BayesWipe: A Multimodal System for Data Cleaning and Consistent Query Answering on Structured BigData

Sushovan De* Yuheng Hu* Yi Chen† Subbarao Kambhampati*

*Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85281, USA
{ sushovan, yuhenghu, rao } @asu.edu

†School of Management
New Jersey Institute of Technology
Newark, NJ 07102, USA
yi.chen@njit.edu

***Abstract*—Recent efforts in data cleaning of structured data have focused exclusively on problems like data deduplication, record matching, and data standardization; none 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 paper, we 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. We thus avoid the necessity for a domain expert or clean master data. We 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. We evaluate our methods over both synthetic and real data.**

***Keywords*-databases; web databases; data cleaning; query rewriting; uncertainty**

I. Introduction

Although data cleaning has been a long standing problem, it has become critically important again because of the increased interest in big data and web data. Most of the focus of the work on big data has been on the volume, velocity, or variety of the data; however, an important part of making big data useful is to ensure the veracity of the data. Enterprise data is known to have a typical error rate of 1–5% [1] (error rates of up to 30% have been observed). This has led to renewed interest in cleaning of big data sources, where manual data cleansing tasks are seen as prohibitively expensive and time-consuming [2], or the data has been generated by users and cannot be implicitly trusted [3]. Among the various types of big data, the need to efficiently handle large scaled structured data that is rife with inconsistency and incompleteness is also more significant than ever. Indeed, multiple studies, such as [4] emphasize the importance of effective, efficient methods for handling "dirty big data".

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. Traditional methods include outlier detection [5], noise removal [6], entity resolution [7], [6], and imputation [8]. 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., [9], [10], [11]) attempt to clean data by exploiting patterns in the data, which they express in the form of conditional functional dependencies (or CFDs). 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 [12] 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, our experimental evaluation shows that even small amounts of noise severely impairs the ability to learn useful constraints from the data.

To avoid dependence on clean master data, in this paper, we propose a novel system called BayesWipe[1] that assumes that a statistical process underlies the generation of clean data (called the *data source model*) as well as the corruption of data (which we call the data *error model*). The noisy data itself is used to learn the generative and error model parameters, 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. We model the data source model through a Bayesian network, and the error process as a mixture of error features (to handle typo-related, substitution, and omission errors). We make the assumption that errors in each attribute are generated independently.

We 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

[1]A demo of the system can be found at http://bayeswipe.sushovan.de.

data crawled from the web, or aggregated from various noisy sources.

The online query processing mode of BayesWipe is motivated by big data scenarios where, due to volume and velocity of data or access restrictions, it is impractical to create a local copy of the data and clean it offline. In such cases, the best way to obtain clean answers is to clean the resultset as we retrieve it, which also provides us the opportunity of improving the efficiency of the system, since we 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. Since write access and clean master data is rarely available in big-data scenarios, this method is particularly suitable for getting clean results.

To summarize our contributions, we:

- Propose that data cleaning should be done using a principled, probabilistic approach.
- Develop a novel algorithm following those principles, which uses a Bayes network as the generative model and maximum entropy as the error model of the data.
- Develop novel query rewriting techniques so that this algorithm can also be used in a big data scenario.
- Empirically evaluate the performance of our algorithm using both controlled and real datasets.

The rest of the paper is organized as follows. We begin by discussing the related work and then describe the architecture of BayesWipe in the next section, where we also present the overall algorithm. Section IV describes the learning phase of BayesWipe, where we find the generative and error models. Section V describes the offline cleaning mode, and the next section details the query rewriting and online data processing. We describe the results of our empirical evaluation in Section VIII, and then conclude by summarizing our contributions. Further details about BayesWipe can be found in the thesis [13].

II. Related Work

Data Cleaning: The current state of the art in data cleaning focuses on deterministic dependency relations such as FD, CFD, and INDs [14], [10], [15]. 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 our experiments show, learning the dependencies from dirty data produces very unsatisfactory results.

