HoloClean: Holistic Data Repairs with Probabilistic Inference

Theodoros Rekatsinas*, Xu Chu†, Ihab F. Ilyas†, Christopher Ré *
{thodrek, chrismre}@cs.stanford.edu, {x4chu, ilyas}@uwaterloo.ca
* Stanford University and † University of Waterloo

ABSTRACT

We introduce HoloClean, a framework for holistic data repairing driven by probabilistic inference. HoloClean unifies qualitative data repairing, which relies on integrity constraints or external data sources, with quantitative data repairing methods, which leverage statistical properties of the input data. Given an inconsistent dataset as input, HoloClean automatically generates a probabilistic program that performs data repairing. Inspired by recent theoretical advances in probabilistic inference, we introduce a series of optimizations which ensure that inference over HoloClean's probabilistic model scales to instances with millions of tuples. We show that HoloClean finds data repairs with an average precision of $\sim 90\%$ and an average recall of above $\sim 76\%$ across a diverse array of datasets exhibiting different types of errors. This yields an average F1 improvement of more than $2\times$ against state-of-the-art methods.

1. INTRODUCTION

The process of ensuring that data adheres to desirable quality and integrity constraints (ICs), referred to as *data cleaning*, is a major challenge in most data-driven applications. Given the variety and voluminous information involved in modern analytics, large-scale data cleaning has re-emerged as the key goal of many academic [11, 29, 31] and industrial efforts (including Tamr [47], Trifacta Wrangler [33], and many more). Data cleaning can be separated in two tasks: (i) *error detection*, where data inconsistencies such as duplicate data, integrity constraint violations, and incorrect or missing data values are identified, and (ii) *data repairing*, which involves updating the available data to remove any detected errors. Significant efforts have been made to automate both tasks, and several surveys summarize these results [23, 31, 42]. For error detection, many methods rely on violations of integrity constraints [11, 14] or duplicate [27, 37, 40] and outlier detection [19, 29] methods to identify errors. For data repairing, state-of-the-art methods use a variety of signals: (i) integrity constraints [9, 15], (ii) external information [16, 24], such as dictionaries, knowledge bases, and annotations by experts, or (iii) quantitative statistics [39, 49].

While automatic error detection methods were shown to achieve precision and recall greater than 0.6 and 0.8 for multiple real-world datasets [4], this is not the case with automatic data repairing [30]. We evaluated state-of-the-art repairing methods [15, 16, 49] on different real-world datasets (Section 6) and found that (i) their average F1-score (i.e., the harmonic mean of precision and recall) across datasets is below 0.35, and (ii) in many cases these methods did not perform any correct repairs. This is because these methods limit themselves to only one of the aforementioned signals, and ignore additional information that is useful for data repairing. We show that if we combine these signals in a unified framework, we obtain data repairs with an average F1-score of more than 0.8. We use a real-world dataset to demonstrate the limitations of existing data repairing methods and motivate our approach.

Example 1. *We consider a dataset from the City of Chicago [1] with information on inspections of food establishments. A snippet is shown in Figure 1(A). The dataset is populated by transcribing forms filled out by city inspectors, and as such, contains multiple errors. Records can contain misspelled entries, report contradicting zip codes, and use different names for the same establishment.*

In our example we have access to a set of functional dependencies (see Figure 1(B)) and an external dictionary of address listings in Chicago (Figure 1(D)). Co-occurrence statistics can also be obtained by analyzing the original input dataset in Figure 1(A).

First, we focus on data repairing methods that rely on integrity constraints [7, 11, 15]. These methods assume the majority of input data to be clean and use the principle of *minimality* [5, 13, 21] as an operational principle to perform repairs. The goal is to update the input dataset such that no integrity constraints are violated. Informally, minimality states that given two candidate sets of repairs, the one with fewer changes with respect to the original data is preferable. Nevertheless, minimal repairs do not necessarily correspond to correct repairs: An example minimal repair is shown in Figure 1(E). This repair chooses to update the zip code of tuple $t1$ so that all functional dependencies in Figure 1(B) are satisfied. This particular repair introduces an error as the updated zip code is wrong. This approach also fails to repair the zip code of tuples $t2$ and $t3$ as well as the "DBAName" and "City" fields of tuple $t4$ since altering those leads to a non-minimal repair.

Second, methods that rely on external data [16, 24] match records of the original dataset to records in the external dictionaries or knowledge bases to detect and repair errors in the former. The matching process is usually described via a collection of *matching dependencies* (see Figure 1(C)) between the original dataset and external information. A repair using such methods is shown in Figure 1(F). This repair fixes most errors but fails to repair the "DBAName" field of tuple $t4$ as no information for this field is provided in the external data. In general, the quality of repairs performed by methods that use external data can be poor due to the

Proceedings of the VLDB Endowment, Vol. 10, No. 11

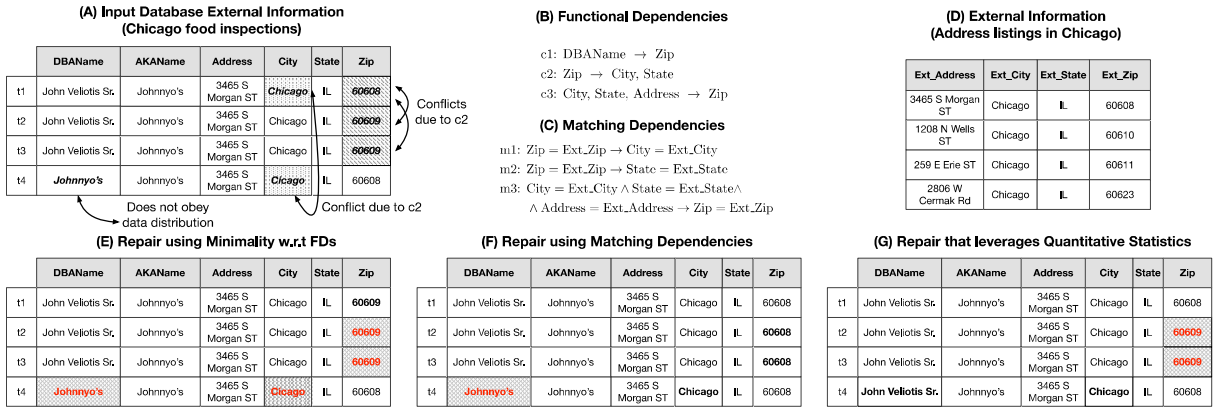

Figure 1: A variety of signals can be used for data cleaning: integrity constraints, external dictionaries, and quantitative statistics of the input dataset. Using each signal in isolation can lead to repairs that do not fix all errors or even introduce new errors.

limited coverage of external resources or these methods may not be applicable as for many domains a knowledge base may not exist.

Finally, data repairing methods that are based on statistical analysis [39, 49], leverage quantitative statistics of the input dataset, e.g., co-occurrences of attribute values, and use those for cleaning. These techniques overlook integrity constraints. Figure 1(G) shows such a repair. As shown the "DBAName" and "City" fields of tuple $t4$ are updated as their original values correspond to outliers with respect to other tuples in the dataset. However, this repair does not have sufficient information to fix the zip code of tuples $t2$ and $t3$.

In our example, if we combine repairs that are based on different signals, we can repair all errors in the input dataset correctly. If we combine the zip code and city repairs from Figure 1(F) with the DBAName repair from Figure 1(G) we can repair all inaccuracies in the input dataset. Nonetheless, combining heterogeneous signals can be challenging. This is not only because each type of signal is associated with different operations over the input data (e.g., integrity constraints require reasoning about the satisfiability of constraints while external information requires efficient matching procedures) but different signals may suggest conflicting repairs. For instance, if we naively combine the repairs in Figure 1 we end up with conflicts on the zip code of tuples $t2$ and $t3$. The repairs in Figure 1(E) and (G) assign value "60609" while the repair in Figure 1(F) assigns value "60608". This raises the main question we answer in this paper: *How can we combine all aforementioned signals in a single unified data cleaning framework, and which signals are useful for repairing different records in an input dataset?*

Our Approach. We introduce HoloClean, the first data cleaning system that unifies integrity constraints, external data, and quantitative statistics, to repair errors in structured data sets. Instead of considering each signal in isolation, we use all available signals to suggest data repairs. We consider the input dataset as a *noisy version* of a hidden clean dataset and treat each signal as *evidence* on the correctness of different records in that dataset. To combine different signals, we rely on probability theory as it allows us to reason about inconsistencies across those.

HoloClean automatically generates a probabilistic model [35] whose random variables capture the uncertainty over records in the input dataset. Signals are converted to features of the graphical model and are used to describe the distribution characterizing the input dataset. To repair errors, HoloClean uses statistical learning and probabilistic inference over the generated model.

HoloClean exhibits significant improvements over state-of-the-art data cleaning methods: we show that across multiple datasets HoloClean finds repairs with an average precision of $\sim 90\%$ and an average recall of $\sim 76\%$, obtaining an average F1-score improvement of more than $2\times$ against state-of-the-art data repairing methods. Specifically, we find that combining all signals yields an F1-score improvement of $2.7\times$ against methods that only use integrity constraints, an improvement of $2.81\times$ against methods that only leverage external information, and an improvement of $2.29\times$ against methods that only use quantitative statistics.

