average posterior probability. I expand them further, until I find k queries of size m with the highest probabilities.

Example: In Figure 6.1, I show an example of the query expansion. The node on the left represents the query given by the user "Make=Honda". First, I look at the Markov Blanket of the attribute Make, and determine that Model and Condition are the nodes in the Markov blanket. I then set "Make=Honda" as evidence in the Bayes network and then run an inference over the values of the attribute Model. The two values of the Model attribute with the highest posterior probability are Accord and Civic. The most probable values of the Condition attribute are "new" and "old". Using each of these values, new queries are constructed and added to the queue. Thus, the queue now consists of the 4 queries: "Make=Honda, Model=Civic", "Make=Honda, Model=Accord" and "Make=Honda, Condition=old". A fragment of these queries are shown in the middle column of Figure 6.1. I dequeue the highest probability item from the queue and repeat the process of setting the evidence, finding the Markov Blanket, and running the inference. I stop when I get the required number of ESQs with a sufficient number of constraints.

6.2 Increasing the Recall

Adding constraints to the query causes the precision of the results to increase, but reduces the recall drastically. Therefore, in this stage, I choose to delete some constraints from the ESQs, thus generating relaxed queries (RQ). Notice that tuples that have corruptions in the attribute constrained by the user (recall tuples t_3 and t_5 from my running example in Table 1.1) can only be retrieved by relaxed queries that do not specify a value for those attributes. Instead, I have to depend on rewritten queries that contain correlated values in other attributes to retrieve these tuples. Using relaxed queries can be seen as a trade-off between the recall of the resultset and the time taken, since there are an exponential number of relaxed queries for any given ESQ. As a result, an important question is the order and number of RQs to execute.

I define the rank of a query as the *expected relevance* of its result set.

$$Rank(q) = \mathbb{E}\left(\frac{\sum_{T_q} Score(T_q|Q^*)}{|T_q|}\right)$$

where T_q are the tuples returned by a query q, and Q^* is the user's query. Executing an RQ with a higher rank will have a more beneficial result on the result set because it will bring in better quality result tuples.

Estimating this quantity is difficult because I do not have complete information about the tuples that will be returned for any query q. The best I can do, therefore, is to approximate this quantity.

Let the relaxed query be Q, and the expanded query that it was relaxed from be

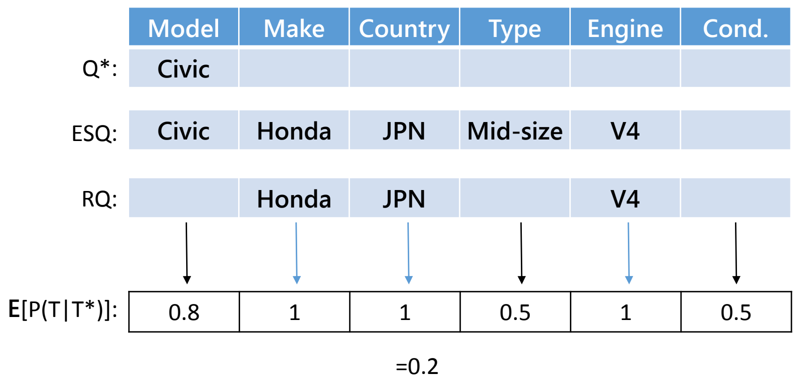


Figure 6.2: Query Relaxation Example.

ESQ. I wish to estimate $\mathbb{E}[P(T|T^*)]$ where T are the tuples returned by Q. Using the attribute-error independence assumption, I can rewrite that as $\prod_{i=0}^{m} P(T.A_i|T^*.A_i)$, where $T.A_i$ is the value of the i-th attribute in T. Since ESQ was obtained by expanding Q^* using the Bayes network, it has values that can be considered clean for this evaluation. Now, I divide the m attributes of the database into 3 classes: (1) The attribute is specified both in ESQ and in Q. In this case, I set $P(T.A_i|T^*.A_i)$ to 1, since $T.A_i = T^*.A_i$. (2) The attribute is specified in ESQ but not in Q. In this case, I know what $T^*.A_i$ is, but not $T.A_i$. However, I can generate an average statistic of how often $T^*.A_i$ is erroneous by looking at my sample database. Therefore, in the offline learning stage, I pre-compute tables of error statistics for every T^* that appears in my sample database, and use that value. (3) The attribute is not specified in either ESQ or Q. In this case, I know neither the attribute value in T nor in T^*. I, therefore, use the average error rate of the entire attribute as the value for $P(T.A_i|T^*.A_i)$. This statistic is also precomputed during the learning phase. This product gives me the expected rank of the tuples returned by Q.

Example: In Figure 6.2, I show an example for finding the probability values of a relaxed query. Assume that the user's query Q^* is "Civic", and the ESQ is shown in the second row. For an RQ that removes the attribute values "Civic" and "Mid-Size" from the ESQ, the probabilities are calculated as follows: For the attributes "Make, Country" and "Engine", the values are present in both the ESQ as well as the RQ, and therefore, the $P(T|T^*)$ for them is 1. For the attribute "Model" and "Type", the values are present in ESQ but not in RQ, hence the value for them can be computed from the learned error statistics. For example, for "Civic", the average value of $P(T|Civic)$ as learned from the sample database (0.8) is used. Finally, for the attribute "Condition", which is present neither in ESQ nor in RQ, I use the average error statistic for that attribute (i.e. the average of $P(T_a|T_a^*)$ for $a =$ "Condition" which is 0.5).

The final value of $\mathbb{E}[P(T|T^*)]$ is found from the product of all these attributes as 0.2. □

Terminating the process: I begin by looking at all the RQs in descending order of their rank. If the current k-th tuple in my resultset has a relevance of λ, and the estimated rank of the Q I am about to execute is $R(T_q|Q)$, then I stop evaluating any more queries if the probability $\mathbf{Pr}(R(T_q|Q) > \lambda)$ is less than some user defined threshold $\mathcal{P}$. This ensures that I have the true top-k resultset with a probability $\mathcal{P}$.

Chapter 7

EXPERIMENTS

I quantitatively study the performance of BayesWipe in both its modes — offline, and online, and compare it against state-of-the-art CFD approaches. I used three real datasets spanning two domains: used car data, and census data. I present experiments on evaluating the approach in terms of the effectiveness of data cleaning, efficiency and precision of query rewriting.

7.1 Experimental Setup

To perform the experiments, I obtained the real data from the web. The first dataset is *Used car sales* dataset D_{car} crawled from Google Base. The second dataset I used was adapted from the *Census Income* dataset D_{census} from the UCI machine learning repository (Asuncion and Newman, 2007). From the fourteen available attributes, I picked the attributes that were categorical in nature, resulting in the following 8 attributes: working-class, education, marital status, occupation, race, gender, filing status. country. The same setup was used for both datasets – including parameter values and error features.

These datasets were observed to be mostly clean. I then introduced [1] three types of noise to the attributes. To add noise to an attribute, I randomly changed it either to a new value which is close in terms of string edit distance (distance between 1 and 4, simulating spelling errors) or to a new value which was from the same attribute (simulating replacement errors) or just deleted it (simulating deletion errors).

[1] I note that the introduction of synthetic errors into clean data for experimental evaluation purposes is common practice in data cleaning research (Cong *et al.*, 2007; Bohannon *et al.*, 2007).

A third dataset was car inventory data crawled from the website 'cars.com'. This dataset was observed to have inaccuracies — therefore, I used this to validate my approach against real-world noise in the data, where I do not control the noise process.

7.2 Experiments

Offline Cleaning Evaluation: The first set of evaluations shows the effectiveness of the offline cleaning mode. In Figure 7.1, I compare BayesWipe against CFDs (Chiang and Miller, 2008). The dotted line that shows the number of CFDs learned from the noisy data quickly falls to zero, which is not surprising: CFDs learning was designed with a clean training dataset in mind. Further, the only constraints learned by this algorithm are the ones that have not been violated in the dataset — unless a tuple violates some CFD, it cannot be cleaned. As a result, the CFD method cleans exactly zero tuples independent of the noise percentage. On the other hand, BayesWipe is able to clean between 20% to 40% of the data. It is interesting to note that the percentage of tuples cleaned increases initially and then slowly decreases. This is because for very low values of noise, there aren't enough errors available for the system to learn a reliable error model from; and at larger values of noise, the data source model learned from the noisy data is of poorer quality.

While Figure 7.1 showed only percentages, in Figure 7.2 I report the actual number of tuples cleaned in the dataset along with the percentage cleaned. This curve shows that the raw number of tuples cleaned always increases with higher input noise percentages.

Setting α and β: The weight given to the distributional similarity (β), and the weight given to the edit distance (α) are parameters that can be tuned, and should be set based on which kind of error is more likely to occur. In my experiments, I performed a grid search to determine the best values of α and β to use. In Figure 7.3,

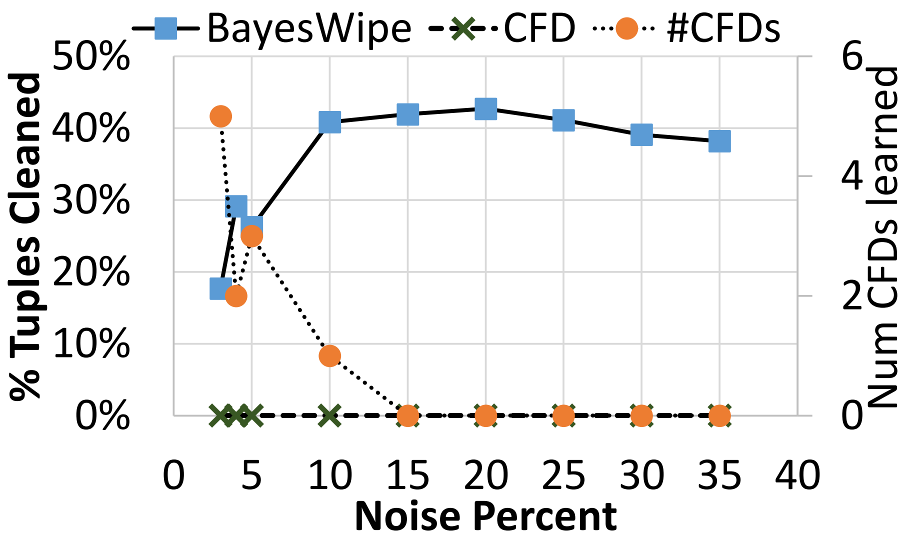


Figure 7.1: % Performance of BayesWipe Compared to CFD, for the Used-car Dataset.

I show a portion of the grid search where $\alpha = 2\beta/3$.

The "values corrected" data points in the graph correspond to the number of erroneous attribute values that the algorithm successfully corrected (when checked against the ground truth). The "false positives" are the number of legitimate values that the algorithm changes to an erroneous value. When cleaning the data, my algorithm chooses a candidate tuple based on both the prior of the candidate as well as the likelihood of the correction given the evidence. Low values of α, β give a higher weight to the prior than the likelihood, allowing tuples to be changed more easily to candidates with high prior. The "overall gain" in the number of clean values is calculated as the difference of clean values between the output and input of the algorithm.

If I set the parameter values too low, I will correct most wrong tuples in the input dataset, but I will also 'overcorrect' a larger number of tuples. If the parameters are set too high, then the system will not correct many errors — but the number of

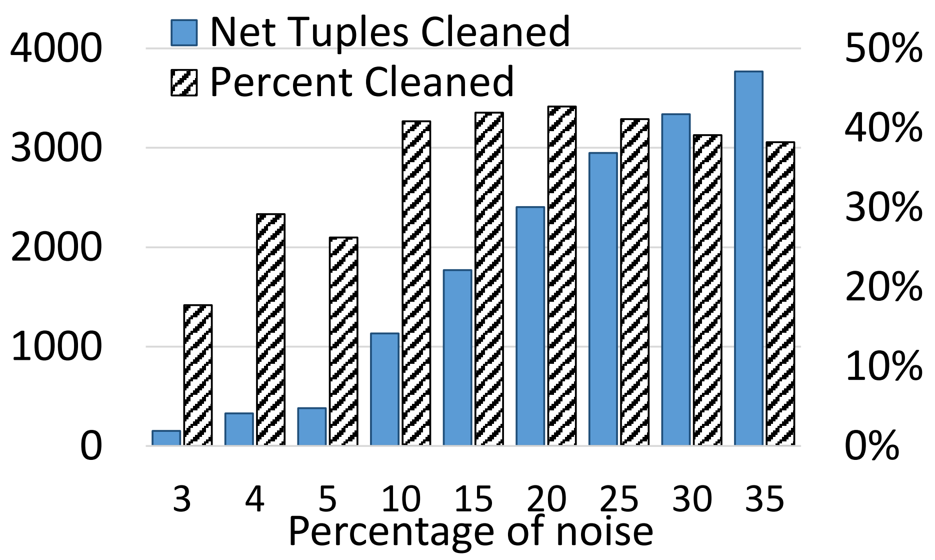


Figure 7.2: % Net Corrupt Values Cleaned, Car Database

'overcorrections' will also be lower. Based on these experiments, I picked a parameter value of $\alpha = 3.7, \beta = 2.1$ and kept it constant for all my experiments.

Using probabilistic databases: I empirically evaluate the PDB-mode of BayesWipe in Figure 7.4. The first figure shows the system using the threshold determinization. I plot the precision and recall as the probability threshold for inclusion of a tuple in the resultset is varied. As expected, with low values of the threshold, the system allows most tuples into the resultset, thus showing high recall and low precision. As the threshold increased, the precision increases, but the recall falls.

In Figure 7.4b, I compare the precision of the PDB mode using top-k determinization against the deterministic mode of BayesWipe. As expected, both the modes show high precision for low values of k, indicating that the initial results are clean and relevant to the user. For higher values of k, the PDB precision falls off, indicating that PDB methods are more useful for scenarios where high recall is important without sacrificing too much precision.