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 [16], [9], [17]; using a human-guided repair [18], or sampling from a space of possible repairs [19]. More recently, there has been work investigating the relative accuracy of tuple attributes

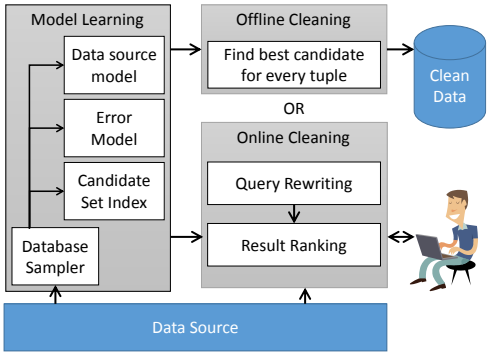


Figure 1: The architecture of BayesWipe. Our framework learns both data source model and error model from the raw data during the model learning phase. It can perform offline cleaning or query processing to provide clean data.

[20], and the relative confidence in tuple attribute values vs. the constraints that are known to hold [21], [22]. On the other hand, BayesWipe provides confidence numbers to each of the repairs, which is the posterior probability (in a Bayesian sense) of the corrected tuple given the source and error models.

To our knowledge, there is no previous work that formally proposes and learns an error model over relational data that takes care of substitutions, deletions and typos together.

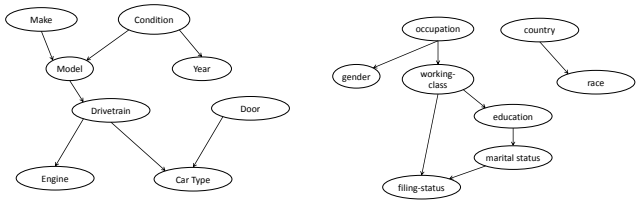
Query Rewriting

The query rewriting part of this paper is inspired by the QPIAD system [23], but significantly improves upon it. QPIAD performed query rewriting over incomplete databases using approximate functional dependencies (AFD), and only cleaned data with null values, not *wrong* values. The problem we are attempting to solve in this paper would not be solvable by QPIAD, since it assumes any values present in the database are completely correct and trustworthy.

There is recent work that performs consistent query answering over a database with primary-key constraint violations [24]. Arenas *et al.* show [16] 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 a set of *curated* deterministic integrity dependencies, and additionally, they do not use generative or error models.

III. BayesWipe Overview

BayesWipe views the data cleaning problem as a statistical inference problem over the structured text data. Let $\mathcal{D} = \{T_1, ..., T_n\}$ be the input structured data which contains a 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. Given a candidate replacement set $\mathcal{C}$ for a possibly corrupted tuple T in $\mathcal{D}$, we can clean the database



(a) Auto dataset (b) Census dataset

Figure 2: The learned Bayes networks

by replacing T with the candidate clean tuple $T^* \in \mathcal{C}$ that has the maximum $\mathbf{Pr}(T^*|T)$. Using Bayes rule (and dropping the common denominator), we can rewrite this to

$$T^*_{best} = \arg\max[\mathbf{Pr}(T|T^*)\mathbf{Pr}(T^*)] \quad (1)$$

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

$$Score(T) = \sum_{T^* \in \mathcal{C}} \underbrace{\mathbf{Pr}(T^*)}_{\text{source model}} \underbrace{\mathbf{Pr}(T|T^*)}_{\text{error model}} \underbrace{R(T^*|Q^*)}_{\text{relevance}} \quad (2)$$

While the relevance function $R(T^*|Q^*)$ can take on various forms, in this paper we use a binary relevance model — if the latent true tuple T^* matches user's query Q^*, the relevance is counted as 1, otherwise it is 0. The score is weighted by the likelihood of T^* being the actual (true) tuple.

Architecture: Figure 1 shows the system architecture for BayesWipe. During the model learning phase (Section IV), we first obtain a sample database by sending some queries to the database. On this sample data, we learn the generative model of the data as a Bayes network (Section IV-A). In parallel, we define and learn an error model which incorporates common kinds of errors (Section IV-B). We also create an index to quickly propose candidate T^*s.