Technical Challenges. Probabilistic models provide a means for unifying all signals. However, it is unclear that inference scales to large data repairing instances. Probabilistic inference involves two tasks: (i) *grounding*, which enumerates all possible interactions between correlated random variables to materialize a *factor graph* that represents the joint distribution over all variables, and (ii) *inference* where the goal is to compute the marginal probability for every random variable. These tasks are standard but non-trivial:

(1) Integrity constraints that span multiple attributes can cause combinatorial explosion problems. Grounding the interactions due to integrity constraints requires considering all value combinations that attributes of erroneous tuples can take. If attributes are allowed to obtain values from large domains, inference can become intractable. For example, we consider repairing the smallest dataset in our experiments, which contains 1,000 tuples, and allow attributes in erroneous tuples to obtain any value from the set of consistent assignments present in the dataset. Inference over the resulting probabilistic model does not terminate after an entire day. Thus, we need mechanisms that limit the possible value assignments for records that need to be repaired by HoloClean's probabilistic model.

(2) Integrity constraints introduce correlations between pairs of random variables associated with tuples in the input dataset. Enumerating these interactions during grounding results in factor graphs of quadratic size in the number of tuples. For example, in our experiments we consider a dataset with more than two million tuples. Enforcing the integrity constraints over all pairs of tuples, yields a factor graph with more than four trillion interactions across random variables. Thus, we need to avoid evaluating integrity constraints for pairs of tuples that cannot result in violations.

(3) Finally, probabilistic inference is #P-complete in the presence of complex correlations, such as hard constraints. Thus, approximate inference techniques such as Gibbs sampling are required. In the presence of complex correlations, Gibbs sampling is known to require an exponential number of samples in the number of random variables to *mix* [45], i.e., reach a stationary distribution. Neverthe-

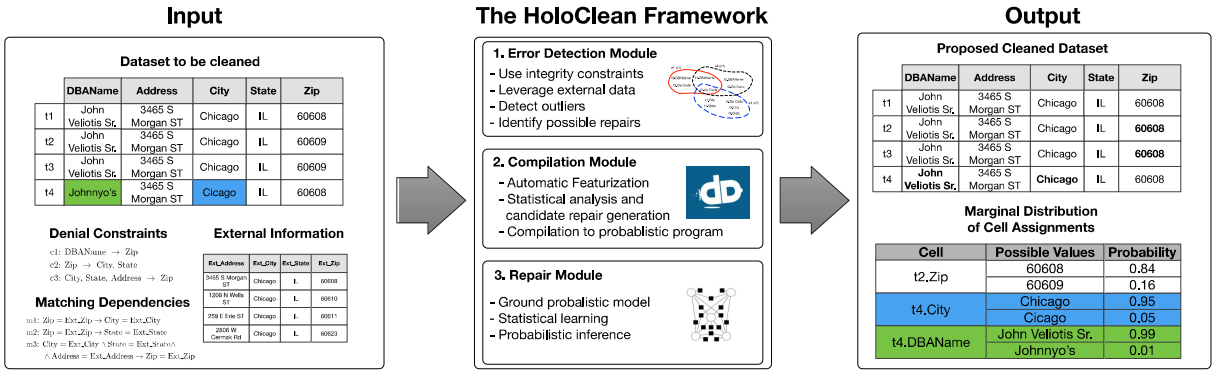

Figure 2: An overview of HoloClean. The provided dataset along with a set of denial constraints is compiled into a declarative program which generates the probabilistic model used to solve data repairing via statistical learning and probabilistic inference.

less, recent theoretical advancements [12, 45] in statistical learning and inference show that relaxing hard constraints to soft constraints introduces a tradeoff between the computational efficiency and the quality of solutions obtained. This raises the technical question, how to relax hard integrity constraints for scalable inference.

Technical Contributions. Our main technical contributions are:

(1) We design a compiler that generates a probabilistic model which unifies different signals for repairing a dataset (see Sections 2 and 4). Our compiler supports different error detection methods, such as constraint violation detection and outlier detection.

(2) We design an algorithm that uses Bayesian analysis to prune the domain of the random variables corresponding to noisy cells in the input dataset (see Section 5.1.1). This algorithm allows us to systematically tradeoff the scalability of HoloClean and the quality of repairs obtained by it. We also introduce a scheme that partitions the input dataset into non-overlapping groups of tuples and enumerates the correlations introduced by integrity constraints only for tuples within the same group (see Section 5.1.2). These two optimizations allow HoloClean to scale to data cleaning instances with millions of tuples. We study the synergistic effect of these optimizations on HoloClean's performance and show that, when random variables have large domains, partitioning can lead to speedups of up to 2x without a significant deterioration in quality (an average drop of 0.5% in F1 is observed).

(3) We introduce an approximation scheme that relaxes hard integrity constraints to priors over independent random variables (see Section 5.2). This relaxation results in a probabilistic model for which Gibbs sampling requires only a polynomial number of samples to mix. We empirically study the tradeoff between the runtime and quality of repairs obtained when relaxing integrity constraints. We show that our approximation not only leads to more scalable data repairing models but also results in repairs of the same quality as those obtain by non-approximate models (see Section 6).

2. THE HoloClean FRAMEWORK

We formalize the goal of HoloClean and provide an overview of HoloClean's solution to data repairing.

2.1 Problem Statement

The goal of HoloClean is to identify and repair erroneous records in a structured dataset D. We denote $A = \{A_1, A_2, \ldots, A_N\}$ the attributes that characterize D. We represent D as a set of tuples, where each tuple $t \in D$ is a set of cells denoted as $Cells[t] = \{A_i[t]\}$. Each cell c corresponds to a different attribute in A (e.g., the dataset in Figure 1 has attributes, "DBAName", "AKAName", "Address", "City", "State", and "Zip", and consists of four tuples). We denote $t[A_n]$ the n-th cell of tuple t for attribute $A_n \in A$.

We assume that errors in D occur due to inaccurate cell assignments, and we seek to repair errors by updating the values of cells in D. This is a typical assumption in many data cleaning systems [15, 16, 39]. For each cell we denote by v_c^* its unknown true value and by v_c its initial observed value. We use Ω to denote the initial observed values for all cells in D. We term an *error* in D to be each cell c with $v_c \neq v_c^*$. The goal of HoloClean is to estimate the latent true values v_c^* for all erroneous cells in D. We use $\hat{v}_c$ to denote the estimated true value of a cell $c \in D$. We say that an inaccurate cell is correctly repaired when $\hat{v}_c = v_c^*$.

2.2 Solution Overview

An overview of HoloClean is shown in Figure 2. HoloClean takes as input a dirty dataset D, along with a set of available repairing constraints Σ. In the current implementation we limit these constraints to: (i) *denial constraints* [13] that specify various business logic and integrity constraints, and (ii) *matching dependencies* [7, 16, 24] to specify lookups to available external dictionaries or labeled (clean) data. We briefly review denial constraints in Section 3.1. HoloClean's workflow follows three steps:

Error Detection. The first step in the workflow of HoloClean is to detect cells in D with potentially inaccurate values. This process separates D into *noisy* and *clean* cells, denoted D_n and D_c, respectively. HoloClean treats error detection as a black box. Users have the flexibility to use any method that detects erroneous cells. The output of such methods is used to form D_n and D_c is set to $D_c = D \setminus D_n$. Our current implementation includes a series of error detection methods, such as methods that leverage denial constraints to detect erroneous cells [14], outlier detection mechanisms [19, 29], and methods that rely on external and labeled data [7, 16, 24]. For a detailed study of state-of-the-art error detection methods we refer the reader to the work of Abedjan et al. [4].

Compilation. Given the initial cell values Ω and the set of repairing constraints Σ, HoloClean follows probabilistic semantics to express the uncertainty over the value of noisy cells. Specifically, it associates each cell $c \in D$ with a random variable T_c that takes values form a finite domain $dom(c)$, and compiles a probabilistic graphical model that describes the distribution of random variables T_c for cells in D. HoloClean relies on factor graphs [35] to represent the probability distribution over variables T_c. HoloClean uses DeepDive [46], a declarative probabilistic inference framework, to run statistical learning and inference. In Section 3.2, we review factor graphs and how probabilistic models are defined in DeepDive.

Data Repairing. To repair D, HoloClean runs statistical learning and inference over the joint distribution of variables $T_1, T_2, \ldots$ to compute the marginal probability $P(T_c = d; \Omega, \Sigma)$ for all values $d \in dom(c)$. Let T be the set of all variables T_c. HoloClean uses empirical risk minimization (ERM) over the likelihood $\log P(T)$ to compute the parameters of its probabilistic model. Variables that correspond to clean cells in D_c are treated as labeled examples to learn the parameters of the model. Variables for noisy cells in D_n correspond to query variables whose value needs to be inferred. Approximate inference via Gibbs sampling [50] is used to estimate the value $\hat{v}_c$ of noisy cells. Variables $\hat{v}_c$ are assigned to the maximum a posteriori (MAP) estimates of variables T_c.