Online Query Processing: Since there is no existing work on querying autonomous

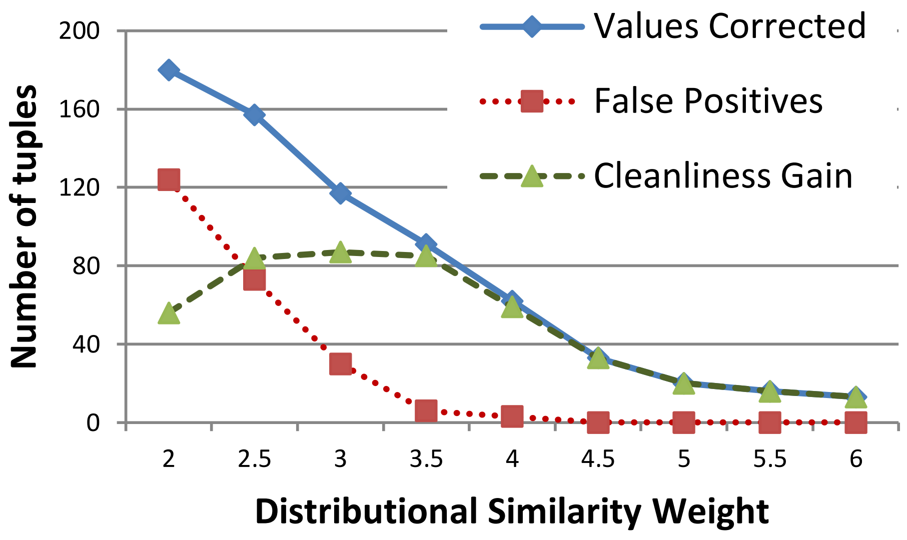
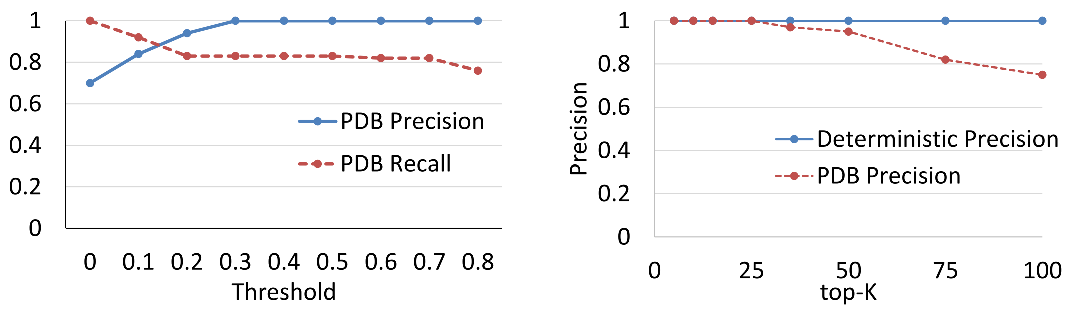


Figure 7.3: Net Corrections vs γ. (The x-axis Values Show the Un-normalized Distributional Similarity Weight, Which is Simply $\gamma \times 3/5$.)

data sources in the presence of data inconsistency, I consider a **keyword query** system as my baseline. I evaluate the precision and recall of my method against the ground truth and compare it with the baseline.

I issued randomly generated queries to both BayesWipe and the baseline system. Figure 7.5 shows the average precision over 10 queries at various recall values. It shows that my system outperforms the keyword query system in precision, especially since my system considers the relevance of the results when ranking them. On the other hand, the keyword search approach is oblivious to ranking and returns all tuples that satisfy the user query. Thus it may return irrelevant tuples early on, leading to a loss in precision.

This shows that my proposed query ranking strategy indeed captures the expected relevance of the to-be-retrieved tuples, and the query rewriting module is able to generate the highly ranked queries.



(a) Precision and recall of the PDB method using a threshold.

(b) top-k precision of PDB vs deterministic method.

Figure 7.4: Results of Probabilistic Method.

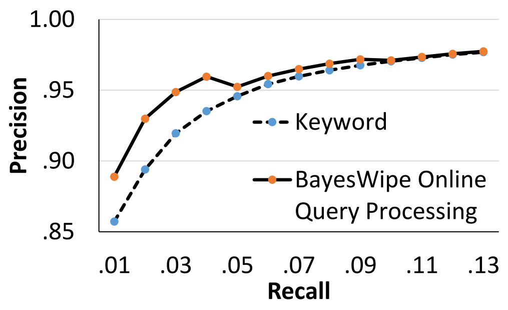


Figure 7.5: Average Precision vs Recall, 20% Noise.

Figure 7.6 shows the improvement in the absolute numbers of tuples returned by the BayesWipe system. The graph shows the number of true positive tuples returned (tuples that match the query results from the ground truth) minus the number of false positives (tuples that are returned but do not appear in the ground truth result set). We also plot the number of true positive results from the ground truth, which is the theoretical maximum that any algorithm can achieve. The graph shows that the BayesWipe system outperforms the keyword query system at nearly every level of noise. Further, the graph also illustrates that — compared to a keyword query baseline — BayesWipe closes the gap to the maximum possible number of tuples to a large extent. In addition to showing the performance of BayesWipe against the

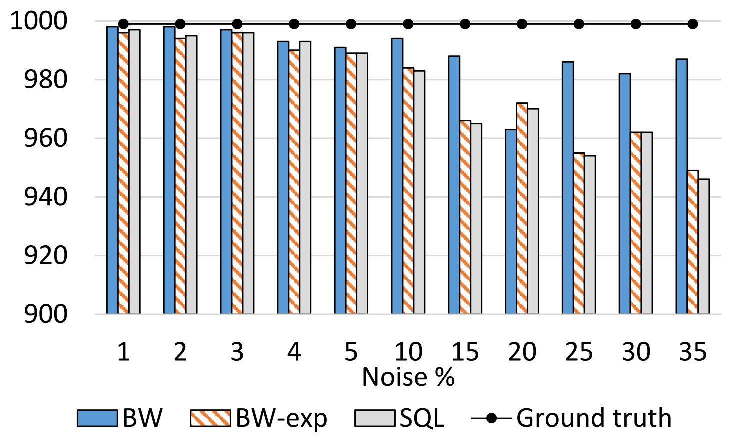


Figure 7.6: The improvement in the quality of the results (true positives minus false positives) by BayesWipe, compared against BayesWipe-exp (without query relaxation) and a keyword query baseline.

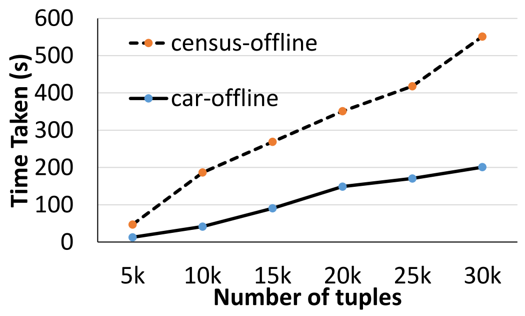
keyword query baseline, we also show the performance of BayesWipe without the query relaxation part (called BW-exp [2]). We can see that the full BayesWipe system outperforms the BW-exp system significantly, showing that query relaxation plays an important role in bringing relevant tuples to the resultset, especially for higher values of noise.

Efficiency:

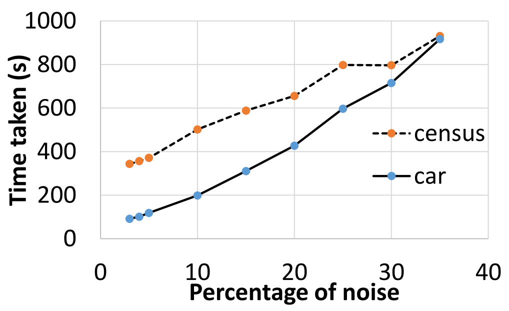
Figure 7.7 shows the performance of BayesWipe. In For the offline mode of BayesWipe, in Figure 7.7a I evaluate the time taken as the number of tuples in the database increases, and in Figure 7.7b I show the time taken as the noise varies. As can be seen from the figures, the offline methods complete in a time that can be considered reasonable for an offline, one-time process on the database.

For the online mode, Figure 7.7c shows how the time taken per query varies with the number of tuples, and Figure 7.7d shows the trend with respect to noise. While

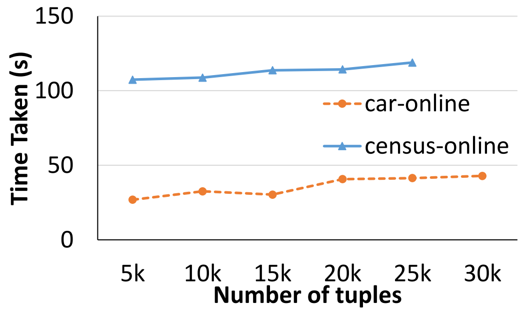
[2]BW-exp stands for BayesWipe-expanded, since the only query rewriting operation done is query expansion.



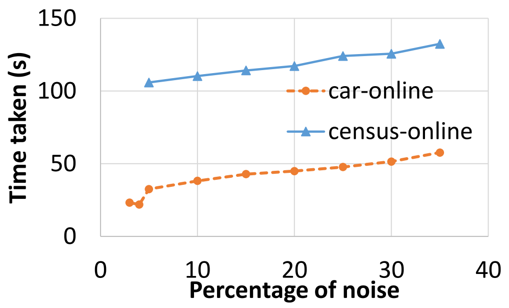
(a) Time vs. #Tuples (Offline)



(b) Time vs. %Noise (Offline)



(c) Time vs. #Tuples (Online)



(d) Time vs. %Noise (Online)

Figure 7.7: Performance Evaluations

the time taken by the online method is higher than expected for an online method, there are a couple of salient points to note. First, the online method is most useful for scenarios where the database is not under the control of the user, so the online method may be the only possible method to get clean data from the system. Second, note that the trend of increase of the time taken as the number of tuples increases is much more gradual when compared to the offline instance, and in fact tends to flatten out towards higher tuple sizes. This is because the online method uses only the portion of the database that is relevant to the query (through the rewritten queries).

Evaluation on real data with naturally occurring errors: In this section I used a dataset of 1.2 million tuples crawled from the cars.com website [3] to check

[3]http://www.cars.com

the performance of the system with real-world data, where the corruptions were not synthetically introduced. Since this data is large, and the noise is completely naturally occurring, I do not have ground truth for this data. To evaluate this system, I conducted an experiment on Amazon Mechanical Turk. First, I ran the offline mode of BayesWipe on the entire database. I then picked only those tuples that were changed during the cleaning, and then created an interface in mechanical turk where only those tuples were shown to the user in random order. Due to resource constraints, the experiment was run with the first 200 tuples that the system found to be unclean.

An example is shown in Figure 7.8. The turker is presented with two cars, and she does not know which of the cars was originally present in the dirty dataset, and which one was produced by BayesWipe. The turker will use her own domain knowledge, or perform a web search and discover that a Mazda CX-9 touring is only available in a 3.7l engine, not a 3.5l. Then the turker will be able to declare the second tuple as the correct option with high confidence.

The results of this experiment are shown in Table 7.1. As can be seen, the users consistently picked the tuples cleaned by BayesWipe more favorably compared to the original dirty tuples, proving that it is indeed effective in real-world datasets. Notice that it is not trivial to obtain a 56% rate of success in these experiments. Finding a tuple which convinces the turkers that it is better than the original requires searching through a huge space of possible corrections. Based on the cardinality of the domain, the probability that a random substitution will provide the correct tuple is vanishingly small; in the given case, it is less than 0.025%.

The first row of Table 7.1 shows the fraction of tuples for which the turkers picked the version cleaned by BayesWipe and indicated that they were either 'very confident' or 'confident'. The second row shows the fraction of tuples for all turker confidence values, and therefore is a less reliable indicator of success.

In order to show the efficacy of BayesWipe I also performed an experiment in which the same tuples (the ones that BayesWipe had changed) were modified by a random perturbation. The random perturbation was done by the same error process as described before (typo, deletion, substitution with equal probability). Then these tuples (the original tuple from the database and the perturbed tuple) were presented as two choices to the turkers. The preference by the turkers for the randomly perturbed tuple over the original dirty tuple is shown in the third column, 'Random'. It is obvious from this that the turkers overwhelmingly do not favor the random perturbed tuples. This demonstrates two things. First, it shows the fact that BayesWipe was performing useful cleaning of the tuples. In fact, BayesWipe shows a tenfold improvement over the random perturbation model, as judged by human turkers. This shows that in the large space of possible modifications of a wrong tuple, BayesWipe picks the correct one most of the time. Second, it provides additional support for the fact that the turkers are picking the tuple carefully, and are not randomly submitting their responses. This fact is further supported by experiments shown later.

Interestingly, note that the percentages shown in this table are not a fraction of the total number of tuples, but only of the number of tuples changed by BayesWipe. Therefore, as long as BayesWipe scores higher than the original data, the resulting database is cleaner.