We can then choose to do either offline cleaning (Section V) or online query processing (Section VI), as per the scenario. In the offline cleaning mode, we iterate over all the tuples in the database and clean them one by one. In the online query processing mode, we 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. We execute these queries and re-rank the results, and then display them.

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.

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

Algorithm 2: The 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

IV. Model Learning

This section details the process by which we estimate the components of Equation 2: the data source model $\mathbf{Pr}(T^*)$ and the error model $\mathbf{Pr}(T|T^*)$

A. Data Source Model

The data that we work with can have dependencies among various attributes (e.g., a car's *engine* depends on its *make*). Therefore, we 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 tuples.

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 us to compute probability distributions over an arbitrary input tuple T.

Whenever the underlying patterns in the source database changes, we have to learn the structure and parameters of the Bayes network again. In our scenario, we observed 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, we spend a larger amount of time learning the structure of the network with a slower, but more accurate tool, Banjo [25]. Figures 2a and 2b show automatically learned structures for two data domains. The learned structure seems to be intuitively correct, since the nodes that are connected (for example, 'country' and 'race' in Figure 2b) are expected to be highly correlated[2].

Then, given a learned graphical structure $\mathcal{G}$ of $\mathcal{D}$, we can estimate the conditional probability tables (CPTs) that parameterize each node in $\mathcal{G}$ using a faster package called Infer.NET [27]. This process of inferring the parameters is run offline, but more frequently than the structure learning.

Once the Bayesian network is constructed, we 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 $\mathcal{G}$.

B. Error Model

Having described the data source model, we now turn to the estimation of the error model $\mathbf{Pr}(T|T^*)$ from noisy data. There are many types of errors that can occur in data. We focus on the most common types of errors that occur in data that is manually entered by naïve users: typos, deletions, and substitution of one word with another. We also make an additional assumption that error in one attribute does not affect the errors in other attributes. This is a reasonable assumption to make, since we are allowing the data itself to have dependencies between attributes, while only constraining the error process to be independent across attributes. With these assumptions, we are able to come up with a simple and efficient error model, where we combine the three types of errors using a maximum entropy model.

Given a set of clean candidate tuples $\mathcal{C}$ where $T^* \in \mathcal{C}$, our error model $\mathbf{Pr}(T|T^*)$ essentially measures how clean T is, or in other words, how similar T is to T^*.

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 paper we use the Levenshtein distance, which uses a uniform cost function. This gives us a distance, which we then convert to a probability using [28]:

$$f_{ed}(T_{A_i}, T^*_{A_i}) = \exp\{-cost_{ed}(T_{A_i}, T^*_{A_i})\} \quad (3)$$

[2]Note that the direction of the arrow in a Bayes network does not necessarily determine causality, see Chapter 14 from Russel and Norvig [26].

Distributional similarity feature: This similarity measure is used to detect both substitution and omission errors. We 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 [29]. Formally, for each string T_{A_i} and $T^*_{A_i}$, we have:

$$f_{ds}(T_{A_i}, T^*_{A_i}) = \sum_{c \in C(T_{A_i}, T^*_{A_i})} \frac{\mathbf{Pr}(c|T^*_{A_i})\mathbf{Pr}(c|T_{A_i})\mathbf{Pr}(T_{A_i})}{\mathbf{Pr}(c)} \quad (4)$$

In this equation, C is the 'context', which is a set of attribute values that co-occur (appear in the same tuple) with T_{A_i} and $T^*_{A_i}$. The conditional probability $\mathbf{Pr}(c|T^*_{A_i})$ is given by $(\#(c, T^*_{A_i}) + \mu)/\#(T^*_{A_i})$, which measures the probability that the value c occurs in a tuple given the value $T^*_{A_i}$ occurs in the same tuple; with a Laplacian smoothing factor μ.