Each repair by HoloClean is associated with a calibrated marginal probability. For example, if the proposed repair for a record in the initial dataset has a probability of 0.6 it means that HoloClean is 60% confident about this repair. Intuitively, this means that if HoloClean proposes 100 repairs then only 60 of them will be correct. As a result, we can use these marginal probabilities to solicit user feedback. For example, we can ask users to verify repairs with low marginal probabilities and use those as labeled examples to retrain the parameters of HoloClean's model using standard incremental learning and inference techniques [46].

Finally, similar to existing automatic data repairing approaches, HoloClean's recall is limited by the error detection methods used. Error detection is out of the scope of this paper. However, we examine its effect on HoloClean's performance in Section 6.3.4. As part of future directions (Section 8), we also discuss how state-of-the-art weak supervision methods can be used to improve error detection.

3. BACKGROUND

We review concepts and terminology used in the next sections.

3.1 Denial Constraints

In HoloClean users specify a set of *denial constraints* [13] to ensure the consistency of data entries in D. Denial constraints subsume several types of integrity constraints such as functional dependencies, conditional functional dependencies [11], and metric functional dependencies [36]. Recent work has also introduced methods that automatically discover denial constraints [14].

Given a set of operators $B = \{=, <, >, \neq, \leq, \approx\}$, with $\approx$ denoting similarity, denial constraints are first-order formulas over *cells* of tuples in dataset D that take the form $\sigma : \forall t_i, t_j \in D : \neg(P_1 \wedge \cdots \wedge P_k \wedge \cdots \wedge P_K)$ where each predicate P_k is of the form $(t_i[A_n]\ \mathrm{o}\ \mathrm{t_j[A_m]})$ or $(t_i[A_n]\ \mathrm{o}\ \alpha)$ where $A_n, A_m \in A$, α denotes a constant and $\mathrm{o} \in \mathrm{B}$. We illustrate this with an example:

Example 2. *Consider the functional dependency Zip → City, State from the food inspection dataset in Figure 1. This dependency can be represented using the following two denial constraints:*

$$\forall t_1, t_2 \in D : \neg(t1[Zip] = t2[Zip] \wedge t1[City] \neq t2[City])$$
$$\forall t_1, t_2 \in D : \neg(t1[Zip] = t2[Zip] \wedge t1[State] \neq t2[State])$$

3.2 Factor Graphs

A factor graph is a hypergraph (T, F, θ) in which T is a set of nodes that correspond to random variables and F is a set of hyperedges. Each hyperedge $\phi \in F$, where $\phi \subseteq T$, is referred to as a *factor*. For ease of exposition only, we assume that all variables T have a common domain $\mathbb{D}$. Each hyperedge ϕ is associated with a *factor function* and a real-valued weight θ_ϕ and takes an assignment of the random variables in ϕ and returns a value in $\{-1, 1\}$ (i.e., $h_\phi : \mathbb{D}^{|f|} \rightarrow \{-1, 1\}$). Hyperedges f, functions h_ϕ, and weights θ_ϕ define a *factorization* of the probability distribution $P(T)$ as:

$$P(T) = \frac{1}{Z} \exp\left(\sum_{\phi \in F} \theta_\phi \cdot h_\phi(\phi)\right) \quad (1)$$

where Z is called the *partition function* and corresponds to a constant ensuring we have a valid distribution.

Recently, declarative probabilistic frameworks [2, 6, 46] have been introduced to facilitate the construction of large scale factor graphs. In HoloClean, we choose to use DeepDive [46]. In DeepDive, users specify factor graphs via *inference rules* in DDlog, a declarative language that is semantically similar to Datalog but extended to encode probability distributions. A probabilistic model in DeepDive corresponds to a collection of rules in DDlog.

DDlog rules are specified over relations in a database. For example, the following DDlog rule states that the tuples of relation Q are derived from R and S via an equi-join on attribute "y".

$$\mathsf{Q(x, y)} : -\mathsf{R(x, y)}, \mathsf{S(y)}, [\mathsf{x} = \text{“a”}]$$

$Q(x, y)$ is the *head* of the rule, and $R(x, y)$ and $S(y)$ are *body atoms*. The body also contains a condition $[x =$ "a"$]$ on the values that Attribute "x" can take. Predicate $[x =$ "a"$]$ is called the scope of the rule. Finally, x and y are variables of the rule. We next describe how such rules can be extended to define a factor graph.

Relations in DDlog can be augmented with a question-mark annotation to specify random variables. For example, if $\mathsf{Fact(x)}$ is a relation of facts for which we want to infer if they are `True` or `False`, then $\mathsf{IsTrue?(x)}$ is a relation such that each assignment to x represents a different random variable taking the value `True` or `False`. The next DDlog rule defines this random variable relation:

$$\mathsf{IsTrue?(x)} : -\mathsf{Fact(x)}$$

Grounding relation Fact generates a random variable for each value of x. These correspond to nodes T in the factor graph. In the remainder of the paper we refer to relations annotated with a question-mark as *random variable relations*.

Given relations that define random variables we can extend Datalog rules with weight annotations to encode *inference rules*, which express the factors of a factor graph. We continue with the previous example and let $\mathsf{HasFeature(x, f)}$ be a relation that contains information about the features f that a fact x can have. We consider the following inference rule:

$$\mathsf{IsTrue?(x)} : -\mathsf{HasFeature(x, f)}\ \mathsf{weight} = \mathsf{w(f)}$$

The head of this rule defines a *factor function* that takes as input one random variable—corresponding to an assignment of variable x—and returns 1.0 when that variable is set to `True` and -1.0 otherwise. This rule associates the features for each fact x with its corresponding random variable. The weights are parameterized by variable f to allow for different confidence levels across features. To generate the factors from the above rule, we ground its body by evaluating the corresponding query. Grounding generates a factor (hyper-edge in the factor graph) for each assignment of variables x and f. The head of inference rules can be a complex boolean function that introduces correlations across random variables.

Finally, variables in the generated factor graph are separated in two types: a set E of *evidence variables* (those fixed to a specific value) and a set Q of *query variables* whose value needs to be inferred. During learning, the values of unknown weights in w are set to the values that maximize the probability of evidence. Then, inference proceeds with the values of all weights w being fixed.

4. COMPILATION IN HOLOCLEAN

HoloClean compiles all available repair signals to a DDlog program. The generated DDlog program contains: (i) rules that capture quantitative statistics of D; (ii) rules that encode matching dependencies over external data; (iii) rules that represent dependencies due to integrity constraints; (iv) rules that encode the principle of minimality. The groundings of these rules construct factors h_ϕ in Equation 1 as in described Section 3.2.

HoloClean's compilation involves two steps: (i) first HoloClean generates relations used to form the body of DDlog rules, and then (ii) uses those relations to generate inference DDlog rules that define HoloClean's probabilistic model. The output DDlog rules define a probabilistic program, which is then evaluated using the DeepDive framework. We describe each of these steps in detail.

4.1 DDlog Relations in HoloClean

HoloClean generates several relations that are transformations of Dataset D. The following two variables are used to specify fields of these relations: (i) t is a variable that ranges over the identifiers of tuples in D, and (ii) a is a variable that ranges over the attributes of D. We also denote by $t[a]$ a cell in D that corresponds to attribute a of tuple t. HoloClean's compiler generates the following relations:

(1) $\mathsf{Tuple(t)}$ contains all identifiers of tuples in D.
(2) $\mathsf{InitValue(t, a, v)}$ maps every cell $t[a]$ to its initial value v.
(3) $\mathsf{Domain(t, a, d)}$ maps every cell $t[a]$ to the possible values it can take, where variable d ranges over the domain of $t[a]$.
(4) $\mathsf{HasFeature(t, a, f)}$ associates every cell $t[a]$ with a series of features captured by variable f.

Relations Tuple, InitValue, and Domain are populated directly from the values in D, and the domain of each attribute in D. In Section 5.1.1, we show how to prune entries in Relation Domain for scalable inference. Finally, HasFeature is populated with two types of features: (i) given a cell c, HoloClean considers as features the values of other cells in the same tuple as c (e.g., "Zip=60608"). These features capture distributional properties of D that are manifested in the co-occurrences of attribute values; and (ii) if the provenance and lineage of $t[a]$ is provided (e.g., the source from which t was obtained) we use this information as additional features. This allows HoloClean to reason about the trustworthiness of different sources [44] to obtain more accurate repairs. Users can specify additional features by adding tuples in Relation HasFeature.

To capture external data, HoloClean assumes an additional relation that is optionally provided as input by the user:

(5) $\mathsf{ExtDict(t_k, a_k, v, k)}$ stores information from external dictionaries identified by the indicator variable k. Variables t_k and a_k range over the tuples and attributes of dictionary k, respectively. Relation ExtDict maps each $t_k[a_k]$ to its value v in Dictionary k.