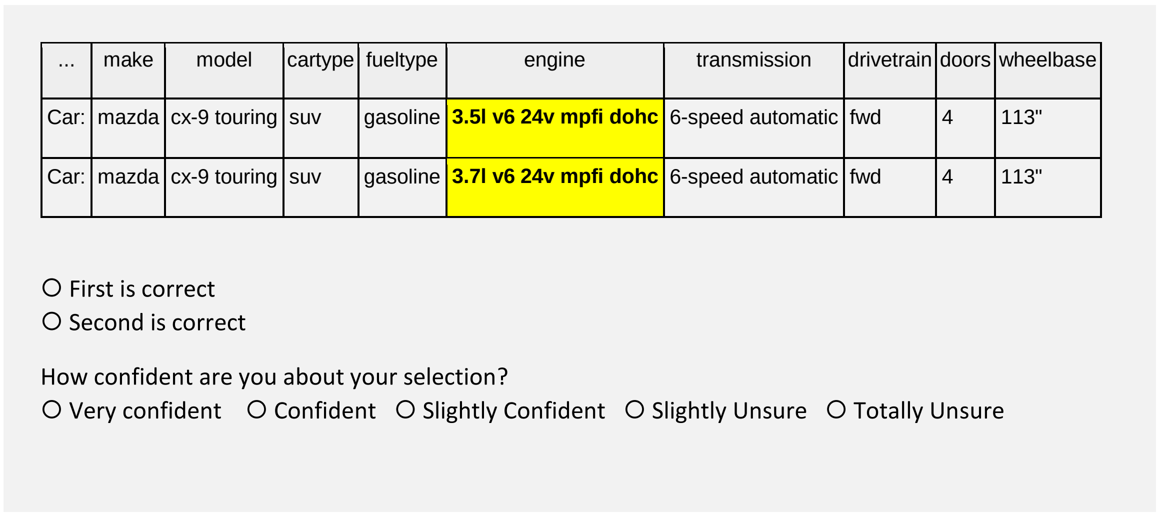
Another concern was that the mechanical turk users will not provide reliable answers, since there is a monetary incentive to answer as many questions as possible in a short amount of time. There were two approaches to mitigate this risk: showing the same question to multiple turkers, and taking the majority vote; or to insert known answers into the questions, and discard any turkers who fail to provide the expected answer. In this experiment, I chose the second approach. I created a small set of manually curated clean tuples, and automatically corrupted them with a substition,

Confidence	**BayesWipe**	**Original**	**Random**	**Increase over Random**
High confidence only	56.3%	43.6%	5.5%	50.8% points (10x better)
All confidence values	53.3%	46.7%	12.4%	40.9% points (4x better)

Table 7.1: Results of the Mechanical Turk Experiment, showing the percentage of tuples for which the users picked the results obtained by BayesWipe as against the original tuple. Also shows performance against a random modification.

deletion or typo. Both the original clean tuple and the corrupted tuple were presented to the turkers, and there was no visible distinction between this control tuple and the actual tuples from the study. For every 10 tuples of the query, I inserted 3 control tuples. If the turker responds correctly to at least 2 out of the 3 control questions, their answers were included in the results, otherwise, they were discarded.

I found that among 20 respondents, only one failed the control test. Repeating the experiment with a higher monetary reward (\$0.30 instead of \$0.20) did not change this observation significantly. As a result of this encouraging result, I determined that further measures to prevent turker misuse of the system was unnecessary. In this experiment, I also found the average fraction of known answers that the turkers gave wrong answers to. This value was 8%. This leads to the conclusion that the difference between the turker's preference of BayesWipe over both the original tuples (which is 12%) and the random perturbation (which is 50%) are both significant.



...	make	model	cartype	fueltype	engine	transmission	drivetrain	doors	wheelbase
Car:	mazda	cx-9 touring	suv	gasoline	**3.5l v6 24v mpfi dohc**	6-speed automatic	fwd	4	113"
Car:	mazda	cx-9 touring	suv	gasoline	**3.7l v6 24v mpfi dohc**	6-speed automatic	fwd	4	113"

○ First is correct
○ Second is correct

How confident are you about your selection?
○ Very confident ○ Confident ○ Slightly Confident ○ Slightly Unsure ○ Totally Unsure

Figure 7.8: A Fragment of the Questionnaire Provided to the Mechanical Turk Workers.

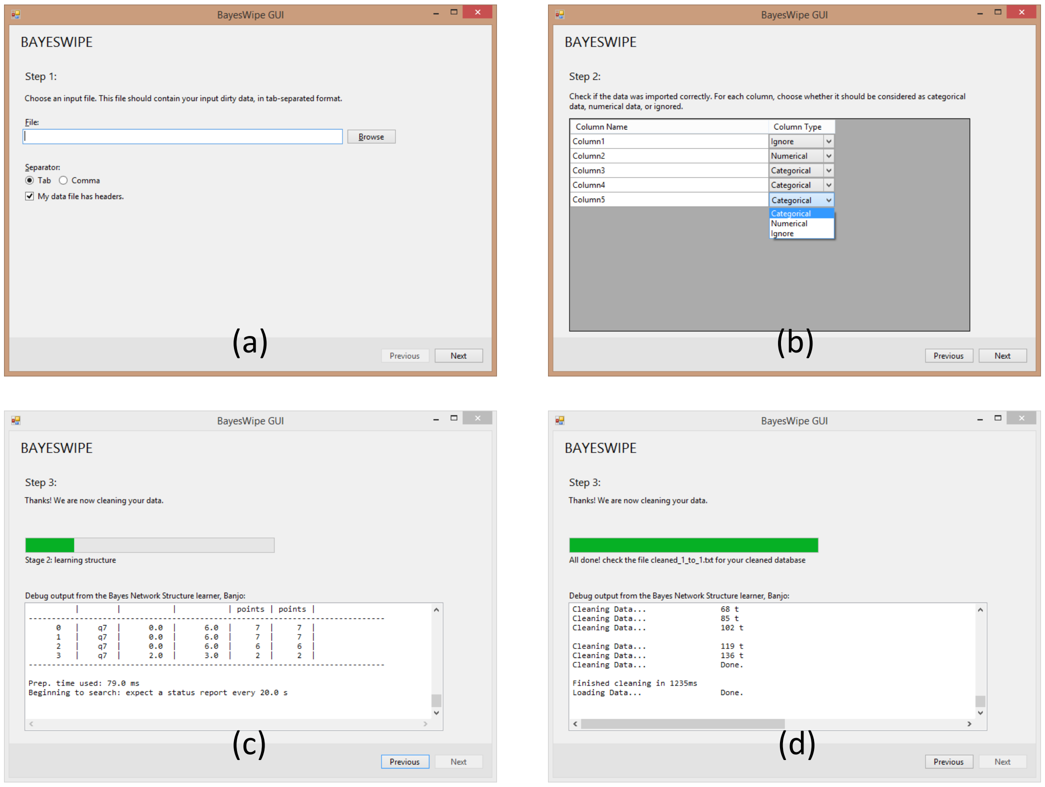
Chapter 8

THE BAYESWIPE APPLICATION

This chapter details the system, BayesWipe (De, 2014), that has been made available publicly, highlighting its working and architecture.

8.1 Objective

The BayesWipe software aims to let anyone perform probabilistic data cleaning of any dataset of their choice easily, using a downloadable, graphical interface. It was



Step 1:

Choose an input file. This file should contain your input dirty data, in tab-separated format.

File:

Separator: [✓ Tab] Comma

[✓] My data file has headers.

Step 2:

Check if the data was imported correctly. For each column, choose whether it should be considered as categorical data, numerical data, or ignored.

Column Name	Column Type
Column1	Ignore
Column2	Numerical
Column3	Categorical
Column4	Categorical
Column5	Categorical

Categorical
Numerical
Ignore

Step 3:

Thanks! We are now cleaning your data.

Stage 2: learning structure

Debug output from the Bayes Network Structure learner, Banjo:

```
          |         |           |           | points | points |
-----------------------------------------------------------------------------
     0    |    q7   |     0.0   |     6.0   |    7   |    7   |
     1    |    q7   |     0.0   |     6.0   |    7   |    7   |
     2    |    q7   |     0.0   |     6.0   |    6   |    6   |
     3    |    q7   |     2.0   |     3.0   |    2   |    2   |
-----------------------------------------------------------------------------

Prep. time used: 79.0 ms
Beginning to search: expect a status report every 20.0 s
```

Step 3:

Thanks! We are now cleaning your data.

All done! check the file cleaned_1_to_1.txt for your cleaned database

Debug output from the Bayes Network Structure learner, Banjo:

```
Cleaning Data...                68 t
Cleaning Data...                85 t
Cleaning Data...                102 t

Cleaning Data...                119 t
Cleaning Data...                136 t
Cleaning Data...                Done.

Finished cleaning in 1235ms
Loading Data...                 Done.
```

Figure 8.1: Screenshots of the BayesWipe system. (a) The initial selection screen, (b) The data type selection screen, (c) Computation processing, (d) Computation complete.

desirable to make the software as easy to use as possible, allowing input and output in a simple, widely accepted format. It was also an objective to make it free from complicated procedures, such as installation of other software.

To this end, BayesWipe has been made available for downloaded from the website (http://bayeswipe.sushovan.de). It runs on the Windows platform (Windows 7 onwards). It requires Java (in order to run the Bayes Network structure learning module, Banjo).

8.2 User Input

The input data can be formatted as either a comma separated, or a tab separated input file. Since most databases (MySQL, SQL Server, Oracle) can export their data to a CSV file with a simple command, this is a convenient container to take the data input from.

The user starts the program and chooses her input data file. BayesWipe then prompts the user to double check if the input data was properly imported — if the column names were not present in the input data file, the user can choose to input it here as well. The user can also help in the quantization of some attributes by specifying if the attributes are numerical in nature. If selected, the application will automatically figure out the bounds of the attribute values and quantize the attribute appropriately.

Certain attributes that are unique keys, such as social security numbers, serial numbers, or the VIN identifier for a car do not exhibit patterns across the dataset, and thus cannot be cleaned by BayesWipe. The user has an opportunity at this point to select those attributes to be ignored, so that the algorithm runs faster (since it has fewer attributes to compute correlations for).

After this, no further input from the user is required.

8.3 Operation

Having collected input from the user, the software now transforms the data. It standardizes the input file format to comma separated, finds the bounds of any numerical attributes and quantizes the data. It also removes any attributes that the user asked to ignore (for example, unique keys).

BayesWipe then invokes Banjo, to learn the structure of the Bayes network. In order to do so, it creates two files: a smaller, sampled and quantized input file for Banjo with attribute values converted to integer identifiers; and a configuration file for Banjo that provides configuration information. This includes information such as the duration for which the structure learner should be run, the algorithm that should be used, etc. Banjo produces the output and writes the graph in the .dot format, which BayesWipe reads back.

BayesWipe then learns the parameters of the Bayes network using Infer.NET. In order to do so, it programmatically recreates the Bayes network structure that was produced by Banjo into a structure that Infer.NET recognizes, and then provides a sampled version of the input file as input. It also learns the error statistic and computes the candidate index.

Finally, BayesWipe cleans the input data tuple by tuple and writes the output to a text file in a comma separated format (that can be read back into the user's database for further processing).

BayesWipe was implemented in C# 3.5 on the Windows platform. Figure 8.1 shows the screenshots of system in operation. The first two screens show how the user provides the input data and information, the second two screens show the data cleaning operation in progress.

8.4 Future Work

To make it even easier for users to run BayesWipe, it can be implemented as a web-application. There are several practical challenges for this to work well: data cleaning is an extremely computation intensive process. Performing cleaning online for whoever uploads their (possibly massive) datasets could easily prove to be prohibitively expensive. Further, if the data is of a sensitive or confidential nature, users may be unwilling to upload their data to a server they do not trust.

A second improvement to the system would be to bring online query capabilities to the system. In such a case, making it a web-based system that can query certain well-known datasets (such as cars.com) could be a valuable idea. Once again, if users wish to run an online query against their own private datasets, making a copy of the software available for download would also be a good idea.

Chapter 9

MAP-REDUCE FRAMEWORK

BayesWipe is most useful for big-data related scenarios. The online mode of BayesWipe already works for big data scenarios by optimising the rewritten queries it issues, but the offline mode has so far been shown as a single-threaded application. It makes sense to implement it in a Map-Reduce architecture, so that I can run it very quickly for massive datasets.

It is very useful and important to have big-data that is actually clean and reliable and can be further processed using external tools, hence this is particularly suited for BayesWipe.

Since it is a nice abstraction to think about when considering parallelizing programs, the map-reduce architecture is a good fit for this problem. Extensive support for running Map-Reduce jobs is also available through many service providers (Amazon AWS, Microsoft Azure, Google App Engine).

9.1 Original Implementation

BayesWipe has two modes: online and offline. This chapter only considers parallelizing the offline mode of BayesWipe, since the online mode can already work for large datasets without affecting the run time too severely.

So far, BayesWipe-Offline has been implemented as a two-phase, single threaded program. In the first phase, the program learns the Bayes network (both structure and parameters), learns the error statistics, and creates the candidate index. In the second phase, the program goes through every tuple in the input database, picks a set of candidate tuples, and then evaluates the $P(T^*|T)P(T^*)$ for every candidate tuple, and replaces T with the T^* that maximises that value. Since the learning is typically done on a sample of the data, it is more important to focus on the second phase for the parallelizing efforts. Later, I will see how the learning of the error statistics can also be parallelized.

9.2 Simple Approach

The simplest approach to parallelizing the tuples is to run the first phase (the learning phase) on a single machine, called the master machine. Then, a copy of the bayes network (structure and CPTs), the error statistics, and the candidate index can be sent to a number of other machines. Each of those machines also receives a fraction of the input data. With the help of the generative model and the input data, it can then clean the tuples, and then create the output.

If I express this in Map-Reduce terminology, I will have a pre-processing step where I create the generative and error models. The Map-Reduce architecture will have only mappers, and no reducers. The result of the mapping will be the tuple $\langle T, T^* \rangle$.

The problem with this approach is that in a truly big data scenario, the candidate index becomes very large. Indeed, as the number of tuples increases, the size of the domain of each attribute also increases. Further, the number of different combinations, and the number of erroneous values for each attribute also increase. All of this results in a rather large candidate index. Transmitting and using the entire index on each mapper node is wasteful of both network, memory, (and if swapped out, disk resources).

This increase is shown in Figure 9.1. The x-axis shows the dataset over which the index is built. The items curves shows the number of entries in the index, computed as the number of attribute values used as the key and the tuples that are part of it.

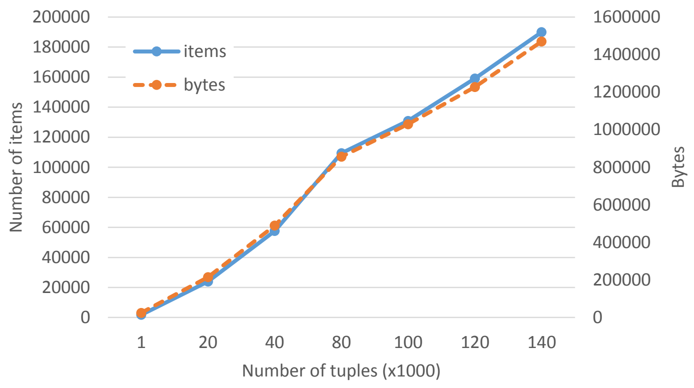


Figure 9.1: Size of the Index as the Size of the Dataset Grows

The bytes curve shows the amount of space consumed in holding the item in memory. As is apparent from the graph, the space taken by the index grows quickly. For a successful distribution of the computation, it is necessary, therefore, to also distribute the size of the index.