Unified error model: In practice, we do not know beforehand which kind of error has occurred for a particular attribute; we 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, we use the well-known maximum entropy framework [30] to leverage both the similarity measures, (Edit distance f_{ed} and distributional similarity f_{ds}). For each attribute of the input tuple T and T^*, we have the unified error model $\mathbf{Pr}(T|T^*)$ given by:

$$\frac{1}{Z}\exp\left\{\alpha\sum_{i=1}^{m} f_{ed}(T_{A_i}, T^*_{A_i}) + \beta\sum_{i=1}^{m} f_{ds}(T_{A_i}, T^*_{A_i})\right\} \quad (5)$$

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

C. Finding the Candidate Set

The set of candidate tuples, $\mathcal{C}(T)$ for a given tuple T are the possible replacement tuples that the system considers as possible corrections to T. The larger the set $\mathcal{C}$ is, the longer it will take for the system to perform the cleaning. If $\mathcal{C}$ contains many unclean tuples, then the system will waste time scoring tuples that are not clean to begin with.

An efficient approach to finding a reasonably clean $\mathcal{C}(T)$ is to consider the set of all the tuples in the sample database that differ from T in not more than j attributes. However, 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, we 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 these $(j+1)$ indices, since we created $j+1$ of them (pigeon hole principle). 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). This ensures that the set of tuples returned from the index would be small in number.

For every possibly dirty tuple T in the database, we 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 we can be sure that by using this method, we have obtained the entire set $\mathcal{C}$ [3].

V. Offline cleaning

A. Cleaning to a Deterministic Database

In order to clean the data *in situ*, we first use the techniques of the previous section to learn the data source model, the error model and create the index. Then, we iterate over all the tuples in the database and use Equation 1 to find the T^* with the best score. We then replace the tuple with that T^*, thus creating a deterministic database using the offline mode of BayesWipe.

1) Setting the parameter: Recall from Section IV-B that there are parameters in the error model called α and β, which need to be set. Interestingly, in addition to controlling the relative weight given to the various features in the error model, these parameters can be used to control overcorrection by the system.

Overcorrection: Any data cleaning system is vulnerable to overcorrection, where a legitimate tuple is modified by the system to an unclean value. Overcorrection can have many causes. In a traditional, deterministic system, overcorrection can be caused by erroneous rules learned from infrequent data. For example, certain makes of cars are all owned by the same conglomerate (GM owns Chevrolet). In a misguided attempt to simplify their inventory, a car salesman might list all the cars under the name of the conglomerate. This may provide enough support to learn the wrong rule (Malibu $\rightarrow$ GM).

Typically, once an erroneous rule has been learned, there is no way to correct it or ignore it without a lot of oversight from domain experts. However, BayesWipe provides a way to regulate the amount of overcorrection in the system with the help of a 'degree of change' parameter. Without loss of generality, we can rewrite Equation 5 to the following:

$$\mathbf{Pr}(T|T^*) = \frac{1}{Z}\exp\left\{\gamma\Big(\delta\sum_{i=1}^{m} f_{ed}(T_{A_i}, T^*_{A_i}) + (1-\delta)\sum_{i=1}^{m} f_{ds}(T_{A_i}, T^*_{A_i})\Big)\right\}$$

Since we are only interested in their relative weights, the parameters α and β have been replaced by δ and $(1-\delta)$ with the help of a normalization constant, γ. This parameter, γ, can be used to modify the degree of variation in $\mathbf{Pr}(T|T^*)$. High values of γ imply that small differences in T and T^* cause a larger difference in the value of $\mathbf{Pr}(T|T^*)$, causing the system to give higher scores to the original tuple (compared to a modified tuple).

[3]There is a small possibility that the true tuple T^* is not in the sample database at all. This probability can be reduced by choosing a larger sample set. In future work, we will expand the strategy of generating $\mathcal{C}$ to include all possible k-repairs of a tuple.

VI. Query rewriting for Online Query Processing

In this section we extend the techniques of the previous section so that it can be used in an online query processing method where the result tuples are cleaned at query time. 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. The process also needs to be efficient, since the time that the users are willing to wait before results are shown to them is very small. We show our query rewriting mechanisms aimed at addressing both.

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

A. Increasing the precision of rewritten queries

We 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, we 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 generated from the user's 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, we 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^*, we restrict the search to the union of all the attributes in the Markov blanket [31]. 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 1.)