4.2 Translating Signals to Inference Rules

HoloClean's compiler first generates a DDlog rule to specify the random variables associated with cells in the input dataset D:

$$\mathsf{Value?(t, a, d) :- Domain(t, a, d)}$$

This rule defines a random variable relation $\mathsf{Value?(t, a, d)}$, which assigns a categorical random variable to each cell $t[a]$. Grounding this rule generates the random variables in HoloClean's probabilistic model. Next, we show how HoloClean expresses each repair signal (see Section 1), as an inference DDlog rule over these random variables. Grounding these rules populates the factors used in HoloClean's probabilistic model, which completes the specification of the full factor graph used for inferring the correct repairs.

Algorithm 1: Denial Constraint Compilation to DDlog Rules

Input: Denial constraints in Σ, constant weight w
Output: DDlog Rules for Denial Constraints
rules = [];
for *each constraint* $\sigma : \forall t_1, t_2 \in D : \neg(P_1 \wedge \cdots \wedge P_K)$ **do**
  /* Initialize the head and scope of the new DDlog rule*/
  $H \leftarrow \emptyset, S \leftarrow \emptyset$;
  for *each predicate* P_k *in* σ **do**
    if P_k *is of the form* $(t_1[A_n] \circ t_2[A_m])$ **then**
      $H = H \cup \{\mathsf{Value?(t1, A_n, v_{1k})} \wedge \mathsf{Value?(t2, A_m, v_{2k})}\}$;
      $S = S \cup \{\mathsf{v_{1k}} \circ \mathsf{v_{2k}}\}$;
    if P_k *is of the form* $(t_1[A_n] \circ \alpha)$ **then**
      $H = H \cup \{\mathsf{Value?(t1, A_n, v_{1k})}\}$;
      $S = S \cup \{\mathsf{v_{1k}} \circ \alpha\}$;
  rules += ! $\bigwedge_{h \in H} h$: −Tuple(t1), Tuple(t2),[S] weight = w;
return rules;

Quantitative Statistics. We use the features stored in Relation $\mathsf{HasFeature(t, a, f)}$, to capture the quantitative statistics of Dataset D. HoloClean encodes the effect of features on the assignment of random variables with the DDlog rule:

$$\mathsf{Value?(t, a, d) :- HasFeature(t, a, f)\ weight = w(d, f)}$$

Weight $w(d, f)$ is parameterized by d and f to allow for different confidence levels per feature. Weights are learned using evidence variables as labeled data (Section 2.2).

External Data. Given Relation $\mathsf{ExtDict(t_k, a_k, v, k)}$, described in Section 4.1, along with a collection of matching dependencies—expressed as implications in first-order logic—HoloClean generates additional DDlog rules that capture the effect of the external dictionaries on the assignment of random variables. First, HoloClean generates DDlog rules to populate a relation Matched, which contains all identified matches. We use an example to demonstrate the form of DDlog rules used to populate Matched:

Example 3. *We consider the matching dependency between Zip and City from Example 1. HoloClean generates the DDlog rule:*

$$\mathsf{Matched(t1, City, c2, k) : -Domain(t1, City, c1),}$$
$$\mathsf{InitValue(t1, Zip, z1), ExtDict(t2, Ext\_Zip, z1, k),}$$
$$\mathsf{ExtDict(t2, Ext\_City, c2, k)}\ , [c1 \approx c2]$$

where $\approx$ is a similarity operator and k is the indicator of the external dictionary used. The rule dictates that for a tuple $t1$ in D if the zip code matches the zip code of a tuple $t2$ in the external dictionary k, then the city of $t1$ has to match the city of $t2$. The DDlog formula populates Matched *with the tuple $\langle t1, City, c2\rangle$, where $c2$ is the lookup value of $t1[City]$ in Dictionary k.*

HoloClean's compiler generates the following inference rule to capture the dependencies between external dictionaries and random variables using Relation Matched:

$$\mathsf{Value?(t, a, d) :- Matched(t, a, d, k)\ weight = w(k)}$$

Weight $w(k)$ is parameterized by the identifier of the dictionary, k, to encode different reliability levels per dictionary.

Dependencies From Denial Constraints. HoloClean uses Algorithm 1 to convert denial constraints to DDlog rules. In Algorithm 1, the quantifier $\forall t_1, t_2$ of a denial constraint σ is converted to a self-join Tuple(t1), Tuple(t2) over Relation Tuple in DDlog. We apply standard ordering strategies to avoid grounding duplicate

factors. The details are omitted from the pseudocode for clarity. We illustrate Algorithm 1 with an example:

Example 4. *Consider the following denial constraint:*

$$\forall t_1, t_2 \in D : \neg(t1[Zip] = t2[Zip] \wedge t1[State] \neq t2[State])$$

This constraint can be expressed as a factor template in DDlog as:

$$!(\mathsf{Value?(t1, Zip, z1)} \wedge \mathsf{Value?(t2, Zip, z2)} \wedge$$
$$\mathsf{Value?(t1, State, s1)} \wedge \mathsf{Value?(t2, State, s2)}) :-$$
$$\mathsf{Tuple(t1)}, \mathsf{Tuple(t2)}, [z1 = z2, s1 \neq s2]\ weight = w$$

Setting $w = \infty$ converts these factors to hard constraints. However, probabilistic inference over hard constraints is in general #-P complete [20]. HoloClean allows users to relax hard constraints to *soft constraints* by assigning w to a constant value. The larger w the more emphasis is put on satisfying the given denial constraints.

Minimality Priors. Using minimality as an operational principle might lead to inaccurate repairs [30]. However, minimality can be viewed as the prior that the input dataset D contains fewer erroneous records than clean records. To capture this prior, HoloClean generates the following DDlog rule:

$$\mathsf{Value?(t, a, d)} :- \mathsf{InitValue(t, a, d)}\ \mathsf{weight = w}$$

Weight w is a positive constant indicating the strength of this prior. So far we showed how HoloClean maps various signals into DDlog rules for constructing the factor graph used for data repairing. In the following section, we show how we manage the complexity of the generated factor graph to allow for efficient inference.

5. SCALING INFERENCE IN HOLOCLEAN

HoloClean uses the DeepDive framework to ground the DDlog generated by its compilation module and to run Gibbs sampling [46] to perform inference. Both grounding and Gibbs sampling introduce significant challenges: (i) grounding is prone to combinatorial explosion [35]; and (ii) in the presence of complex correlations, Gibbs sampling requires an exponential number of iterations in the number of random variables to mix, i.e., reach a stationary distribution, and accurately estimate the marginal probabilities of query variables [45]. To address these challenges we introduce: two optimizations to limit the combinatorial explosion during grounding, and one optimization that guarantees $O(n \log n)$ iterations for Gibbs sampling to mix, where n is the number of random variables.

5.1 Scalable Grounding in HoloClean

Combinatorial explosion during grounding can occur due to random variables with large domains which participate in factors that encode dependencies due to denial constraints (see Section 4). We consider the DDlog rule in Example 4 to demonstrate this problem:

Example 5. *Consider an input instance, such that all random variables associated with "Zip" take values from a domain Z, all random variables for "State" take values from a domain S, and there are T tuples in D. Given the constraints $z1 = z2$ and $s1 \neq s2$, we have that the total number of groundings just for all tuples is $O(|T|^2 \cdot |Z| \cdot |S|^2)$. The combinatorial explosion is apparent for large values of either $|S|$ or $|T|$.*

There are two aspects that affect HoloClean's scalability: (i) random variables with large domains (e.g., $|S|$ in our example), and (ii) factors that express correlations across all pairs of tuples in D (e.g., $|T|$ in our example). We introduce two optimizations: (i) one for pruning the domain of random variables by leveraging co-occurrence statistics over the cell values in D, and (ii) one for pruning the pairs of tuples over which denial constraints are evaluated.

5.1.1 Pruning the Domain of Random Variables

Each cell c in D corresponds to a random variable T_c. These random variables are separated in evidence variables whose value is fixed—these correspond to clean cells in D_c—and query variables whose value needs to be inferred—these correspond to noisy cells in D_n (see Section 2). HoloClean needs to determine the domain of query random variables. In the absence of external domain knowledge, data repairing algorithms usually allow a cell to obtain any value from the active domain of its corresponding attribute, namely, the values that have appeared in the attribute associated with that cell [10, 15].

In HoloClean, we use a different strategy to determine the domain of random variables T_c: Consider a cell $c \in D_n$ and let t denote its tuple. We consider the values that other cells in tuple t take. Let c' be a cell in t different than c, $v_{c'}$ its value, and $A_{c'}$ its corresponding attribute. We consider candidate repairs for c to be all values in the domain of c's attribute, denoted A_c, that co-occur with value $v_{c'}$. To limit the set of candidate repairs we only consider values that co-occur with a certain probability that exceeds a pre-defined threshold τ. Following a Bayesian analysis we have: Given a threshold τ and values v for A_c and $v_{c'}$ for $A_{c'}$, we require that the two values co-occur if $Pr[v|v_{c'}] \geq \tau$. We define this as:

$$Pr[v|v_{c'}] = \frac{\#(v, v_{c'}) \text{ appear together in } D}{\#v_{c'} \text{ appears in } D}$$

The overall algorithm is shown in Algorithm 2.