9.3 Improved Approach

In order to split both the input tuples and the candidate index, I use a two-stage approach. In the first stage, I run a map-reduce that splits the problem into multiple shards, each shard having a small fraction of the candidate index. Each input tuple may be sent to multiple shards. In the second stage, I run a simple map-reduce that picks the best output from stage 1 for each input tuple to produce the final clean database.

Stage 1: I can intelligently partition both the tuples and the candidate index into classes, so that I have a smaller index at each node. Fundamentally, I am operating on an input tuple T and a set of candidate tuples, the T^*s. Suppose the candidate index is created on k attributes, $A_1 ... A_k$. Therefore, I can say that for every tuple T, and one of its candidate tuples T^*, they will have at least one matching attribute

a_i from this set. The idea is that I can use this common element a_i to predict which shards the candidate T^*s might be available in.

In the map-reduce architecture, it is possible to define a 'partition' function. Given a mapped key-value pair, this function determines which reducer nodes will process the data. I use the value of the matching attribute, a_i as the partition function. However, notice that the number of possible values that $A_1 ... A_k$ can take is rather large. If I naïvely usea_i as the partition function, I will have to create those many reducer nodes. Therefore, more generally, I hash this value into a fixed number of reducer nodes, using a deterministic hash function. This will then find all candidate tuples that are eligible for this tuple, compute the similarity, and output it.

Example: Suppose I have tuple T_1 that has values $(a_1, a_2, a_3, a_4, a_5)$. Suppose our candidate index is created on attributes A_1, A_2, A_4. This means that any candidates T^* that are eligible for this tuple have to match one of the values a_1, a_2 or a_4. Then the mapper will create the pairs (a_1, T), (a_2, T) and (a_4, T), and send to the reducers. The partition function is hash of the key - so in this case, the first one will be sent to the reducer number $hash(A1 = a1)$, the second will be sent to the reducer numbered $hash(A2 = a2)$, and so on. □

In the reducer, the similarity computation and prior computation part of BayesWipe is run. Since each reducer only has a fraction of the candidate index (the part that matches $A1 = a1$, for instance), it can hold it in memory and computation is quite fast. Each reducer produces a pair $(T_1, (T_1^*, \text{score}))$.

Stage 2: This stage is a simple max calculation. The mapper does nothing, it simply passes on the key-value pair $(T_1, (T_1^*, \text{score}))$ that was generated in the previous Map-Reduce job. Notice that the key of this pair is the original, dirty tuple T_1. The Map-Reduce architecture thus automatically groups together all the possible clean versions of T_1 along with their scores. The reducer picks the best T* based on

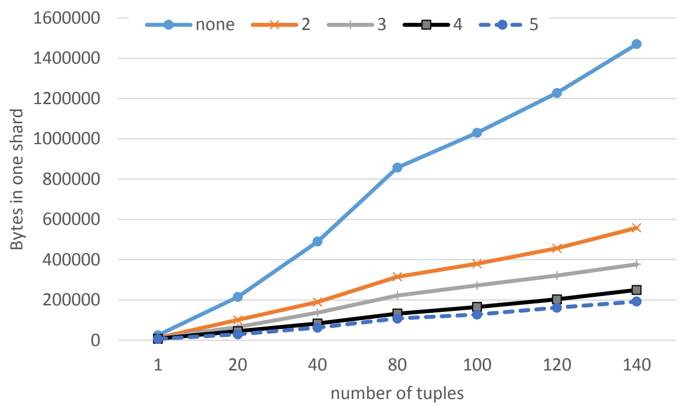


Figure 9.2: Size of the Index vs the Number of Tuples (in Thousands) in the Dataset, for Various Number of Shards.

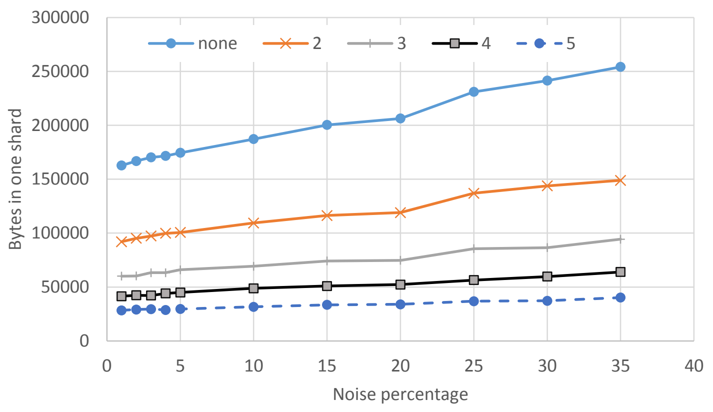


Figure 9.3: Size of the Index vs the Noise in the Dataset, for Various Number of Shards.

the score (using a simple max function), and outputs it to the database.

9.4 Results of This Strategy

In Figure 9.2 and Figure 9.3 I can see how this map reduce strategy helps in reducing the memory footprint of the reducer. First, I plot the size of the index that needs to be held in each node as the number of tuples in the input increases. The topmost curve shows the size of index in bytes if there was no sharding - as expected, it increases sharply. The other curves show how the size of the index in the one of the nodes varies for the same dataset sizes. From the graph, it can be seen that as the number of tuples increases, the size of the index grows at a lower rate when the number of shards is increased. This shows that increasing the number of reduce nodes is a credible strategy for distributing the burden of the index.

In the second figure (Figure 9.3), we see how the size of the index varies with the percentage of noise in the dataset. As expected, when the noise increases, the number of possible candidate tuples increase (since there are more variations of each attribute value in the pool). Without sharding, we see that the size of the dataset increases. While the increase in the size of the index is not as sharp as the increase due to the size of the dataset, it is still significant. Once again, we observe that as the number of shards is increased, the size of the index in the shard reduces to a much more manageable value.

These graphs show the size in bytes; the number of items shows a very similar trend. I refer the reader to Figure 9.1 to show how close this similarity is.

9.5 Potential Disadvantages

While this architecture does solve the problem of the index size, the disadvantage of using a 2-stage map-reduce is that it requires a very large temporary disk-space to hold the (T, (T*, score)) pair. Recall that this is the output of the first Map-Reduce job. This is the price I pay for implementing this architecture in Map-Reduce directly, without worrying about modifying map-reduce architecture further.

The alternate implementation to this would be to slightly change the Map-Reduce architecture, so that the second stage MR is not necessary. For example, this can be accomplished by implementing a multi-layer Map-Map-Reduce framework. Recall that the second Map-Reduce job did not process anything in the Mapper, it simply passed the key-value pair generated by the previous stage. If we can implement that as a ‘middle-layer’ mapper, then we can directly run the entire process in a single run, without having to store the intermediate results. Other solutions, such as streaming the output of the first stage to the second stage are also viable.

9.6 Further Possible Improvements

In future work, in addition to parallelizing the data cleaning part of the project, the error statistic learning part can also be easily parallelized. First, error statistic for each attribute is computed separately. Thus splitting the problem on the attribute is a straightforward way to parallelize the computation.

For each attribute, the computation is quadratic because it looks at every possible pair of attributes. Ideas similar to the classic Map Reduce problem of matrix multiplication can be applied to solve this.

Chapter 10

DEPENDENCIES AMONG ATTRIBUTES IN PROBABILISTIC DATABASES

10.1 Functional Dependencies Compared to Bayes Networks

In the previous chapters of this thesis, I developed a system to clean data based on Bayes networks. Bayes networks were very useful as a tool to model the generative model of the data, since it can effectively encode the conditional independences between attributes and provide an efficient method of storing and reasoning over them. However, in addition to serving as a generative model, Bayes networks can also be used to gain insight into the dependencies between the attributes themselves. For example, Mellott (2013) uses the parameters of the Bayes network learned from the data to infer a set of conditional functional dependencies.

Such dependencies between attributes (functional dependencies, and its more generalized versions such as approximate and conditional functional dependencies) are faster to reason with, compared to Bayes networks, and can be more intuitively presented for inspection by domain experts. Thus, they can be used to verify that the internal state of the system that is performing the data cleaning is reasonable. They can also be used for generating automated explanations for the output of the program (similar to recommender systems).

Recall that in the probabilistic mode of BayesWipe, the generated output is a probabilistic database. It is quite well-known how to find the dependencies for a deterministic database, but there is very little prior work on doing the same for probabilistic databases. In this chapter, I show how to determine the confidence of such dependencies in probabilistic databases, and hence, how to mine them.

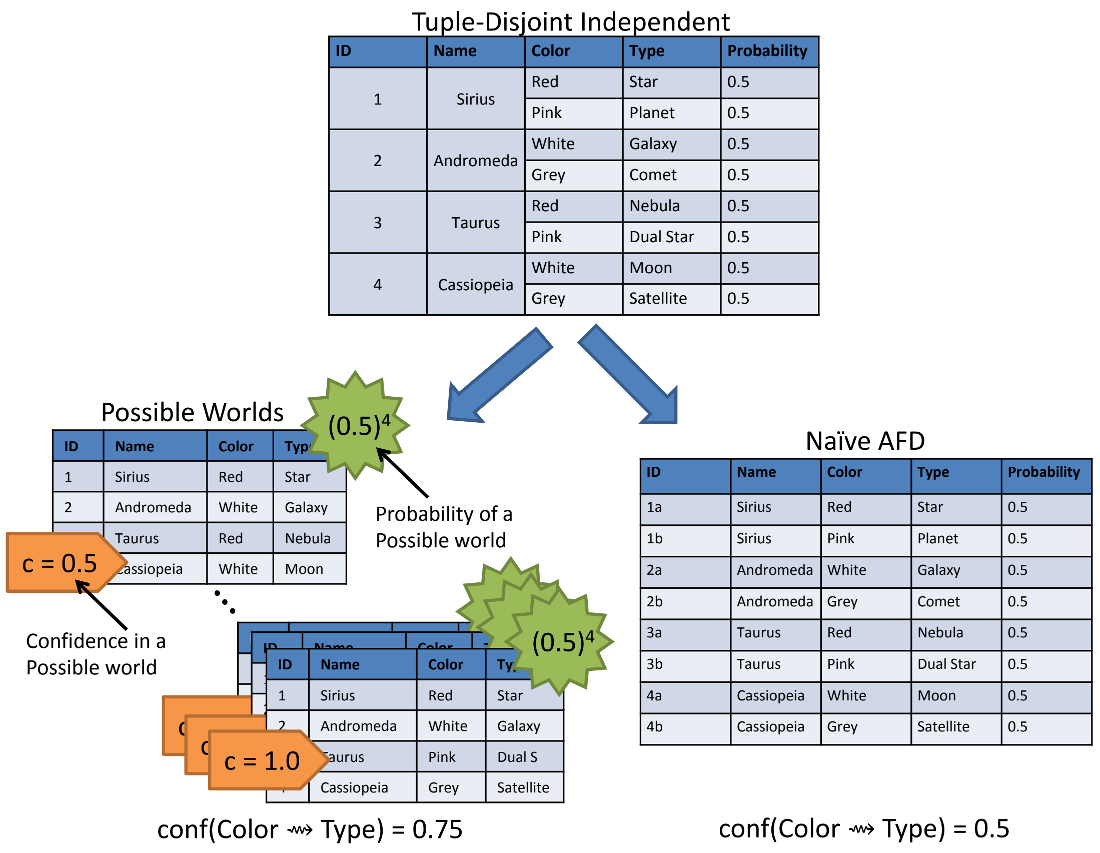
10.2 Motivation for Functional Dependencies in Probabilistic Databases

A lot of data generated today, especially that obtained from the web, is dirty, untrustworthy or uncertain. Yet we continue to store them in database engines that are ill-equipped to handle uncertainty. Handling uncertainty isn't a simple case of adding a 'probability' attribute – uncertain data has correlations, causations and query processing on such data is a probabilistic inference problem. Probabilistic databases (PDB) allow the data to be represented in a manner that fully reflects the different possibilities for the ground truth. Further processing can then take into account all the alternatives, not just the most likely one.

In the case of traditional databases, dependencies in the form of exact and approximate functional dependencies have proven to be very useful. They are used for fast query processing, rectification of data and can also be used to gain insight into the data and generate explanations for recommender systems. There are algorithms that will evaluate functional dependencies between attributes to aid in schema normalization (slo, 2013), that will evaluate approximate functional dependencies (AFD), which helps filling out missing data in incomplete databases (Agrawal and Srikant, 1994). There are also conditional functional dependencies (CFDs), which help in cleaning and correcting data (Bohannon *et al.*, 2007). However, such dependencies and algorithms are missing for probabilistic data. In this chapter, I extend these very useful dependencies so that they work with PDBs in general. I generalize FDs to probabilistic functional dependencies (pFD), AFDs to probabilistic approximate functional dependencies (pAFD), and their conditional counterparts respectively to CpFD and CpAFD. I also investigate the relationship between these dependencies. In particular, I point out which of these dependencies are generalizations of, and hence subsume, others. I also provide fast algorithms for evaluating the confidence of these dependencies on probabilistic database, with a special focus on tuple- independent and tuple-disjoint independent databases (Brin *et al.*, 1997). With the help of these algorithms, I describe how I can mine these dependencies from data, by using efficient methods to prune the search space of dependencies.

Motivating example Consider the case where a large corporation has deployed a set of sensor networks across its various departments to track operating environment, power supply and malfunctioning equipment. Suppose that the data records of each department are kept in separate relations, and are recorded in the following format: each row has a timestamp, the exception in the environment conditions (if any) such as 'high temperature' or 'high humidity', exceptions in power supply such as 'fluctuating' or 'low voltage', and the level of functioning of the equipment 'working' or 'not working'. These readings are inherently uncertain, since they are taken by sensors. Finding the confidence of the pAFD (environment $\rightsquigarrow$ malfunctioning) can tell us in which department, equipment malfunctioning is correlated with environmental concerns. This can then be compared with the confidence of the malfunctioning being correlated with electrical reasons. Running a pAFD mining algorithm might also bring out certain dependencies that were previously unknown. In this example, finding a dependency like (power supply $\rightsquigarrow$ environment) may give an indication that the operating environment is not ideal because of the erratic power supply.