Given a Q^*, we 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; we therefore approximately search for it by performing a heuristic-informed search. Our objective is to create an ESQ with m attribute-value pairs as constraints. We begin with the constraints specified by the user query Q^*. We 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. We 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 $\mathbf{Pr}(Q)^{m/p}$. 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. We expand them further, until we find k queries of size m with the highest probabilities.

B. 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, we 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 can only be retrieved by relaxed queries that do not specify a value for those attributes. Instead, we 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 choice of RQs to execute. We take the approach of generating every possible RQ, and then ranking them according to their expected relevance. This operation is performed entirely on the learned error statistics, and is thus very fast.

We score each relaxed query by 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 we do not have complete information about the tuples that will be returned for any query q. The best we can do, therefore, is to approximate this quantity.

Let the relaxed query be Q, and the expanded query that it was relaxed from be ESQ. We wish to estimate $\mathbb{E}[P(T|T^*)]$ where T are the tuples returned by Q. Using the attribute-error independence assumption, we can rewrite that as $\prod_{i=0}^{m} \mathbf{Pr}(T.A_i|T^*.A_i)$, 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, we divide the m attributes of the database into 3 classes: (1) The attribute is specified both in ESQ and in Q. In this case, we set $\mathbf{Pr}(T.A_i|T^*.A_i)$ to 1, since $T.A_i = T^*.A_i$. (2) The attribute is specified in ESQ but not in Q. In this case, we know what $T^*.A_i$ is, but not $T.A_i$. However, we can generate an average statistic of how often $T^*.A_i$ is erroneous by looking at our sample database. Therefore, in the offline learning stage, we precompute tables of error statistics for every T^* that appears in our sample database, and use that value. (3) The attribute is not specified in either ESQ or Q. In this case, we know neither the attribute value in T nor in T^*. We, therefore, use the average error rate of the entire attribute as the value for $\mathbf{Pr}(T.A_i|T^*.A_i)$. This statistic is also precomputed during the learning phase. This product gives the expected rank of the tuples returned by Q.

C. Terminating the process

We begin by looking at all the RQs in descending order of their rank. If the current k-th tuple in our resultset has a relevance of λ, and the estimated rank of the Q we are about to execute is $R(T_q|Q)$, then we 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 we have the true top-k resultset with a probability $\mathcal{P}$.

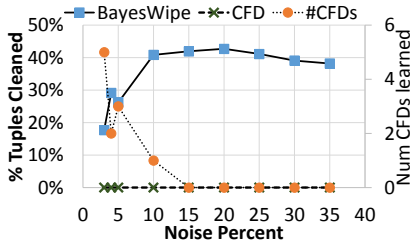
VII. Map-Reduce framework

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. Now, we show that the offline mode can also be optimized for a big-data scenario by implementing it as a Map-Reduce application.

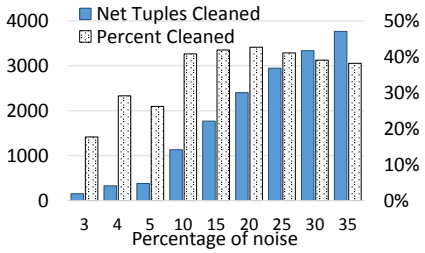
A. Simple Approach

The simplest approach to parallelizing the tuples is to run the first phase (the learning phase) on a single 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.

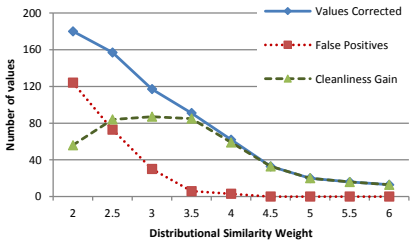
The problem with this approach is that in a truly big data scenario, the candidate index can become very large. Indeed, as the number of tuples increases, the size of the domain of each attribute also increases (see Figure 4a for 1 shard). Further, the number of different combinations, and the number of erroneous values for each attribute also increase (Figure 4b). All of this results in a rather large candidate index. Transmitting and using the entire index on



(a) % performance of BayesWipe compared to CFD, for the used-car dataset.