Algorithm 2: Domain Pruning of Random Variables

Input: Set of Noisy Data Cells D_n, Dataset D, Threshold τ
Output: Repair Candidates for Each Cell in D_n
for *each cell c in D_n* **do**
  /* Initialize repair candidates for cell c */
  $R_c \leftarrow \emptyset$;
  for *each cell c in D_n* **do**
    $A_c \leftarrow$ the attribute of cell c;
    for *each cell $c' \neq c$ in c's tuple* **do**
      $U_{A_c} \leftarrow$ the domain of attribute A_c;
      $v_{c'} \leftarrow$ the value of cell c';
      for *each value $v \in U_{A_c}$* **do**
        if $Pr[v|v_{c'}] \geq \tau$ **then**
          $R_c \leftarrow R_c \cup \{v\}$;
return repair candidates R_c for each $c \in D_n$;

Discussion. Varying threshold τ allows users to tradeoff the scalability of HoloClean and the quality of repairs obtained by it. Higher values of τ lead to smaller domains for random variables, thus, smaller size factor graphs and more efficient inference. At the same time, τ introduces a tradeoff between the precision and recall of repairs output by HoloClean. We study the effect of τ on the performance of HoloClean in Section 6. Setting the value for τ is application-dependent and is closely tied to the requirements that users have on the performance (both with respect to quality and runtime) of data cleaning. Many data cleaning solutions, including tools for entity resolution [47], expose similar knobs to users.

5.1.2 Tuple Partitioning Before Grounding

Grounding the DDlog rules output by Algorithm 1 requires iterating over all pairs of tuples in D and evaluating if the body of each DDlog rule is satisfied for the random variables corresponding to their cells. However, in practice, there are many tuples that will never participate in a constraint violation (e.g., the domains of their cells may never overlap). To avoid evaluating DDlog rules for

such tuples, we introduce a scheme that partitions D in groups of tuples such that tuples in the same group have a high probability of participating in a constraint violation. DDlog rules are evaluated only over these groups, thus, limiting the quadratic complexity of grounding the rules generated by Algorithm 1.

To generate these groups we leverage *conflict hypergraphs* [34] which encode constraint violations in the original dataset D. Nodes in conflict hypergraph H correspond to cells that participate in detected violations and hyperedges link together cells involved in the same violation. Hyperedges are also annotated with the constraint that generated the violation. For each constraint $\sigma \in \Sigma$ we consider the subgraph of H containing only hyperedges for violations of σ. Let H_σ be the induced subgraph. We let each connected component in H_σ define a group of tuples over which the factor for constraint σ will be materialized.

Algorithm 3: Generating Tuple Groups

Input: Dataset D, Constraints Σ, Conflict Hypergraph H
Output: Groups of Tuples
/* Initialize set of tuple groups */
$G \leftarrow \emptyset$;
for *each constraint* σ *in* Σ **do**
  $H_\sigma \leftarrow$ subgraph of H with violations of σ;
  for *each connected component* cc *in* H_σ **do**
    $G \leftarrow G \cup \{(\sigma, \text{tuples from } D \text{ present in } cc\})$;
return set of tuple groups G;

We use Algorithm 3 to restrict the groundings of rules generated by Algorithm 1 only over tuples in the same connected component with respect to each denial constraint $\sigma \in \Sigma$. Our partitioning scheme limits the number of factors generated due to denial constraints to $O(\sum_{g \in G} |g|^2)$ as opposed to $O(|\Sigma||D|^2)$. In the worst case the two quantities can be the same. In our experiments, we observe that, when random variables have large domains, our partitioning optimization leads to more scalable models—we observe speed-ups up to $2\times$—that output accurate repairs; compared to inference without partitioning, we find an F1-score decrease of 6% in the worst case and less than 0.5% on average.

5.2 Rapid Mixing of Gibbs Sampling

Gibbs sampling requires that we iterate over the random variables in the factor graph and, at every step, sample a single variable from its conditional distribution (i.e., keep all other variables fixed). If a factor graph has only independent random variables then Gibbs sampling requires $O(n \log n)$ steps to mix [28, 45].

Motivated by this result, we introduce an optimization that relaxes the DDlog rules generated by Algorithm 1 to obtain a model with independent random variables. Instead of enforcing denial constraints for any assignment of the random variables corresponding to noisy cells in D, we generate features that provide evidence on random variable assignments that lead to constraint violations. To this end, we introduce an approximation of our original probabilistic model that builds upon two assumptions: (i) erroneous cells in D are fewer than correct cells, i.e., there is sufficient redundancy to fix errors in D, and (ii) each integrity constraint violation can be fixed by updating a single cell in the participating tuples.

We relax the DDlog rules: For each rule generated by Algorithm 1, iterate over each Value?() predicate and generate a new DDlog rule whose head contains only that predicate, while all remaining Value?() predicates are converted to InitValue() predicates in the body of the rule. Also the weights of the original rules are relaxed to learnable parameters of the new model. The above procedure decomposes each initial DDlog rule into a series of new rules whose head contains a single random variable. We use an example to demonstrate the output of this procedure.

Table 1: Parameters of the data used for evaluation. Noisy cells do not necessarily correspond to erroneous cells.

Parameter	Hospital	Flights	Food	Physicians
Tuples	1,000	2,377	339,908	2,071,849
Attributes	19	6	17	18
Violations	6,604	84,413	39,322	5,427,322
Noisy Cells	6,140	11,180	41,254	174,557
ICs	9 DCs	4 DCs	7 DCs	9 DCs

Example 6. *We revisit Example 4. Our approximation procedure decomposes the initial DDlog rule into the following rules:*

$$\mathsf{!Value?(t1, Zip, z1)} :- \mathsf{InitValue(t2, Zip, z2)},$$
$$\mathsf{InitValue(t1, State, s1)}, \mathsf{InitValue(t2, State, s2)}),$$
$$\mathsf{Tuple(t1)}, \mathsf{Tuple(t2)}, \ [t1! = t2, z1 = z2, s1 \neq s2] \ weight = w$$

and

$$\mathsf{!Value?(t1, State, s1)} :- \mathsf{InitValue(t1, Zip, z1)},$$
$$\mathsf{InitValue(t2, Zip, z2)}, \mathsf{InitValue(t2, State, s2)}),$$
$$\mathsf{Tuple(t1)}, \mathsf{Tuple(t2)}, \ [t1! = t2, z1 = z2, s1 \neq s2] \ weight = w$$

where in contrast to the fixed weight of the original rule, w for the two rules above is a weight to be estimated during learning.

Our relaxed model comes with two desired properties: (i) the factor graph generated by relaxing the original DDlog rules contains only independent random variables, hence, Gibbs sampling is guaranteed to mix in $o(n \log n)$ steps, and (ii) since random variables are independent learning the parameters of Equation 1 corresponds to a convex optimization problem. In Section 6.3, we show that this model not only leads to more scalable data repairing methods but achieves the same quality repairs as the non-relaxed model.

Discussion. HoloClean's initial probabilistic model enforces denial constraints *globally*, i.e., for each possible assignment of the random variables corresponding to noisy cells in D. On the other hand, the approximate model can be viewed as a model that enforces *local consistency* with respect to the initially observed values in D. In Section 6, we empirically find that when there is sufficient redundancy in observing the correct value of cells in a dataset, our approximate model obtains more accurate repairs and is more robust to misspecifications of the domain of random variables in HoloClean's probabilistic model (i.e., less sensitive to the value that parameter τ takes). We are actively working on theoretically analyzing the connections between the above model and involved notions of minimality, such as cardinality-set-minimality [8].

6. EXPERIMENTS

We compare HoloClean against state-of-the-art data repairing methods on a variety of synthetic and real-world datasets.The main points we seek to validate are: (i) how accurately can HoloClean repair real-world datasets containing a variety of errors, (ii) what is the impact of different signals on data repairing, and (iii) what is the impact of our pruning methods on the scalability and accuracy of HoloClean. Finally, we study the impact of error detection on HoloClean's performance.

6.1 Experimental Setup

We describe the datasets, metrics, and experimental settings used to validate HoloClean against competing data repairing methods.