Another scenario that benefits from uncertain dependencies is the following. Assume that two or more astronomers are observing and recording various objects in the sky, like in the Sloan Digital Sky Survey (slo, 2013). They note various attributes of the objects, including the color, type, speed, frequency of oscillation and position. Such data is most naturally represented as a probabilistic database, where each tuple represents a different object in the sky and reflects the curator's confidence in the


Tuple-Disjoint Independent
Possible Worlds
(0.5)^4
Probability of a Possible world
c = 0.5
Confidence in a Possible world
(0.5)^4
c = 1.0
Naïve AFD
conf(Color ⇝ Type) = 0.75
conf(Color ⇝ Type) = 0.5

Tuple-Disjoint Independent

ID	Name	Color	Type	Probability
1	Sirius	Red	Star	0.5
		Pink	Planet	0.5
2	Andromeda	White	Galaxy	0.5
		Grey	Comet	0.5
3	Taurus	Red	Nebula	0.5
		Pink	Dual Star	0.5
4	Cassiopeia	White	Moon	0.5
		Grey	Satellite	0.5

Possible Worlds

ID	Name	Color	Typ
1	Sirius	Red	Star
2	Andromeda	White	Galaxy
	Taurus	Red	Nebula
	assiopeia	White	Moon

ID	Name	Color	Ty
1	Sirius	Red	Star
2	Andromeda	White	Galaxy
	aurus	Pink	Dual S
	Cassiopeia	Grey	Satellite

Naïve AFD

ID	Name	Color	Type	Probability
1a	Sirius	Red	Star	0.5
1b	Sirius	Pink	Planet	0.5
2a	Andromeda	White	Galaxy	0.5
2b	Andromeda	Grey	Comet	0.5
3a	Taurus	Red	Nebula	0.5
3b	Taurus	Pink	Dual Star	0.5
4a	Cassiopeia	White	Moon	0.5
4b	Cassiopeia	Grey	Satellite	0.5

Figure 10.1: Why pAFDs differ from a naïve interpretation of AFD. **(top)** A tuple-disjoint independent database. **(right)** An attempt to find the AFD naïvely results in confidence of 0.5. **(left)** The semantically correct value of 0.75.

observer as well as the observer's confidence in the data. Having represented this data, the curator can run a pAFD finding algorithm to discover dependencies that were as yet unknown, of the form $(color, speed \rightsquigarrow type)$. If the curator is aware of some dependencies that are expected to hold, he can verify their validity by running the appropriate dependency checking algorithm on this data.

One pertinent question is whether these extensions are important or interesting enough to consider, or whether AFDs can be used directly. Pending empirical study, (see Section 10.6.2), I demonstrate their importance with an example. Let us say that I have a probabilistic database as shown in Figure 10.1. I am interested in finding out whether or not the dependency Color $\rightsquigarrow$ Type has a high confidence. Strictly speaking, I cannot evaluate its AFD, because the concept of AFD does not apply to probabilistic data. However, I can apply what might be called an 'intuitive' extension of an AFD to probabilistic data by considering each option as an independent tuple and weighing them according to their probabilities. If I do that, Figure 10.1 shows that this naïve method calculates a probability that is quite low, and different from the correct value dictated by probabilistic semantics.

The rest of the chapter is organized as follows. I start by defining the various dependencies for probabilistic databases, and in the following section I explore the theoretical connections between them. In Section 10.5 I propose some algorithms to evaluate the confidence of these quantities and show how to mine them, and in the section following that, show experiments that evaluate the effectiveness of these algorithms. I end with a discussion of related work and present our conclusions in Section 10.7.

10.3 Definitions

10.3.1 Probabilistic Database

In this chapter I follow the possible worlds model for a probabilistic database: a probabilistic database is a collection of possible worlds, with each possible world being a deterministic database with an associated probability of existence. Figure 10.1 shows a probabilistic database. The possible worlds representation is the set of relations on the bottom left.

I denote a deterministic relation by symbol R, which has attributes $(A_1, A_2, ..., A_n)$.

Sets of attributes are denoted by X or Y. Uncertain relations are denoted by D, and each uncertain relation comprises possible worlds $(P_1, P_2, ...P_m)$, with and attributes $(A_1, A_2, ..., A_n)$.

10.3.2 Probabilistic Functional Dependencies (pFD)

Given a deterministic relation R a functional dependency (FD) is defined as $(X \rightarrow Y)$, where X and Y are sets of attributes. The FD is said to hold if whenever two tuples share the same values of X, they have the same values of Y.

I can generalize this idea to probabilistic databases, as shown in Figure 10.2 by the 'uncertainty in data' axis. Given an uncertain relation D, a probabilistic functional dependency, pFD, is defined as $(X \rightarrow Y)$. The pFD is associated with a quantity called its confidence which is the fraction of possible worlds in which the corresponding FD holds.

Consider the possible world representation in Figure 10.1. In the first possible world, the data in tuple 1, *(Red, Star)* conflicts with tuple 3, *(Red, Nebula)*. As a result the contribution of that possible world towards the confidence of the pFD $Color \rightarrow Type$ is zero. The last possible world in the figure shows a non-zero contribution. The FD holds within the world, hence the entire probability of the world is added to the pFD confidence score.

It should be noted that pFDs suffer from the same kind of flaws that traditional FDs do. If the data is dirty, just a few tuples that do not conform to the pFD might cause an entire possible world to be not counted. I can address these concerns with pAFDs.

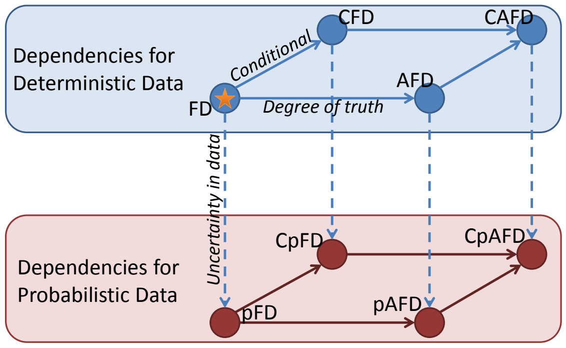


Figure 10.2: The relationship between various dependencies. FDs are the least general, when extended by adding degree of truth, I get AFDs; when extended to probabilistic databases, I get pFDs, when extended to include conditional dependencies, I get CFDs. Combinations of these properties give rise to the other dependencies.

10.3.3 Probabilistic Approximate Functional Dependencies (pAFD)

AFDs generalize FDs by adding the concept of a 'degree of truth' to an FD, which is also illustrated in Figure 10.2. Given a deterministic relation R, an approximate functional dependency AFD is defined as $(X \rightsquigarrow Y)$, where X and Y are sets of attributes, with X approximately determining Y. The confidence of an AFD may be defined in various ways. Following (Huhtala *et al.*, 1999), I define the AFD confidence as as one minus the minimum fraction of tuples that need to be removed from the relation for the FD to hold.

In an uncertain relation D, a probabilistic approximate functional dependency, pAFD, is defined as $(X \rightsquigarrow Y)$. The confidence of the pAFD is the expected confidence of the AFD over the possible worlds, (i.e. average of the confidence of the corresponding AFD in each possible world weighted by the probability of that world).

For example, in Figure 10.1, in the first possible world, the data in tuple 1, *(Red, Star)* conflicts with tuple 3, *(Red, Nebula)*. As a result only one of them can be considered as contributing towards the AFD within that possible world. A similar argument holds for tuples 2 and 4. I can therefore say that among the four tuples in the possible world, two support the AFD, and two don't; hence the confidence of the AFD is 0.5. In situations where data is both uncertain and noisy, pAFDs are useful for judging the relationship between attributes.

10.3.4 Conditional Probabilistic Functional Dependencies (CpFD)

I can extend functional dependencies in yet another way – by making them conditional on specific values of the data. Given a deterministic relation R, a CFD is a pair $(X \rightarrow Y, T_p)$. $X \rightarrow Y$ is a standard FD. T_p is the pattern tableau, which is a relation with the attributes $(X \cup Y)$. The semantics of a CFD (Bohannon *et al.*, 2007) are as follows: A CFD holds on R if the corresponding FD holds on the subset of tuples that *matches* the pattern tableau. Every tuple in T_p is either a constant or the wildcard character (_). A constant a in T_p matches only the constant a in R. The wildcard '_' in T_p matches any value in the real tuple. For a pair of tuples to violate the CFD, they must agree on every attribute in X but different values for some attributes in Y, and the set of attributes $X \cup Y$ must match the pattern tableau.

I can extend this concept to probabilistic databases. Given an uncertain relation D, a conditional probabilistic functional dependency, CpFD, is the pair $(X \rightarrow Y, T_p)$, where X and Y are sets of attributes and T_p is the pattern tableau. The confidence of a CpFD is the fraction of possible worlds where the corresponding CFD holds.

A CpFD is an extension of the concept of CFD to uncertain databases, but it does not tolerate dirty data. If even one tuple in a possible world violates the CpFD, the entire possible world probability is not counted.

10.3.5 Conditional Probabilistic Approximate Functional Dependencies (CpAFD)

Given a deterministic relation R, a conditional approximate functional dependency, CAFD, is defined as the pair $(X \rightsquigarrow Y, T_p)$. The confidence of the CAFD is one minus the fraction of tuples that need to be removed from the subset of tuples that match the pattern tableau T_p such that the CFD $(X \rightarrow Y)$ holds.

CAFDs do support a fractional confidence value, thus they can be used where the data is expected to be noisy and when the dependency holds only conditionally. Therefore it is useful to extend for probabilistic data. Given an uncertain relation D, a conditional probabilistic approximate functional dependency (CpAFD) is the pair $(X \rightsquigarrow Y, T_p)$. Its confidence is the weighted average of the confidence of the corresponding CAFD in each possible world, weighted by the probability of that possible world.

A CpAFD extends the notion of a functional dependency in the most general way among all of the dependencies discussed in this chapter. It supports fractional confidence values, possible world semantics, as well as operating on a select part of the database.

10.4 Relationships Among Dependencies

The dependencies defined in Section 10.3 are all generalizations of FDs, as shown in Figure 10.2. It is natural therefore, that I can express the less general dependencies in terms of the more general ones, and that I can induce relationships among them. That is what I will attempt to do in this section.

An FD is an AFD of confidence 1. AFDs allow dependencies that hold approximately, thus they extend FDs along the 'degree of truth' axis as shown in Figure 10.2. It should be noted that the confidence of an AFD can never be zero. This is because, in a relation R with at least one tuple, even if all different values of Y occur with the same values of X, I can remove all tuples except for one for each different set of values of X and have a non-zero set of tuples left in our database. Thus, the confidence of the AFD, which is is one minus the fraction of tuples removed, is non-zero. I can make a similar comparison between pFDs and pAFDs.

Theorem 1. *The confidence of a pAFD is always larger than the confidence of the corresponding pFD.*

Proof. In every possible world that the pFD holds, the confidence of the pAFD is 1. In every possible world that the pFD does not hold, the confidence of the pAFD is non-zero. Thus the confidence of the pAFD, which is the weighted sum of the confidences in each possible worlds, would be no smaller than that of the pFD. □

However, the reverse is not true. The information that a pAFD holds with a very high confidence does not imply that the pFD will have a high confidence, in fact, the confidence of the pFD may even be zero, as there might be a few tuples in every possible world that do not conform to the pFD.

Conditional dependencies extend each of the previous dependencies so that they can be specified over only a part of the data. This is illustrated in Figure 10.2 by the 'conditional' axis. However, the generalization to conditional dependencies can introduce inconsistencies. For example, if I know that an FD holds on a relation, the corresponding CFD is not guaranteed to hold, since the tableau could introduce impossible cases (Bohannon *et al.*, 2007). For example, in a CFD $(A \rightarrow B)$, if the pattern tableau has two tuples, one requiring the value of B to be b, the other requiring the value to be c, the CFD is clearly inconsistent, and no relation can satisfy it. However, in normal use cases, where the tableau of the CFD has been induced from the data, or else calculated in a non-malicious manner, intuitively I can state that if the FD holds, the CFD will also hold. The converse, however, is not true. If a CFD with a non-trivial tableau holds over a relation R, then I cannot guarantee that the corresponding FD holds.

A pattern tableau may select a non-zero number of tuples from some possible worlds but no tuples from others. In the special case where the tableau selects some tuples from each possible world, I can state that the confidence of the CpFD and the confidence of a CFD are equal. More generally, I can state the following theorem:

Theorem 2. *If a pFD $(X \rightarrow Y)$ holds over an probabilistic relation D with confidence p, then the CpFD $(X \rightarrow Y, T_p)$ also holds over R with a confidence not less than p, provided the CpFD is not inconsistent.*

Proof. Suppose the pFD holds on a certain possible world of D. The pattern tableau would then cause certain tuples to be eliminated from consideration. Even after elimination of a few tuples, the pFD will continue to hold, by definition. Thus that possible world will contribute towards the confidence of the CpFD. On the other hand, if there is a possible world in which the pFD does not hold, it is possible that those tuples that cause the pFD not to hold will be eliminated by the pattern tableau, causing the possible world to contribute to the confidence of the CpFD. So the fraction of possible worlds contributing to the CpFD is not smaller than the fraction contributing to confidence of the pFD. □

The same argument does not hold for pAFD and CpAFDs. In the case of CpAFDs, elimination of certain tuples may reduce its confidence. For example, consider the CpAFD $(A \rightsquigarrow \{B, C\}, T_1)$ where T_1 consists of the tuple $(\_, b_2, \_)$. If one of the possible worlds, P, of relation D contains a 50 tuples of (a_1, b_1, c_1) and 50 tuples of the form (a_1, b_2, c_2), (a_1, b_2, c_3) $\ldots (a_1, b_2, c_{51})$ the pAFD would have confidence 0.50, but the CpAFD would have confidence 0.02 (after eliminating the 50 tuples not matching T_1).