(b) % net corrupt values cleaned, car database



(c) Net corrections vs γ. (The x-axis values show the un-normalized distributional similarity weight, which is simply $\gamma \times 3/5$.)

Figure 3: Offline cleaning of BayesWipe

each mapper node is wasteful of both network, memory, (and if swapped out, disk resources).

B. Improved Approach

In order to split both the input tuples and the candidate index, we use a two-stage approach. In the first stage, we run a map-reduce that splits the problem into multiple shards, each shard having a small fraction of the candidate index. The second stage is a simple map-reduce that picks the best output from stage 1 for each input tuple.

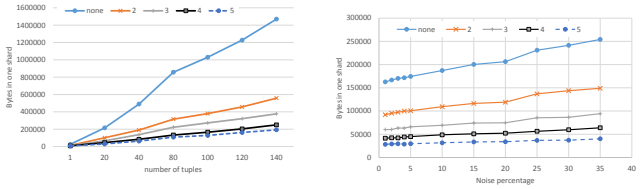
Stage 1: Given an input tuple T and a set of candidate tuples, the T^*s, suppose the candidate index is created on k attributes, $A_1 ... A_k$. We can say that for every tuple T, and one of its candidate tuples T^*, they will have at least one matching attribute a_i from this set. We can use this common element a_i to predict which shards the candidate T^*s might be available in. We therefore, send the tuple T to each shard that matches the hash of the value a_i.

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

C. Results of This Strategy

In Figure 4a and Figure 4b we can see how this map reduce strategy helps in reducing the memory footprint of the reducer. First, we 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



(a) vs the Number of Tuples (in Thousands) in the Dataset, for Various Number of Shards.

(b) vs the Noise in the Dataset, for Various Number of Shards.

Figure 4: Map-Reduce index sizes

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 4b), 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.

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.

VIII. Empirical Evaluation

We quantitatively study the performance of BayesWipe in both its modes — offline, and online, and compare it

against state-of-the-art CFD approaches. We used three real datasets spanning two domains: used car data, and census data. Among the used car datasets, we used a real dataset crawled from cars.com to validate the error model against real world noisy data. We also used a dataset with real values from Google Base with synthetic noise to measure how the precision and recall changes as the parameters of the data and system are changed. The census dataset was obtained from the UCI machine learning database [32]. We present experiments on evaluating the approach in terms of the effectiveness of data cleaning, efficiency and precision of query rewriting.

A demo for the offline cleaning mode of BayesWipe can be downloaded from http://bayeswipe.sushovan.de/.

A. Experiments

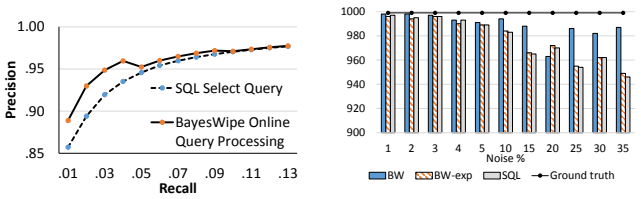
Offline Cleaning Evaluation: The first set of evaluations shows the effectiveness of the offline cleaning mode. In Figure 3a, we compare BayesWipe against CFDs [33].

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 incorrect values. It is interesting to note that the percentage of tuples cleaned increases initially and then slowly decreases, because for very low values of noise, there aren't enough errors 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 3a showed only percentages, in Figure 3b we 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 γ: As explained in Section V-A1, the weight given to the edit distance (δ) compared to the weight given to the distributional similarity ($1-\delta$); and the overcorrection parameter (γ) are parameters that can be tuned, and should be set based on which kind of error is more likely to occur. In our experiments, we performed a grid search to determine the best values of δ and γ to use. In Figure 3c, we show a portion of the grid search where $\delta = 2/5$, and varying γ.

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, our 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 γ 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.