Datasets. We use four real data sets. For all datasets we seek to repair cells that participate in violations of integrity constraints. Table 1 shows statistics for these datasets. As shown, the datasets span different sizes and exhibit various amounts of errors:

Hospital. This is a benchmark dataset used in the literature [15, 18]. Errors amount to $\sim 5\%$ of the total data. Ground truth information is available for all cells. This dataset exhibits significant duplication across cells. *We use it to evaluate how effective HoloClean is at leveraging duplicate information during cleaning.*

Flights. This dataset [38] contains data on the departure and arrival time of flights from different data sources. We use four denial constraints that ensure a unique scheduled and actual departure and arrival time for each flight. Errors arise due to conflicts across data sources. Ground truth information is available for all cells. The majority of cells in Flights are noisy and the lineage of each tuple is know. *We use this dataset to examine how robust HoloClean is in the presence of many errors, and to evaluate if HoloClean can exploit conflicts across data sources to identify correct data repairs.*

Food. This is the dataset from Example 1. Errors correspond to conflicting zip codes for the same establishment, conflicting inspection results for the same establishment on the same day, conflicting facility types for the same establishment and many more. These errors are captured by seven denial constraints. The majority of errors are introduced in non-systematic ways. The dataset also contains many duplicates as records span different years. *We use this dataset to evaluate HoloClean against real data with duplicate information and non-systematic errors.*

Physicians. This is the Physician Compare National dataset published in Medicare.gov [3]. We used nine denial constraints to identify errors in the dataset. The majority of errors correspond to systematic errors. For example, the location field for medical organizations is misspelled, thus, introducing systematic errors across entries of different professionals. For instance, "Sacramento, CA" is reported as "Scaramento, CA" in 321 distinct entries. Other errors include zip code to state inconsistencies. *We use this dataset to evaluate HoloClean against datasets with systematic errors.*

Competing Methods. For the results in this section, denial constraints in HoloClean are relaxed to features (see Section 5.2). No partitioning is used. We evaluate HoloClean against:

- **Holistic** [15]: This method leverages denial constraints to repair errors. Holistic is shown to outperform other methods based on logical constraints, thus, we choose to compare HoloClean against this method alone.
- **KATARA** [16]: This is a knowledge base (KB) powered data cleaning system that, given a dataset and a KB, interprets table semantics to align it with the KB, identifies correct and incorrect data, and generates repairs for incorrect data. We use an external dataset containing a list of States, Zip Codes, and Location information as external information.
- **SCARE** [49]: This is a method that uses machine learning to clean dirty databases by value modification. This approach does not make use of integrity or matching constraints.

Features, Error Detection, and External Signals. The probabilistic models generated by HoloClean capture all features described in Section 4. Source-related features are only available for Flights. To detect erroneous cells in HoloClean, we used the same mechanism as Holistic [15]. Finally, for micro-benchmarking purposes we use the dictionary used for KATARA on Hospital, Food, and Physicians. Unless explicitly specified HoloClean does not make use of this external information.

Evaluation Methodology. To measure the quality of repairs by different methods we use the following metrics:

- **Precision (Prec.)**: the fraction of correct repairs, i.e., repairs that match the ground truth, over the total number of repairs performed for cells in the labeled data.
- **Recall (Rec.)**: correct repairs over the total number of errors. The total number of errors is computed over the available labeled data for each dataset.
- **F1-score (F1)**: the harmonic mean of precision and recall computed as $2 \times (Prec. \times Rec.)/(Prec. + Rec.)$.

For Hospital and Flights we have full ground truth information. For Food and Physicians we manually labeled a subset of the data as described below. For each method we also measure the overall *wall-clock runtime*. For HoloClean this is: (i) the time for detecting violations, (ii) the time for compilation, and (iii) the time needed to run learning and inference. Finally, we vary the threshold τ of our pruning optimization for determining the domain of cell-related random variables (see Algorithm 2) in $\{0.3, 0.5, 0.7, 0.9\}$.

Obtaining Groundtruth Data. To evaluate data repairing on Food and Physicians, we manually labeled 2,000 and 2,500 cells, respectively: We focused on tuples identified as erroneous by the error detection mechanisms of Holistic, KATARA, and SCARE. From this set of cells we randomly labeled 2,000 cells for Food and 2,500 cells for Physician. Not all cells were indeed noisy. This process leads to unbiased estimates for the precision of each method. However, recall measurements might be biased.

Implementation Details. HoloClean's compiler is implemented in Python while the inference routines are executed in DeepDive v0.9 using Postgres 9.6 for backend storage. Holistic is implemented in Java and uses the Gurobi Optimizer 7.0 as its external QP tool. KATARA and Scare are also implemented in Java. All experiments were executed on a machine with four CPUs (each CPU is a 12-core 2.40 GHz Xeon E5-4657L), 1TB RAM, running Ubuntu 12.04. While all methods run in memory, their footprint is significantly smaller than the available resources.

6.2 Experimental Results

We compare HoloClean with competing data repairing approaches on the quality of the proposed repairs. We find that in all cases HoloClean outperforms all state-of-the-art data repairing methods and yields an average F1-score improvement of more than $2\times$.

6.2.1 Identifying Correct Data Repairs

We report the precision, recall, and F1-score obtained by HoloClean and competing approaches. The results are shown in Table 2. For each dataset, we report the threshold τ used for pruning the domain of random variables. The effect of τ on the performance of HoloClean is studied in Section 6.3.1. As shown in Table 2 HoloClean outperforms other data repairing methods significantly with relative F1-score improvements of more than 40% in all cases. This verifies our hypothesis that unifying multiple signals leads to more accurate automatic data cleaning techniques.

We focus on HoloClean's performance for the different datasets. For Hospital, HoloClean leverages the low number of errors and the presence of duplicate information to correctly repair the majority of errors, achieving a precision of 100% and a recall of 71.3%. HoloClean also achieves high precision for Flights (88.8%), as it

Table 2: Precision, Recall and F1-score for different datasets. For each dataset, the threshold used for pruning the domain of random variables is reported in parenthesis.

Dataset (τ)	Metric	HoloClean	Holistic	KATARA	SCARE
Hospital (0.5)	Prec.	**1.0**	0.517	0.983	0.667
	Rec.	**0.713**	0.376	0.235	0.534
	F1	**0.832**	0.435	0.379	0.593
Flights (0.3)	Prec.	**0.887**	0.0	n/a	0.569
	Rec.	**0.669**	0.0	n/a	0.057
	F1	**0.763**	0.0*	n/a	0.104
Food (0.5)	Prec.	**0.769**	0.142	1.0	0.0
	Rec.	**0.798**	0.679	0.310	0.0
	F1	**0.783**	0.235	0.473	0.0+
Physicians (0.7)	Prec.	**0.927**	0.521	0.0	0.0
	Rec.	**0.878**	0.504	0.0	0.0
	F1	**0.897**	0.512	0.0#	0.0+

* Holistic did not perform any correct repairs.
\+ SCARE did not terminate after three days.
\# KATARA performs no repairs due to format mismatch for zip code.

Table 3: Runtime analysis of different data cleaning methods. A dash indicates that the system failed to terminate after a three day runtime threshold.

Dataset	HoloClean	Holistic	KATARA	SCARE
Hospital	147.97 sec	5.67 sec	2.01 sec	24.67 sec
Flights	70.6 sec	80.4 sec	n/a	13.97 sec
Food	32.8 min	7.6 min	1.7 min	-
Physicians	6.5 hours	2.03 hours	15.5 min	-

uses the information on which source provided which tuple to estimate the reliability of different sources [44] and leverages that to propose repairs. Nonetheless, we see that recall is limited (66.9%) since most of the cells contains errors. Finally, for Food and Physician HoloClean obtains F1-scores of 0.783 and 0.897, respectively.

We now turn our attention to competing methods. Holistic yields repairs of fair quality (around 50% F1) for datasets with a large number of duplicate information (e.g., Hospital) or a large number of systematic errors (e.g., Physicians). When datasets contain mostly noisy cells (as in Flights) or errors that follow random patterns (as in Food) using logical constraints and minimality yields very poor results—the precision of performed repairs is 0.0 for Flights and 0.14 for Foods.

In contrast, KATARA obtains repairs of very high precision but limited recall. This is expected as the coverage of external knowledge bases can be limited. Finally, SCARE performs reasonably well in datasets such as Hospital, where a large number of duplicate records is available and qualitative statistics can help repair errors. Similar to HoloClean it is able to leverage existing correct tuples to perform repairs. However, for Flights, where the number of duplicates is limited, SCARE has limited recall. Also SCARE failed to terminate after running for three days on Food and Physicians.

Takeaways. HoloClean's holistic approach obtains data repairs that are significantly more accurate—we find an F1-score improvement of more than 2× on average—than existing state-of-the-art approaches that consider isolated signals for data repairing.

6.2.2 *Runtime Overview*

We measure the total wall-clock runtime of each data repairing method for all datasets. The results are shown in Table 3. Reported runtimes correspond to end-to-end execution with data preprocessing and loading. For Holistic, pre-processing corresponds to loading input data from raw files and running violation detection. SCARE operates directly on the input database, while KATARA loads data in memory and performs matching and repairing.

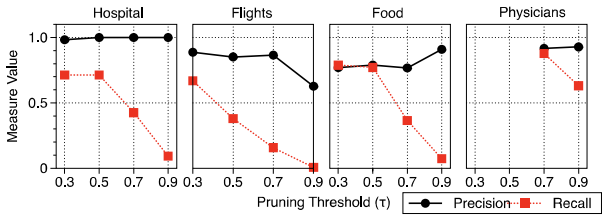

Figure 3: Effect of pruning on Precision and Recall. Missing values correspond to time-outs with a threshold of one day.

As shown HoloClean can scale to large real-world data repairing scenarios. For small datasets, i.e., Hospital and Flights, the total execution time of HoloClean is within one order of magnitude of Holistic's runtime but still only a few minutes in total. For Food, HoloClean exhibits a higher runtime but for Physicians both systems are within the same magnitude. KATARA is faster as it only performs matching operations. Finally, while SCARE is very fast for the small datasets, it fails to terminate for the larger ones. While HoloClean's runtime is higher than that of competing methods, the accuracy improvements obtained justify the overhead.