10.5 Assessing and Mining Probabilistic Dependencies

I consider two problems in the presented framework:

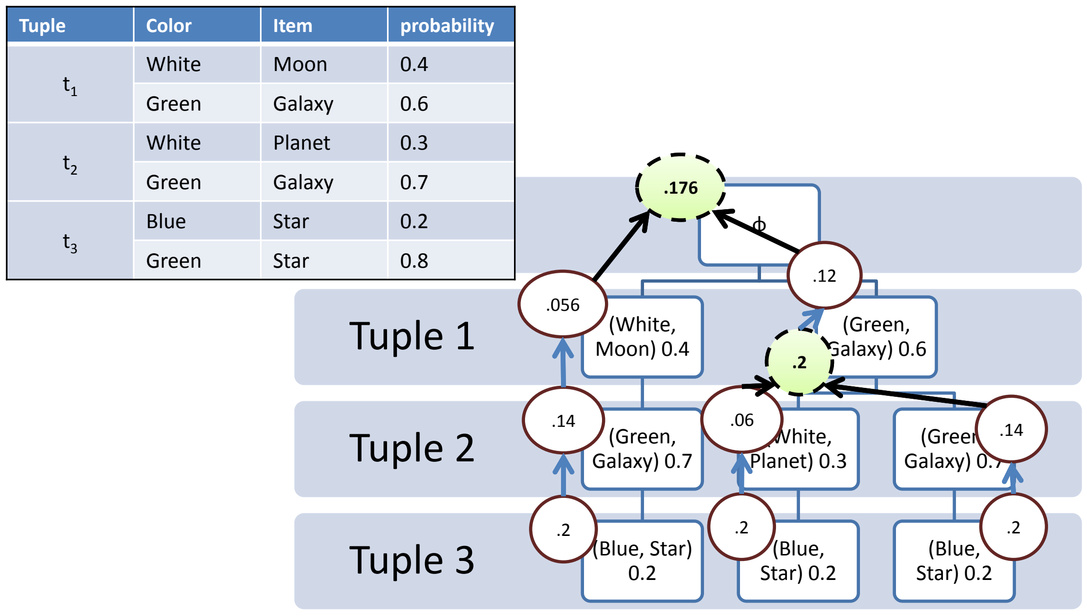
Evaluating confidence: Given a relation R, and a specified dependency find the confidence of the dependency.

Mining dependencies: Given a relation R, find a minimal set of dependencies that is equivalent to or more general[1]than any set of dependencies that holds over R with a confidence higher than a given threshold.

In the following sections I focus on evaluating the confidence of the dependencies, since it is the first step towards mining them. Once I have fast algorithms for evaluating the confidence, I then use methods in (Wolf *et al.*, 2009b) to prune the space of dependencies I have to search through in order to mine them.

While I am interested in computing the confidence for any probabilistic database, I shall show that in the very general case, evaluation is exponential. So I also consider special cases, mainly focusing on tuple-disjoint independent (TDI) databases (Dalvi and Suciu, 2007b), which are a popular special case of a probabilistic database, in which every tuple with distinct keys is independent. I can think of this as a set of uncertain relations, where each tuple has a set of "options" with each option having a probability. The decision about which option to pick for each tuple is taken independently. This significantly reduces the types of uncertainty and correlations that can occur among the tuples, but also makes many operations on the database tractable. These algorithms also work for Tuple-independent databases (TI), where each tuple has an existential probability, but does not have any options. The straightforward

[1]A set of dependencies Σ_1 is said to be more general than another set Σ_2 if every dependency in Σ_2 can be inferred from Σ_1 using Armstrong's axioms and the inference rules for conditional dependencies in (Bohannon *et al.*, 2007).



Tuple	Color	Item	probability
t_1	White	Moon	0.4
	Green	Galaxy	0.6
t_2	White	Planet	0.3
	Green	Galaxy	0.7
t_3	Blue	Star	0.2
	Green	Star	0.8

Figure 10.3: The algorithm for computing the confidence of pFD. The dotted circles represent adding the probabilities from the child branches, the solid circles represent multiplication of probabilities of the child and the parent.

adaptation of these algorithms to TI databases is explained in Section 10.5.6.

10.5.1 Assessing the Confidence of a pFD

In a generalized probabilistic database, represented by its possible worlds, finding the pFD would be polynomial in the combined size of the possible worlds, which is typically exponential in the number of entities it represents. For a more compact representation of a probabilistic database like TDI or TI, a naïve evaluation of the confidence of a pFD in a probabilistic database would likely take an exponential time in the number of tuples, since I would have to effectively generate the possible worlds and add up the probability of those in which the corresponding FD holds. Ordinarily, I would use Monte Carlo to sample the possible worlds (such as I will employ later to find the confidence of pAFDs), however, that approach does not work very well for pFD, since a single option that violates the dependency can bring the contribution of the entire sample to zero.

I now present an efficient algorithm that finds the confidence of a pFD in a tuple-disjoint independent database. This algorithm is exact and has the property of being exponential only in the *cardinality of the domain* of the attributes, rather than the number of tuples. In practice our algorithm finds useful probabilistic functional dependencies very efficiently, as most desirable pFDs have low specificity. Before I present the algortihm, I briefly introduce the concept of specificity and show why a low specificity pFD is more interesting.

10.5.2 Adapting Specificity for Probabilistic Databases

In this section I will first introduce the notion of specificity as described by Kalavagattu and Wolf *et al.* (Kalavagattu, 2008; Wolf *et al.*, 2009b) for deterministic databases and then show how it is adapted to probabilistic databases.

Deterministic databases: The distribution of values for the determining set is an important measure to judge the "usefulness" of an AFD. For an AFD $X \rightsquigarrow A$, having fewer distinct values of X means that there exist more tuples in the database that have the same values of X. This makes the AFD potentially more relevant. This is because if every value of X is distinct (i.e. it is a key) then the AFD trivially holds; however if the dependency holds in spite of X having only a few distinct values, the AFD has a deeper semantic meaning.

To quantify this, I first define the *support of a value* α_i of an attribute set X, $\mathit{support}(\alpha_i)$, as the occurrence frequency of value α_i in the training set. The support is defined as $\mathit{support}(\alpha_i) = \mathit{count}(\alpha_i)/N$, where N is the number of tuples in the training set.

Now I measure how the values of an attribute set X are distributed using *specificity*. Specificity is defined as the information entropy of the set of all possible values of attribute set X: $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, normalized by the maximal possible entropy (which is achieved when X is a key). Thus, specificity is a value that lies between 0 and 1.

$$\textit{specificity}\ (X) = \frac{-\sum_1^m \textit{support}(\alpha_i) \times \log_2(\textit{support}(\alpha_i))}{\log_2(N)}$$

When there is only one possible value of X, then this value has the maximum support and is the least specific, thus I have specificity equals to 0. When there are many distinct values in X, each having a low support and are specific, I have a high value of specificity. When all values of X are distinct (when X is a key), each value has the minimum support and is most specific and has specificity equal to 1.

Now I overload the concept of specificity on AFDs. The specificity of an AFD is defined as the specificity of its determining set. i.e. $\textit{specificity}\ (X \rightsquigarrow A) = \textit{specificity}\ (X)$. The lower specificity of an AFD, potentially the more relevant possible answers can be retrieved using the rewritten queries generated by this AFD, and thus a higher recall for a given number of rewritten queries.

Intuitively, specificity increases when the number of distinct values for a set of attributes increases. Consider two attribute sets X and Y such that Y⊃X. Since Y has more attributes than X, the number of distinct values of Y is no less than that of X, specificity(Y) is no less than specificity(X).

Probabilistic Databases: In a probabilistic database, the specificity of an attribute set X would be defined as the weighted average of the specificity of X in each possible world. Computing this is potentially exponential in the number of tuples, since every possible world will have a different set of association rules with different support.

In this chapter, I am using specificity to prune our search space. I need to be able to compute the specificity very quickly so that I do not spend too much time deciding whether or not to prune the current subspace of dependencies. As a result, I decide to approximate the computation of specificity by using a method similar to the union method described in Section 10.5.3. I ignore the intra-tuple correlations, and create a TI database by taking the union of all the options in our TDI database. Computing the specificity of a TI database is a straightforward adaptation of the deterministic algorithm. The definition for specificity(X) remains the same, but I redefine *support*(α_i) as (where t represents all the tuples in the TI database):

$$support(\alpha_i) = \sum_{\alpha_i \in t} prob(t) / \sum prob(t)$$

The complexity of the algorithm can be analyzed in terms of specificity is defined as the support of the association rules that make up the dependency (Kalavagattu, 2008). Specificity is high when the association rules have a very low support, and the rule becomes less valuable. For example, a rule with high specificity such as *(Social Security Number → Color of hair)* will definitely hold, since SSN is a key, but is not very useful. On the other hand, an FD that states *(Zip Code → Street Name)* for addresses in England is a useful one. Each zip code appears multiple times in the database and the dependency gives us useful semantic information even though zip code is not a key. For a low specificity pFD, the number of values a particular attribute can take is much less than the number of tuples, which makes our algorithm run more efficiently.

The algorithm exploits the fact that I am using a tuple-disjoint independent database. It keeps track of a set of association rules that comprise the pFD at any point in the algorithm. I pick the tuples one by one, and treat them independently. I next optimize the calculation of the pFD using two pruning criteria. First, if a particular option does not comply with the current set of rules, then the entire set of possible worlds that include that option will contribute zero confidence for the pFD. So I can terminate that branch right away. Second, all the options in the tuple comply with the ruleset, then the confidence of the pFD does not change whether or not I pick that tuple (it's contribution is 1). I can therefore ignore the tuple.

I can express the evaluation of this algorithm as a tree, see Figure 10.3. The tree branches whenever more than one option in a tuple is consistent with the current set of rules. Using the two criteria in the previous paragraph, I can choose an optimal order in which to pick the tuples so that the expression tree has the minimum width. I can then prove that the algorithm is exponential only in the cardinality of the domain of the attributes.

Once the execution reaches the leaf node, I track back. At each stage, I sum up the probability of all the branches that originated at that point. Then I multiply the result with the probability of the parent to compute the contribution of this branch to the confidence of the pFD.

The algorithm is formally presented as Algorithm 3. The running time of the algorithms is improved by the introduction of the function $ChooseBestRemainingTuple$, which picks from the remaining tuples those that do not cause branching in the evaluation tree. There are three kinds of such tuples – those that completely comply with the current set of rules (their contribution is 1), or those that have no options that comply with the current set of rules (the branch is immediately terminated), or those that have only one option that complies with the current set of rules (there is no branching).

Algorithm 3: FindPfdRecursive

```
input  : A TDI database $D$, a pFD $P$ and the current set of rules $R$
output: The confidence of $P$ in $D$
begin
    $P \longleftarrow 0$
    $T = ChooseBestRemainingTuple(D, R)$
    if $EntireTupleisCompatible(R)$ then
        return $FindPfdRecursive(D \setminus T, R)$
    end
    for $Options\ O \in T$ do
        if $IsCompatible(O,\ R)$ then
            $AddRule(O, R)$
            $P \longleftarrow P + Prob.(O) \times FindPfdRecursive(D \setminus T, R)$
            $RemoveFromRule(O, R)$
        end
    end
    return $P$
end
```

10.5.3 Assessing the Confidence of a pAFD

It was possible to considerably speed up the calculation of the confidence of a pFD because as soon as an execution branch violated the association rules for that branch, I was able to terminate it. However, this technique does not work for calculating the confidence of a pAFD. During the execution of a similar algorithm for pAFD, if a tuple violates the majority association rule, it does not terminate the branch – it merely reduces the confidence within that possible world. In addition to this, the association rules themselves might change once enough counterexamples are observed. As a result the algorithm becomes exponential time for a pAFD. I can, however, attempt to calculate the confidence of a pAFD approximately.

Deterministic approximation: One of the ways in which a pAFD may be approximated is to first create a deterministic database by completely ignoring the probabilities and treating every uncertain option as a tuple of a deterministic database. Obviously, this will violate the semantics of disjoint possible worlds. I then find the AFD over this deterministic database, and report the confidence of that AFD as the confidence of the pAFD.

Unioned approximation: A better approximation would be to create a tuple independent (TI) database by taking every uncertain option in the database along with its probability and making it a tuple of the TI database. This causes it to lose all the correlation between the options of the same tuple in a TDI database, and I treat them as independent. Since this is effectively creating a union of all the options, I call this approach the unioned approximation to finding the confidence of a pAFD. I then find the pAFD over this tuple-independent database. Having the tuples completely independent of each other makes finding the confidence much easier. To find the confidence of the dependency $(X \rightsquigarrow Y)$, I can find all the association rules (x, y) in the database. For each distinct value of X, I find that value y_{max} of Y which has the maximum support in the database. The pAFD value is given by $\sum support(y_{max}) / \sum support(x)$.

Monte Carlo: In order to maintain intra-tuple correlations, I can sample a subset of the possible worlds, and compute the confidence of the pAFD over that subset. I can then scale it up appropriately to find the confidence of the pAFD over the entire relation. It is clear that the more representative a subset I sample, the more accurate our pAFD computation will be. To choose a well-represented subset

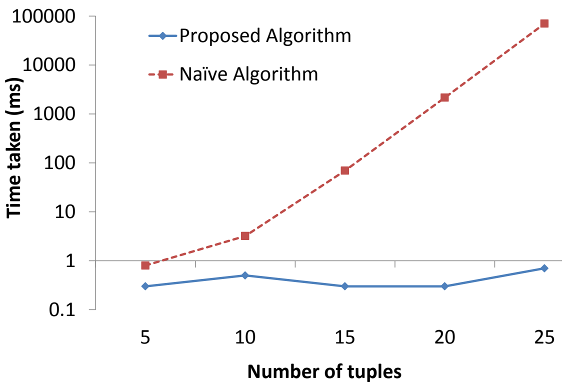


Figure 10.4: Comparison between the time taken by the naïve algorithm and the proposed algorithm for the confidence of a pFD on a log-scale.