(a) Average precision vs recall

(b) Net improvement in data quality (TP-FP)

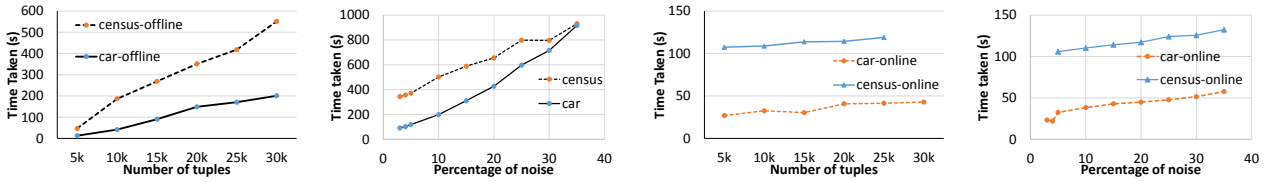
Figure 5: Online mode of BayesWipe

If we set the parameter values too low, we will correct most wrong tuples in the input dataset, but we 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 'overcorrections' will also be lower. Based on these experiments, we picked a parameter value of $\delta = 0.638, \gamma = 5.8$ and kept it constant throughout.

Online Query Processing: While in the offline mode, we had the luxury of changing the tuples in the database itself, in online query processing, we use query rewriting to obtain a resultset that is similar to the offline results, without modification to the database. We consider a SQL select query system as our baseline. We evaluate the precision and recall of our method against the ground truth and compare it with the baseline, using randomly generated queries.

We issued randomly generated queries to both BayesWipe and the baseline system. Figure 5a shows the average precision over 10 queries at various recall values. It shows that our system outperforms the SQL select query system in top-k precision, especially since our system considers the relevance of the results when ranking them. On the other hand, the SQL 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 less precision.

Figure 5b 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 SQL query system at nearly every level of noise. Further, the graph also illustrates that — compared to a SQL 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



(a) Time taken vs number of tuples in offline mode

(b) Time taken vs noise percentage in offline mode

(c) Time taken vs number of tuples in online mode

(d) Time taken vs noise percentage in online mode

Figure 6: Performance evaluations

the SQL query baseline, we also show the performance of BayesWipe without the query relaxation part (called BW-exp[4]). 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.

This shows that our 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.

Efficiency: In Figure 6 we show the time taken by the system in its various modes. The first two graphs show the offline mode, and the second two show the online mode. As can be seen from the graphs, BayesWipe performs reasonably well both in datasets of large size and datasets with large noise. The offline modes show that the time taken increases as

Evaluation on real data with naturally occurring errors: In this section we used a dataset of 1.2 million tuples crawled from the cars.com website[5] to check 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, we do not have ground truth for this data. To evaluate this system, we conducted an experiment on Amazon Mechanical Turk. First, we ran the offline mode of BayesWipe on the entire database. We 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. We inserted 3 known answers into the questionnaire, and removed any responses that failed to annotate at least 2 out of the 3 answers correctly.

An example is shown in Figure 7. 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.

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

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

Confidence	BayesWipe	Original
High confidence only	56.3%	43.6%
All confidence values	53.3%	46.7%

Table I: 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.

...	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: A fragment of the questionnaire provided to the Mechanical Turk workers.

The results of this experiment are shown in Table I. As we can see, 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. An algorithm that picks a possible correction randomly from this space is likely to get a near 0% accuracy.

The first row of Table I 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.

IX. Conclusion

In this paper we presented a novel system, BayesWipe that works using end-to-end probabilistic semantics, and without access to clean master data. We showed how to

effectively learn the data source model as a Bayes network, and how to model the error as a mixture of error features. We showed the operation of this system in two modalities: (1) offline data cleaning, an *in situ* rectification of data and (2) online query processing mode, a highly efficient way to obtain clean query results over inconsistent data. We empirically showed that BayesWipe outperformed existing baseline techniques in quality of results, and was highly efficient. We also showed the performance of the BayesWipe system at various stages of the query rewriting operation. User experiments showed that the system is useful in cleaning real-world noisy data. In future work, we will implement the system on a distributed architecture (like map-reduce), since many of the algorithms are easily parallelizable.

ACKNOWLEDGMENTS

We thank Dr. K. Selçuk Candan for his valuable advice, and Preet Inder Singh Rihan for his help with probabilistic databases. This research is supported in part by the ONR grants N00014-13-1-0176, N0014-13-1-0519, ARO grant W911NF-13-1-0023 and a Google Research Grant.