6.3 Micro-benchmark Results

We evaluate the tradeoff between the runtime of HoloClean and the quality of repairs obtained by it due to the optimizations in Section 5. We also evaluate the quality of data repairs performed by HoloClean when external dictionaries are incorporated and the impact of error detection on HoloClean's performance.

6.3.1 *Tradeoffs Between Scalability and Quality*

We evaluate the runtime-quality tradeoffs for: (i) pruning the domain of random variables, which restricts the domain of the random variables in HoloClean's model, (ii) partitioning, and (iii) relaxing the denial constraints to features that encode priors. Domain pruning can be applied together with the other two optimizations, thus, is applicable to all variations of HoloClean listed next:

- **DC Factors**: Denial constraints are encoded as factors (see Section 4). No other optimization is used.
- **DC Factors + partitioning**: Same as the above variation with partitioning (see Section 5.1.2).
- **DC Feats**: Denial constraints are used to extract features that encode priors over independent random variables (see Section 5.2). This version of HoloClean was used for the experiments in Section 6.2.
- **DC Feats + DC Factors**: We use denial constraints to extract features that consider only the initial values of the values in D and also add factors that enforce denial constraints for any assignment of the cell random variables.
- **DC Feats + DC Factors + partitioning**: Same as the above variation with partitioning.

The Effect of Domain Pruning. First, we consider the DC Feats variation of HoloClean and vary threshold τ. We examine how the precision and recall of HoloClean's repairs change. The results are shown in Figure 3. Increasing threshold τ in Algorithm 2 introduces a *tradeoff between the precision and recall* achieved by HoloClean. Lower values of threshold τ provide HoloClean with an increased search space of possible repairs, thus, allowing the recall of HoloClean to be higher.

As we increase threshold τ the recall of HoloClean's output drops significantly. For example, in Food increasing the pruning threshold from 0.5 to 0.7 has a dramatic effect on recall, which drops

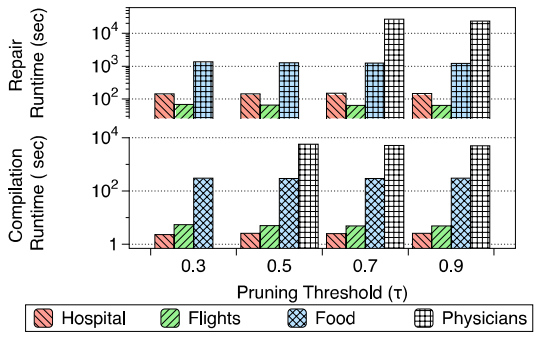

Figure 4: Effect of pruning on Compilation and Repairing runtimes. Runtimes are reported in log-scale. Missing values correspond to time-outs with a threshold of one day.

from 0.77 to 0.36. On the other hand, we see that precision increases. One exception is the Flights dataset where a large number of the pruning threshold has a negative impact on precision. This result is expected since Flights contains a small number of duplicates: Setting $\tau = 0.9$ requires that a candidate value for a cell has high co-occurrence probability with the values that other cells in the same tuple obtain. In the absence of noisy duplicates, severe pruning can lead to a set of candidate assignments that may not contain the truly correct value for an erroneous cell.

The effect of the pruning threshold τ on the runtime of HoloClean is shown in Figure 4. Violation detection is not affected by this threshold, thus, we focus on the compile and repair phase. The corresponding runtimes are in log-scale due to account for dataset differences. As shown the effect of τ is not that significant on the runtime of HoloClean. Compilation runtime is similar as τ varies. However, the time required for repairing decreases as threshold τ increases and this allows HoloClean to perform accurate repairs to large datasets such as Physicians (containing $37M$ cells).

Takeaways. Our domain pruning strategy plays a key role in achieving highly accurate repairs and allows HoloClean to scale to large datasets with millions of rows.

Runtime versus Quality Tradeoff. We now evaluate the runtime, precision, and recall for all variations of HoloClean listed above. Figure 5 reports the results for Food. The same findings hold for all datasets. We make the following observations:

(1) Runtime: When random variables are allowed to have large domains (i.e., for small values of τ) using partitioning or relaxing denial constraints to features (DC Feats) lead to runtime improvements of up to 2x. When the domain of random variables is heavily pruned, all variants of HoloClean exhibit comparable runtimes. This is expected as the underlying inference engine relies on database optimizations, such as indexing, to perform grounding. Not surprisingly, encoding denial constraints as factors (DC Factors) instead of features (DC Feats) exhibits a better runtime. This is because the model for DC Factors contains fewer factors—recall that relaxing denial constraints to features introduces a separate factor for each attribute predicate in a constraint. While one would expect partitioning to have a significant impact on the time required to perform grounding, we find that limiting the number of possible repairs per records is more effective at speeding-up grounding. The reason is that modern inference engines leverage database optimizations such as indexing during grounding.

(2) Quality of Repairs: Pruning the domain of random variables leads to an increase in the precision and a decrease in the recall of repairs obtained for the different variants of HoloClean. An interesting observation is that relaxing denial constraints (e..g, when DC Feats is used), allows HoloClean to obtain higher quality repairs. We conjecture that this is due to two reasons: (i) the fact that the input noisy datasets are statistically close to their true clean versions, i.e., the noise is limited, and (ii) when the domain of random variables is misspecified (e.g., too large) using a complex model that enforces denial constraints leads to harder, ill-posed inference problems. A theoretical study of when encoding denial constraints as features is sufficient to obtain high quality repairs is an exciting future direction of research.

Takeaways. Relaxing denial constraints leads to more scalable models and models that obtain higher quality repairs when the domains of random variables are misspecified.

6.3.2 External Dictionaries in HoloClean

We evaluate the performance of HoloClean when incorporating external dictionaries and use matching dependencies. We use the same dictionary used for KATARA. The dictionary contains a list of Zip codes, cities, and states in the US. We find that using external dictionaries can improve the quality of repairs obtained by HoloClean but the benefits are limited: for all datasets we observed F1-score improvements of less than 1%. This restricted gain is not a limitation of HoloClean, which can natively support external data, but is due to the limited coverage of the external data used.

6.3.3 Qualitative Analysis on Real-Data

We perform a qualitative analysis to highlight how the marginal probabilities output by HoloClean allow users to reason about the validity of different repairs obtained by HoloClean, thus, obviating the need for exploration strategies based on active learning. We conduct the following experiment: we consider repairs suggested by HoloClean and measure the error-rate (i.e., the rate of correct versus total repairs) for repairs in different buckets of marginal probabilities. We use the same setup as in Section 6.2.1.

As shown in Figure 6, the error-rate rate decreases as the marginal probabilities increase. For example, repairs whose marginals belong in the $[0.5 - 0.6)$ probability bucket exhibit an average error rate of 0.58 across all datasets, while marginals in $[0.7-0.8)$ bucket have an average error rate of 0.24. These marginal probabilities can be used to control the quality of repairs by HoloClean.

6.3.4 Impact of Error Detection on HoloClean

Finally, we study the impact of error detection on the quality of HoloClean's output. To evaluate the quality of repairs performed by HoloClean we use the precision and recall metrics introduced in Section 6.1. Recall, as defined previously, is computed with respect to all errors in the dataset. Nonetheless, HoloClean is restricted to perform repairs only for cells that are identified as potentially erroneous by the error detection mechanism used before HoloClean is applied for data repairing. Naturally, recall is sensitive to error detection, since an undetected error will never be fixed by HoloClean. To better understand the performance of HoloClean we also evaluate a new recall metric that we refer to as *repairing recall*. We define repairing recall as the fraction of correct repairs over the total number of correctly erroneous cells identified by error detection.

We focus on the Hospital dataset for which all errors are known. We use HoloClean with DC Feats and a value of $\tau = 0.3$ and evaluate the quality of repairs as we vary the *true positive rate* (which relates to the false negatives of the detection tool, i.e., the missed errors) and *false positive rate* (i.e., cells that are declared errors but they are correct) of error detection. The results are shown in Figure 7. To study the effect of arbitrary error detection mechanisms,

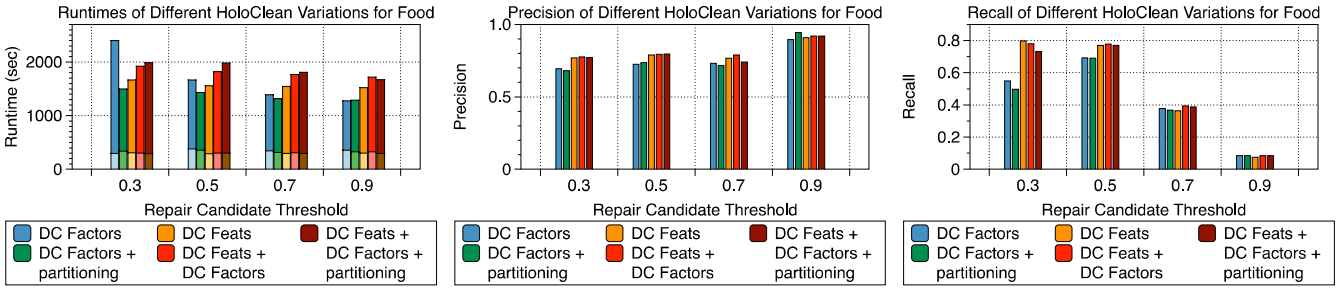

Figure 5: Runtime, precision, and recall for all variations of HoloClean on Food. For runtime, the lower part of the stacked bars corresponds to compilation time while the upper part to the runtime required for learning and inference.