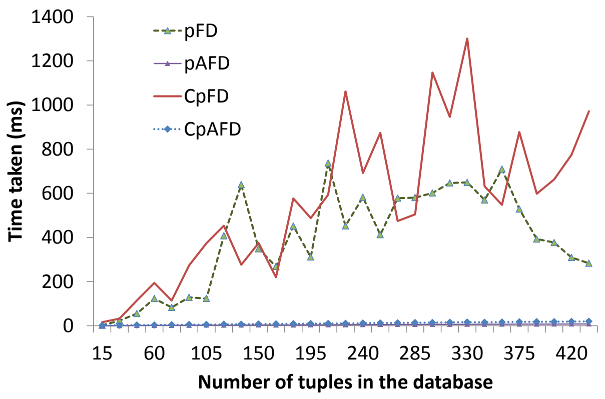


Figure 10.5: Comparison of time taken for the algorithms for various dependencies to run vs the number of tuples.

of possible worlds, I use a Monte Carlo simulation.

It is worth noting that the Monte Carlo technique does not make any assumptions about the Probabilistic Database under consideration, specifically, *it is not restricted to TDI databases*. For the case of the TDI database, I take one tuple at a time. For every tuple, I generate a random number to choose which option is to be picked. This is done proportional to the probability of the option. Since by definition, different tuples are independent of each other, this results in generating a Monte Carlo sample of the TDI database. I find the AFD confidence of the sampled possible world, and weigh it with the probability of the possible world. I repeat this process till the the weighted average of the AFD values observed converges. That is reported as the confidence of the pAFD.

10.5.4 Assessing the Confidence of a CpFD

Our strategy to find the confidence of the conditional dependencies – the CpFD and the CpAFD – is a simple one. I first select the tuples from the database that match the pattern tableau, and then I run our corresponding pFD or pAFD on the resulting relation.

As Dalvi and Suciu show in (Dalvi and Suciu, 2007a), query processing on a probabilistic database can be a #P-hard problem. However, there are certain queries that are guaranteed to have *safe-plans* which can be evaluated in polynomial time. Fortunately, selecting the tuples matching a pattern tableau is a safe query, and can be efficiently evaluated using the algorithms in (Dalvi and Suciu, 2007a).

I follow Bohannon *et. al.* (Bohannon *et al.*, 2007) for the query to find tuples that do not match the pattern tableau, appropriately modified for a probabilistic relation. Given a probabilistic relation R with attributes $A_1, ...A_n$, and a CpFD $(X \to Y, T_p)$, I can use the following query to find the probabilistic options of the tuples that do not match the pattern tableau T_p and hence can be removed from consideration:

$$\begin{aligned}&\textbf{select } t \textbf{ from } R\,t, T_p\,t_p\\ &\textbf{where } \text{NOT}(t[X_1] \asymp t_p[X_1]\,\text{AND}...\text{AND}\,t[X_n] \asymp t_p[X_n])\end{aligned}$$

Here, $t[X] \asymp t_p[X]$ represents the condition that either $t[X] = t_p[X]$ or $t_p[X] = \_$. I replace all these tuples with the special symbol $\circledcirc$. Then I run the following query to find all tuples that violate the pattern tableau:

$$\begin{aligned}&\textbf{select } t \textbf{ from } R\,t, T_p\,t_p\\ &\textbf{where } t[X_1] \asymp t_p[X_1]\,\text{AND}...\text{AND}\,t[X_n] \asymp t_p[X_n]\,\text{AND}\\ &\qquad (t[Y_1] \not\asymp t_p[Y_1]\,\text{OR}\,...\,\text{OR}\,t[Y_n] \not\asymp t_p[Y_n])\end{aligned}$$

Here $t[Y] \not\asymp t_p[Y]$ represents the condition that both $t[Y] \neq t_p[Y]$ and $t_p[Y] \neq \_$. I replace these options with the special symbol ϕ to denote that it violates the tableau. The main difference from (Bohannon *et al.*, 2007) in finding mismatches is that they found entire tuples, but here I find options of tuples.

I now run our algorithm of section 10.5.1 on this modified relation. Whenever I encounter the ⊚ symbol, I treat it as if the tuple does not exist. Whenever I encounter the ϕ symbol, I treat it as if it violates the current set of rules. The resulting confidence is the confidence of the CpFD.

I illustrate this process with an example. Consider the database of Figure 10.1. Consider the CpFD $(Color \rightarrow Type, T_1)$, where T_1 is the single-tuple $(Red, Star)$. I find that any tuple that does not have Red for the $Color$ attribute matches the first query and is replaced with ⊚. The only two remaining options are *(Sirius, Red, Star)* and *(Taurus, Red, Nebula).* since $Nebula \not\asymp Star$, the *(Taurus, Red, Nebula)* option matches the second query, and thus the tuple is replaced with a ϕ. I then run the pFD algorithm and obtain the confidence 0.5.

10.5.5 Assessing the Confidence of a CpAFD

I follow the same principle as Section 10.5.4. However, in this case I use the Monte Carlo algorithm of Section 10.5.3 instead of the algorithm for pFD to evaluate the confidence over the resulting relation.

10.5.6 Adapting the Algorithms for TI Databases

I can use the pFD algorithm for TI databases with a slight modification. Recall the symbol ⊚ introduced in Section 10.5.4 for handling options eliminated for not matching the pattern tableau. The symbol represents that in further computation, the option will be ignored for the purpose of computing the confidence, that is, it will not conflict with any existing rule. For every tuple in the TI database whose probability p is less than 1, I convert it into a TDI database by adding an option to it consisting of the ⊚ symbol and probability $1-p$. I then have a TDI database which is essentially equivalent to the TI database. I can now apply the pFD algorithm on this database to assess its confidence.

The union approximation for assessing the confidence of a pAFD from Section 10.5.3 can be applied directly to the TI database to get the accurate value of the pAFD.

10.5.7 Mining Dependencies

I adapt the *AFDMiner* algorithm from Wolf *et al.* (Wolf *et al.*, 2009b) to mine dependencies in our data. In this section I describe the outline of the algorithm, and the adaptations to probabilistic data.

The algorithm searches through the set-containment lattice of the attributes of the relation. This lattice consists of all possible sets of attributes. Each set of attribute has a directed edge that points to all sets that contain one attribute more than itself. The algorithm performs a breadth-first search through this lattice, starting with the null set of attributes and working its way up to the set of all attributes. For each directed edge $(X, X \cup \{A\})$ the algorithm travels along, the dependencies $(X \rightsquigarrow A)$ are tested. *AFDMiner* outputs those dependencies whose confidence is larger than the supplied confidence threshold. I adapt *AFDMiner* by supplying our own confidence assessing functions for pAFD.

Pruning: Each attribute set X is tested for its specificity value. If the value is higher than the specificity threshold, then all outgoing edges from that set are removed from the lattice. This lets the algorithm prune the space of dependencies whose body is X or its superset, since they are guaranteed to be above the specificity threshold.

AFDMiner further prunes the space of dependencies based on redundancy. For any dependencies that hold exactly, any superset of the dependency would also hold (by Armstrong's Axioms), and hence need not to be checked. So, effectively, a list of exact FDs is maintained, and any superset of the FDs in this list are not checked. In our case, the algorithm to check for a pFD is significantly more expensive than the algorithm to compute the confidence of a pAFD. As a result, I adapt this condition by replacing FDs with *very high confidence* pAFDs. For any pAFD that has a confidence larger than the preset high-confidence threshold, I prune the outgoing edges from that attribute.

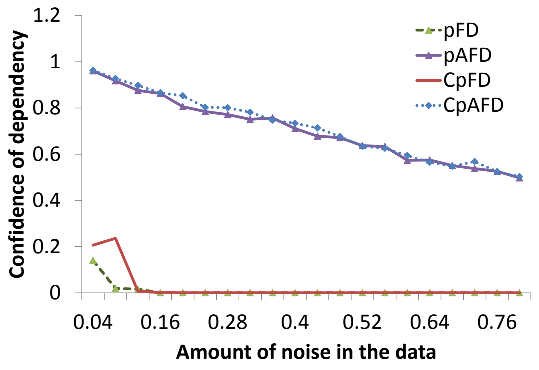


Figure 10.6: Comparison between the average confidence reported for the dependencies in a database for varying noise.

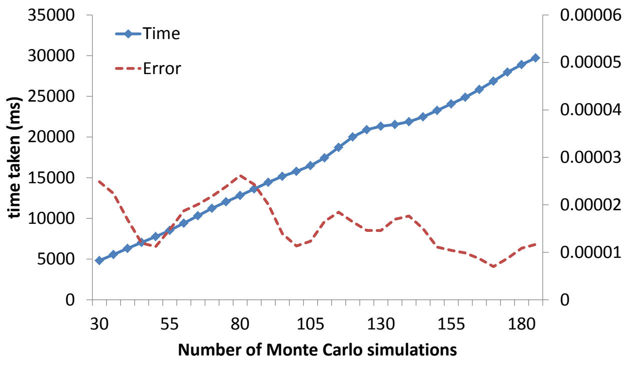


Figure 10.7: The average error and the time taken vs the number of Monte Carlo simulations for a 200,000 tuple database of DBLP data.

10.6 Experimental Evaluation

I empirically verify our algorithms using two sets of data – one generated synthetically, and the other real data extracted from DBLP.

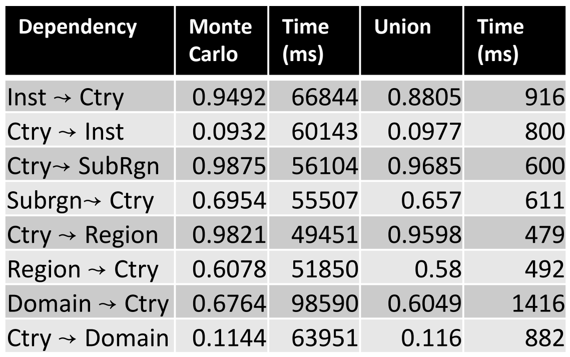
10.6.1 Synthetic Data

Data: I am using the tuple-disjoint independent model. I generate synthetic data by creating a TDI database with n tuples, each tuple having 2 options, and the options generated from the domain of the attributes of cardinality m. Each option may choose not to follow the specified FD with the probability called *noise*.

Results: I evaluate the time taken by the pFD algorithm to run on synthetic data, while varying the number of tuples in the data. I compare this performance with the naïve algorithm, which would enumerate all the possible worlds. The results are shown in Figure 10.4. As can be clearly seen from the graph, the time taken by the naïve algorithm grows exponentially, and quickly becomes infeasible to run, even with as few as 30 tuples.

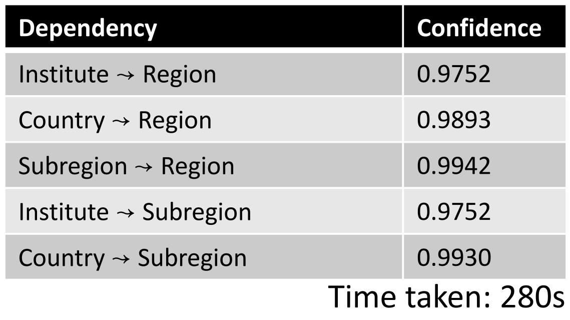
In Figure 10.5, I compare the time taken by the algorithms for the calculating the confidence of various dependencies. The cardinality of the domain of the attribute is held constant, while the number of tuples is increased. The Monte Carlo approximation algorithms for pAFD and CpAFD take significantly less time to converge compared to the pFD and CpFD algorithms. When plotted on a separate graph, it can be seen that they grow approximately linearly with the number of tuples. The pFD and CpFD show a general increasing trend with the number of tuples. The fluctuation observed is due to the algorithm quickly finding conflicting data and terminating early in some cases.

I also assessed the robustness of the dependencies to noise in the data in Figure 10.6. I observe that with slight introduction of corruption, the confidences of a pFD and CpFD drop sharply. The confidence of a pAFD does not fall too sharply, which shows that when the data is likely to be noisy, pAFD should be used to mine dependencies. However, for data rectification, or in cases where the data is guaranteed to



Dependency	Monte Carlo	Time (ms)	Union	Time (ms)
Inst → Ctry	0.9492	66844	0.8805	916
Ctry → Inst	0.0932	60143	0.0977	800
Ctry→ SubRgn	0.9875	56104	0.9685	600
Subrgn→ Ctry	0.6954	55507	0.657	611
Ctry → Region	0.9821	49451	0.9598	479
Region → Ctry	0.6078	51850	0.58	492
Domain → Ctry	0.6764	98590	0.6049	1416
Ctry → Domain	0.1144	63951	0.116	882

Figure 10.8: The confidence of pAFD and time taken as computed by Monte Carlo method vs the union method.



Dependency	Confidence
Institute → Region	0.9752
Country → Region	0.9893
Subregion → Region	0.9942
Institute → Subregion	0.9752
Country → Subregion	0.9930

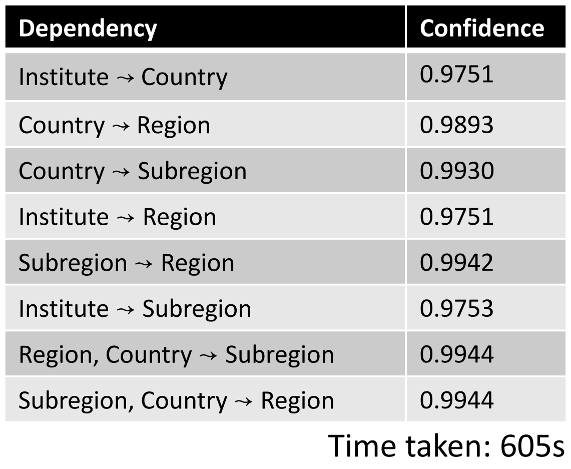
Time taken: 280s

Figure 10.9: The dependencies discovered in DBLP data by mining pAFDs for specificity threshold = 0.3.

be clean, pFD will come in very useful, since the confidence value is very sensitive.

10.6.2 DBLP Data

I use a set of data modified from (Kimura *et al.*, 2010). The database consists of DBLP (Ley, 2009) data, with additional probabilistic attributes added to it by various information retrieval and machine learning sources. I use the "Author" rela-



Dependency	Confidence
Institute → Country	0.9751
Country → Region	0.9893
Country → Subregion	0.9930
Institute → Region	0.9751
Subregion → Region	0.9942
Institute → Subregion	0.9753
Region, Country → Subregion	0.9944
Subregion, Country → Region	0.9944

Time taken: 605s

Figure 10.10: The dependencies discovered in DBLP data by mining pAFDs with specificity threshold = 0.6.

tion from this source, which contains information about approximately 700,000 computer science authors. This table has some deterministic attributes such as *Name, MinYearOfPublication, MaxYearOfPublication, NumPublication*. It also has the following uncertain attributes: *Institute, Country, Domain, Region and Subregion*. I modify this dataset by re-indexing it and converting it into a tuple-disjoint independent format.