REFERENCES

[1] W. Fan and F. Geerts, "Foundations of data quality management," *Synthesis Lectures on Data Management*, vol. 4, no. 5, pp. 1–217, 2012.

[2] P. Gray, "Before Big Data, clean data," 2013. [Online]. Available: http://www.techrepublic.com/blog/big-data-analytics/before-big-data-clean-data/

[3] H. Leslie, "Health data quality – a two-edged sword," 2010. [Online]. Available: http://omowizard.wordpress.com/2010/02/21/health-data-quality-a-two-edged-sword/

[4] Computing Research Association, "Challenges and opportunities with big data," http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf, 2012.

[5] E. Knorr, R. Ng, and V. Tucakov, "Distance-based outliers: algorithms and applications," *The VLDB Journal*, vol. 8, no. 3, pp. 237–253, 2000.

[6] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar, "Enhancing data analysis with noise removal," *TKDE*, vol. 18, no. 3, pp. 304–319, 2006.

[7] P. Singla and P. Domingos, "Entity resolution with markov logic," in *ICDM*. IEEE, 2006, pp. 572–582.

[8] I. Fellegi and D. Holt, "A systematic approach to automatic edit and imputation," *J. American Statistical association*, pp. 17–35, 1976.

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

[10] W. Fan, F. Geerts, L. Lakshmanan, and M. Xiong, "Discovering conditional functional dependencies," in *ICDE*. IEEE, 2009.

[11] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan, "Data cleaning and query answering with matching dependencies and matching functions," in *ICDT*, 2011.

[12] A. Fuxman, E. Fazli, and R. J. Miller, "Conquer: Efficient management of inconsistent databases," in *SIGMOD*. ACM, 2005, pp. 155–166.

[13] S. De, "Unsupervised bayesian data cleaning techniques for structured data," Ph.D. dissertation, ARIZONA STATE UNIVERSITY, 2014.

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

[15] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang, "Nadeef: A commodity data cleaning system," *SIGMOD*, 2013.

[16] M. Arenas, L. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in *PODS*. ACM, 1999, pp. 68–79.

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

[18] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas, "Guided data repair," *VLDB*, vol. 4, no. 5, pp. 279–289, 2011.

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

[20] Y. Cao, W. Fan, and W. Yu, "Determining the relative accuracy of attributes," in *SIGMOD*. New York, NY, USA: ACM, 2013, pp. 565–576. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465309

[21] F. Chiang and R. J. Miller, "A unified model for data and constraint repair," in *ICDE*. IEEE, 2011, pp. 446–457.

[22] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin, "On the relative trust between inconsistent data and inaccurate constraints," in *ICDE*. IEEE, 2013.

[23] G. Wolf, 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*, 2009.

[24] P. G. Kolaitis, E. Pema, and W.-C. Tan, "Efficient querying of inconsistent databases with binary integer programming," *Proceedings of VLDB*, vol. 6, no. 6, 2013.

[25] A. Hartemink., "Banjo: Bayesian network inference with java objects." http://www.cs.duke.edu/ amink/software/banjo.

[26] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Prentice Hall, 2010.

[27] T. Minka, W. J.M., J. Guiver, and D. Knowles, "Infer.NET 2.4," 2010, microsoft Research Cambridge. http://research.microsoft.com/infernet.

[28] E. Ristad and P. Yianilos, "Learning string-edit distance," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 1998.

[29] M. Li, Y. Zhang, M. Zhu, and M. Zhou, "Exploring distributional similarity based models for query spelling correction," in *ICCL*. Association for Computational Linguistics, 2006, pp. 1025–1032.

[30] A. Berger, V. Pietra, and S. Pietra, "A maximum entropy approach to natural language processing," *Computational linguistics*, 1996.

[31] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, 1988.

[32] A. Asuncion and D. Newman, "UCI machine learning repository," 2007.

[33] F. Chiang and R. Miller, "Discovering data quality rules," *Proceedings of the VLDB Endowment*, 2008.