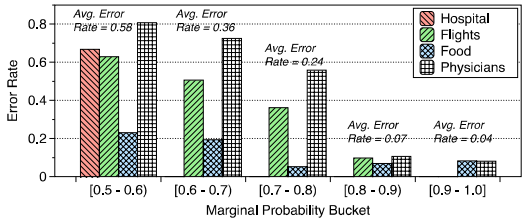

Figure 6: The error-rate of HoloClean repairs for different probability buckets for different datasets.

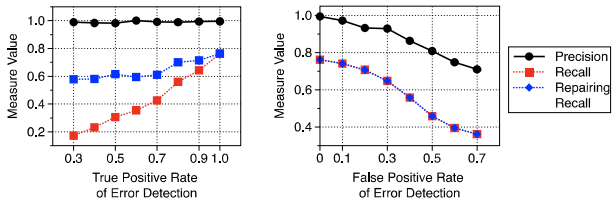

Figure 7: Impact of error detection on the quality of repairs output by HoloClean as we vary (a) the true positive rate and (b) the false positive rate of error detection.

we emulate error detection as follows: For our first experiment (see Figure 7(a)), we start with the set of all, truly erroneous cells and vary the percentage of those revealed to HoloClean—this corresponds to varying the true positive rate of error detection. For the second experiment (see Figure 7(b)), we start with the set of all, truly erroneous cells and extend it by adding correct cells as errors. For the latter, we iterate over each correct cell in the dataset and randomly add it to the set of erroneous cells with a fixed probability that is equal to the false positive rate of error detection.

As shown in Figure 7(a), when we vary only the true rate of error detection while keeping its false positive rate to zero only the recall of HoloClean is affected. This is because HoloClean is limited to repairing cells that are only detected as erroneous. On the other hand, the precision of repairs performed by HoloClean remains close to one. Finally, as expected, the repairing recall of HoloClean remains fairly stable and is always significantly higher than the overall recall. This verifies that HoloClean's performance is restricted by the quality of the error detection algorithm used.

When we vary the false positive rate of error detection while the true positive rate is set to one both the precision and recall of HoloClean are affected (see Figure 7(b)). This is because imprecise error detection affects the quality of training data available to HoloClean for learning the parameters of its underlying probabilistic model. Nonetheless, we find that the quality of repairs output by HoloClean is quite robust. In particular, the drop in precision and recall is around 7% even when cells are incorrectly reported as erroneous 20% of the time. Finally, as expected, repairing recall and the overall recall have exactly the same value.

7. RELATED WORK

Data Cleaning. There has been a significant amount of work on methods for data cleaning, including approaches that rely on integrity constraints [15, 17, 22, 26], methods that leverage external information, such as data obtained by experts [25, 47] or data in existing knowledge bases or dictionaries [16, 25, 32, 48], and techniques that focus on outlier detection [29]. All these methods rely on isolated signals to repair erroneous cells. On the other hand, HoloClean builds upon probabilistic graphical models to combine domain knowledge, such as qualitative constraints, with evidence—either in the form of external data or statistical properties of the dataset to be cleaned—to retrieve more accurate data cleaning solutions. Hence, HoloClean introduces an extensible approach for combining heterogeneous data cleaning methods.

Scalable Probabilistic Inference. Recent works have introduced general-purpose systems for specifying probabilistic models and running probabilistic inference [6, 41, 46]. In these engines, users can declaratively define probabilistic models that not only capture the uncertainly of their data but also model relevant domain-specific constraints. To scale up inference these engines rely on approximate inference methods, such as Gibbs sampling [50]. HoloClean is a compiler that automatically generates probabilistic programs for data cleaning. HoloClean guarantees the scalability of inference by optimizing the structure of the generated program for cleaning.

8. CONCLUSIONS

We introduced HoloClean, a data cleaning system that relies on statistical learning and inference to unify a range of data repairing methods under a common framework. We introduced several optimization to scale inference for data repairing, and studied the tradeoffs between the quality of repairs and runtime of HoloClean. We showed that HoloClean obtains repairs that are significantly more accurate than state-of-the-art data cleaning methods.

Our study introduces several future research directions. Understanding when integrity constraints need to be enforced versus when it is sufficient to encode them as features has the potential to generate a new family of data repairing tools that not only scale to large instances but also come with theoretical guarantees. Additionally, data cleaning is limited by the error detection methods used before. Recently, the paradigm of data programming [43] has been introduced to allow users to encode domain knowledge in inference tasks. Exploring how data programming and data cleaning can be unified under a common probabilistic framework to perform better detection and repairing is a promising future direction.

9. ACKNOWLEDGEMENTS

The authors would like to thank the members of the Hazy Group for their feedback and help. We would like to thank Intel, Toshiba, the Moore Foundation, and Thomson Reuters for their support, along with NSERC through a Discovery Grant and DARPA through MEMEX (FA8750-14-2-0240), SIMPLEX (N66001-15-C-4043), and XDATA (FA8750-12-2-0335) programs, and the Office of Naval Research (N000141210041 and N000141310129). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, ONR, or the U.S. government.

10. REFERENCES

[1] https://data.cityofchicago.org.

[2] https://alchemy.cs.washington.edu/.

[3] https://data.medicare.gov/data/physician-compare.

[4] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *Proc. VLDB Endow.*, 9(12):993–1004, Aug. 2016.

[5] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT*, pages 31–41, 2009.

[6] S. H. Bach, M. Broecheler, B. Huang, and L. Getoor. Hinge-loss markov random fields and probabilistic soft logic. *CoRR*, abs/1505.04406, 2015.

[7] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*, pages 268–279, 2011.

[8] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1-2):197–207, 2010.

[9] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *ICDE*, pages 541–552, 2013.

[10] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154. ACM, 2005.

[11] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[12] V. Chandrasekaran and M. I. Jordan. Computational and statistical tradeoffs via convex relaxation. *PNAS*, 110(13):1181–1190, 2013.

[13] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1):90–121, 2005.

[14] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[15] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[16] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, pages 1247–1261, 2015.

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

[18] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, pages 541–552, 2013.

[19] K. Das and J. Schneider. Detecting anomalous records in categorical datasets. KDD, pages 220–229, 2007.

[20] S. Ermon, C. P. Gomes, and B. Selman. Uniform solution sampling using a constraint solver as an oracle. In *UAI*, pages 255–264, 2012.

[21] R. Fagin, B. Kimelfeld, and P. G. Kolaitis. Dichotomies in the complexity of preferred repairs. PODS, pages 3–15, 2015.

[22] W. Fan. Dependencies revisited for improving data quality. In *SIGMOD*, pages 159–170, 2008.

[23] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. 2012.

[24] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2(1):407–418, 2009.

[25] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *PVLDB*, 3(1-2):173–184, 2010.

[26] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The llunatic data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.

[27] L. Getoor and A. Machanavajjhala. Entity resolution: theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.

[28] T. P. Hayes and A. Sinclair. A general lower bound for mixing of single-site dynamics on graphs. *The Annals of Applied Probability*, 17(3):931–952, 2007.

[29] J. M. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[30] I. F. Ilyas. Effective data cleaning with continuous evaluation. *IEEE Data Eng. Bull.*, 39:38–46, 2016.

[31] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[32] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, 2015.

[33] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

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

[35] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[36] N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In *ICDE*, pages 1275–1278, 2009.

[37] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, pages 802–803, 2006.

[38] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava. Truth finding on the deep web: Is the problem solved? PVLDB, pages 97–108, 2013.

[39] C. Mayfield, J. Neville, and S. Prabhakar. Eracer: A database approach for statistical inference and data cleaning. SIGMOD, pages 75–86, 2010.

[40] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. 2010.

[41] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *Proc. VLDB Endow.*, 4(6):373–384, Mar. 2011.

[42] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[43] A. J. Ratner, C. D. Sa, S. Wu, D. Selsam, and C. Ré. Data programming: Creating large training sets, quickly. In *NIPS*, pages 3567–3575, 2016.

[44] T. Rekatsinas, M. Joglekar, H. Garcia-Molina, A. G. Parameswaran, and C. Ré. SLiMFast: Guaranteed results for data fusion and source reliability. In *SIGMOD*, 2017.

[45] C. D. Sa, C. Zhang, K. Olukotun, and C. Ré. Rapidly mixing gibbs sampling for a class of factor graphs using hierarchy width. In *NIPS*, pages 3097–3105, 2015.

[46] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using DeepDive. *Proc. VLDB Endow.*, 8(11):1310–1321, July 2015.

[47] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[48] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, pages 457–468. ACM, 2014.

[49] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*, pages 553–564, 2013.

[50] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, pages 1283–1294, 2014.