In Figure 10.7 I show the results of running the Monte Carlo pAFD algorithm on a 200,000-tuple subset of this dataset to evaluate the confidence of the dependency $Institute \rightsquigarrow Country$. I show how the accuracy of the evaluated confidence varies with the number of Monte Carlo simulations. I see that with increase in the number of simulations, the time taken increases, and the average error decreases, as expected. I terminate the simulation once the computed confidence converges. From this graph it is apparent that the it takes only around 100 simulations before the value stabilizes and the algorithm can be terminated.

In Figure 10.8 I show the confidence values of the pAFD mined from this data for all 700,000 tuples using two different approaches. As shown in Section 10.5.3, I can approximate the the confidence of a pAFD using three different approaches - the deterministic, union and Monte Carlo approximations. This experiment shows that even the union approximation method gives significant differences in confidence values. In the context of mining the dependencies I would typically choose dependencies by placing a threshold on the confidence values or by taking the top-k mined dependencies. The difference I observed in the confidence values is significant enough that the union method would give different dependencies when mining. I also see that the unioned method can both underestimate (e.g. the first dependency in Figure 10.8) and overestimate the probabilities (e.g. the last dependency). Thus it seems that the Monte Carlo method is most suited to finding the confidence of pAFDs.

10.6.3 Dependency Mining

In order to mine pAFD dependencies, I adapted the $AFDMiner$ algorithm as described in (Wolf *et al.*, 2009b). I start with 200,000 tuples from the DBLP dataset. I choose four uncertain attributes *Institute, Country, Region and Subregion*. I build a heirarchy of possible dependencies (the set-containment lattice of the attributes). Using the $AFDMiner$ algorithm, I prune our search space using two criteria: the redundancy and specificity. The redundancy condition prunes those dependencies that are guaranteed to hold since they are subsumed by the current set of dependencies. The specificity condition prunes those dependencies that are too specific to be considered as pAFDs. For each candidate dependency that did not get pruned off, I compute the confidence using the Monte Carlo algorithm. The exact details of the adaptation is described in the Section 10.5.7.

Figures 10.9 and 10.10 show dependencies mined from the DBLP data using two different thresholds for specificity. In the first case I set a very low specificity requirement, so only those dependencies that are likely to be more general across the data are discovered. In the second case I set a high specificity threshold value, allowing much more specific dependencies to be also discovered. As I can see, this results in more dependencies being discovered, at the cost of quality of the dependencies.

10.7 Probabilistic Dependency Conclusions

In this chapter I defined a spectrum dependencies for probabilistic databases. These dependencies are logical extensions of their deterministic counterparts. I explained how these dependencies are related to each other. I showed that pAFD would always have a larger confidence than the pFD. I showed the CpAFDs were the most general of all and that it subsumed every other kind of dependency. I then presented algorithms to assess the confidence of each of these dependencies. I empirically verified the algorithms – the ones for pFD and CpFD were exponential in the number of values of the attribute, and approximately linear in the number of tuples. The Monte Carlo algorithms for the approximate dependencies converged fast and were accurate. I also showed experiments with real data that demonstrated that the Monte Carlo algorithm converges quickly. Finally I showed how I can use these algorithms to effectively mine dependencies from a real probabilistic database and discover useful dependencies. I am currently exploring the use of these dependencies in the QPIAD project (Wolf *et al.*, 2009b).

Chapter 11

CONCLUSIONS

Many recent publications have talked about the importance of big data, and how that leads to more informed solutions for everyone. However, informed decisions are only correct if the underlying data is correct. In this thesis, I showed a probabilistically principled method to perform data cleaning. Unlike other 'data cleaning' systems, BayesWipe cleans attribute values present in the data, and it does so without requiring experts to provide a curated set of rules, or a curated clean master data set.

The components of BayesWipe are flexible enough to accommodate many of the real life errors that actually occur in data, and to learn directly from moderately noisy data. Since the components are probabilistic in nature, the users can also control the fraction of tuples the system changes by changing a single parameter.

Further, BayesWipe recognizes that very often users don't have write access to the data itself, either due to access restrictions, or because the data moves so quickly that writing to database is impractical. Therefore, there are two modes of BayesWipe: (1) Offline data cleaning, an *in situ* rectification of data. There is an option to generate a standard, deterministic database as the output, as well as a probabilistic database, where all the alternatives are preserved for further processing. (2) Online query processing mode, a highly efficient way to obtain clean query results over inconsistent data.

By evaluating the output of the system over synthetic data, I showed results in a controlled environment proving both the efficacy and efficiency of the system. By performing crowdsourced experiments over real life data, I showed encouraging results of BayesWipe's performance in the real world.

I also showed a publicly available version of BayesWipe, and demonstrated how it can be run on the map-reduce architecture so that it can scale to huge data sizes.

Acknowledging that BayesWipe only considered deterministic input, I also showed how variations of functional dependencies can be learned from probabilistic data, so that they could potentially be used for data cleaning of PDB data. Further, users can grasp an intuitive understanding of the relationships between attributes from these dependencies, since the internal representation of a Bayes network is opaque to the user.

Overall, BayesWipe provides a complete, probabilistically principled, large-dataset capable data cleaning package for the modern world.

REFERENCES

"The Sloan Digital Sky Survey", URL: http://www.sdss.org/ (2013).

Agrawal, R. and R. Srikant, "Fast algorithms for mining association rules", in "VLDB", pp. 487–499 (1994).

Arenas, M., L. Bertossi and J. Chomicki, "Consistent query answers in inconsistent databases", in "PODS", pp. 68–79 (ACM, 1999).

Asuncion, A. and D. Newman, "UCI machine learning repository", (2007).

Berger, A., V. Pietra and S. Pietra, "A maximum entropy approach to natural language processing", Computational linguistics (1996).

Bertossi, L. E., S. Kolahi and L. V. S. Lakshmanan, "Data cleaning and query answering with matching dependencies and matching functions", in "ICDT", (2011).

Beskales, G., "Modeling and querying uncertainty in data cleaning", Ph.D. Thesis (2012).

Beskales, G., I. F. Ilyas, L. Golab and A. Galiullin, "On the relative trust between inconsistent data and inaccurate constraints", in "ICDE", (IEEE, 2013a).

Beskales, G., I. F. Ilyas, L. Golab and A. Galiullin, "Sampling from repairs of conditional functional dependency violations", The VLDB Journal pp. 1–26 (2013b).

Bohannon, P., W. Fan, M. Flaster and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification", in "SIGMOD", (ACM, 2005).

Bohannon, P., W. Fan, F. Geerts, X. Jia and A. Kementsietsidis, "Conditional functional dependencies for data cleaning", in "ICDE", pp. 746–755 (IEEE, 2007).

Boulos, J., N. Dalvi, B. Mandhani, S. Mathur, C. Re and D. Suciu, "Mystiq: a system for finding more answers by using probabilities", in "SIGMOD", pp. 891–893 (2005).

Brin, S., R. Motwani, J. Ullman and S. Tsur, "Dynamic itemset counting and implication rules for market basket data", in "1997 ACM SIGMOD", p. 264 (ACM, 1997).

Chiang, F. and R. Miller, "Discovering data quality rules", Proceedings of the VLDB Endowment (2008).

Computing Research Association, "Challenges and opportunities with big data", `http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf` (2012).

Cong, G., W. Fan, F. Geerts, X. Jia and S. Ma, "Improving data quality: Consistency and accuracy", in "VLDB", pp. 315–326 (VLDB Endowment, 2007).

Dalvi, N. and D. Suciu, “Efficient query evaluation on probabilistic databases”, in “VLDB”, vol. 30, pp. 864–875 (VLDB Endowment, 2004).

Dalvi, N. and D. Suciu, “Efficient query evaluation on probabilistic databases”, VLDB Journal **16**, 4, 544 (2007a).

Dalvi, N. and D. Suciu, “Management of probabilistic data: foundations and challenges”, in “PODS”, p. 12 (ACM, 2007b).

Dasu, T. and J. M. Loh, “Statistical distortion: Consequences of data cleaning”, VLDB **5**, 11, 1674–1683 (2012).

De, S., “BayesWipe Database Cleaner”, (2014).

De, S., Y. Hu, Y. Chen and S. Kambhampati, “BayesWipe: A Multimodal System for Data Cleaning and Consistent Query Answering on Structured Data.”, Big Uncertain Data (BUDA) (2014).

Dong, X. L., L. Berti-Equille and D. Srivastava, “Truth discovery and copying detection in a dynamic world”, VLDB **2**, 1, 562–573 (2009).

Fan, W., F. Geerts, X. Jia and A. Kementsietsidis, “Conditional functional dependencies for capturing data inconsistencies”, TODS **33**, 2, 6 (2008).

Fan, W., F. Geerts, L. Lakshmanan and M. Xiong, “Discovering conditional functional dependencies”, in “ICDE”, (IEEE, 2009).

Fan, W., F. Geerts, N. Tang and W. Yu, “Inferring data currency and consistency for conflict resolution”, in “ICDE”, (IEEE, 2013).

Fan, W., J. Li, S. Ma, N. Tang and W. Yu, “Towards certain fixes with editing rules and master data”, Proceedings of the VLDB Endowment (2010).

Fellegi, I. and D. Holt, “A systematic approach to automatic edit and imputation”, J. American Statistical association pp. 17–35 (1976).

Fuxman, A., E. Fazli and R. J. Miller, “Conquer: Efficient management of inconsistent databases”, in “SIGMOD”, pp. 155–166 (ACM, 2005).

Gupta, R. and S. Sarawagi, “Creating probabilistic databases from information extraction models”, in “PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES”, vol. 32, p. 965 (Citeseer, 2006).

Hartemink., A., “Banjo: Bayesian network inference with java objects.”, `http://www.cs.duke.edu/ amink/software/banjo` (????).

Huhtala, Y., J. Karkkainen, P. Porkka and H. Toivonen, “TANE: An efficient algorithm for discovering functional and approximate dependencies”, The Computer Journal **42**, 2, 100 (1999).

Jampani, R., F. Xu, M. Wu, L. Perez, C. Jermaine and P. Haas, “MCDB: a monte carlo approach to managing uncertain data”, in “SIGMOD”, pp. 687–700 (ACM, 2008).

Kalavagattu, A., *Mining Approximate Functional Dependencies as condensed representations of Association rules*, Master’s thesis, Arizona State University (2008).

Kimura, H., S. Madden and S. Zdonik, “UPI: A Primary Index for Uncertain Databases”, Proceedings of the VLDB Endowment **3**, 1 (2010).

Knorr, E., R. Ng and V. Tucakov, “Distance-based outliers: algorithms and applications”, The VLDB Journal **8**, 3, 237–253 (2000).

Koudas, N., S. Sarawagi and D. Srivastava, “Record linkage: similarity measures and algorithms”, in “SIGMOD”, pp. 802–803 (ACM, 2006).

Kubica, J. and A. Moore, “Probabilistic noise identification and data cleaning”, in “ICDM”, pp. 131–138 (IEEE, 2003).

Ley, M., “DBLP: some lessons learned”, Proceedings of the VLDB Endowment **2**, 2, 1493–1500 (2009).

Li, M., Y. Zhang, M. Zhu and M. Zhou, “Exploring distributional similarity based models for query spelling correction”, in “ICCL”, pp. 1025–1032 (Association for Computational Linguistics, 2006).

Mellott, M., “CFD-Based Data Cleaning With Probabilistic Sampling for Web Data”, in “Fulton Undergraduate Research Initiative Spring Symposium”, p. 24 (Arizona State University, 2013).

Minka, T., W. J.M., J. Guiver and D. Knowles, “Infer.NET 2.4”, Microsoft Research Cambridge. `http://research.microsoft.com/infernet` (2010).

Mo, L., R. Cheng, X. Li, D. Cheung and X. Yang, “Cleaning uncertain data for top-k queries”, in “ICDE”, (IEEE, 2013).

Papakonstantinou, Y. and V. Vassalos, “Query rewriting for semistructured data”, ACM SIGMOD Record **28**, 2, 455–466 (1999).

Pearl, J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* (Morgan Kaufmann Publishers, 1988).

Redman, T., “The impact of poor data quality on the typical enterprise”, Communications of the ACM (1998).

Ristad, E. and P. Yianilos, “Learning string-edit distance”, Pattern Analysis and Machine Intelligence, IEEE Transactions on (1998).

Sarma, A., J. Ullman and J. Widom, “Schema design for uncertain databases”, in “Proceedings of the 3rd Alberto Mendelzon Workshop on Foundations of Data Management”, (Citeseer, 2009).

Suciu, D. and N. Dalvi, “Foundations of probabilistic answers to queries”, SIGMOD **14**, 16, 963–963 (2005).

Wang, D. Z., L. Dong, A. D. Sarma, M. J. Franklin and A. Halevy, “Functional Dependency Generation and Applications in pay-as-you-go data integration systems *”, in “WebDB”, pp. 1–6 (2009).

Wolf, G., A. Kalavagattu, H. Khatri, R. Balakrishnan, B. Chokshi, J. Fan, Y. Chen and S. Kambhampati, “Query processing over incomplete autonomous databases: query rewriting using learned data dependencies”, The VLDB Journal (2009a).

Wolf, G., A. Kalavagattu, H. Khatri, R. Balakrishnan, B. Chokshi, J. Fan, Y. Chen and S. Kambhampati, “Query processing over incomplete autonomous databases: query rewriting using learned data dependencies”, The VLDB Journal **18**, 5, 1167–1190 (2009b).

Xiong, H., G. Pandey, M. Steinbach and V. Kumar, “Enhancing data analysis with noise removal”, TKDE **18**, 3, 304–319 (2006).

Yakout, M., A. K. Elmagarmid, J. Neville, M. Ouzzani and I. F. Ilyas, “Guided data repair”, VLDB **4**, 5, 279–289 (2011).