PClean: Bayesian Data Cleaning at Scale with Domain-Specific Probabilistic Programming—Supplementary Materials

A Baseline Inference Algorithms

The paper's Figure 6 shows median accuracy vs. time for five independent runs of nine inference algorithms. These results were computed using the PClean program shown in Appendix B.4.1, on a version of the *Hospital* dataset (Appendix B.1) with 20% of its cells deleted at random, to test both repair and imputation (the original *Hospital* dataset has many errors, but very few missing cells). Below, we give descriptions of each inference algorithm we test:

1. **PClean SMC (2 particles) followed by PClean rejuvenation** is the inference algorithm described in Section 3. First, a complete run of 2-particle sequential Monte Carlo, using PClean's enumeration-based compiled proposals, is completed, incorporating all 1000 rows of the dataset. Then, one of the two particles is selected, and for each object in its latent database, a block rejuvenation MCMC kernel is run, also using PClean's enumeration-based compiled proposal. (The number of MCMC moves completed during this sweep will depend on the number of objects inferred for the latent database, a quantity that varies from run to run. See note below this list for an explanation of how median accuracies were computed across runs with different numbers of iterations.)

2. **PClean SMC (2 particles)** is the same as the above except that no rejuvenation sweep is performed.

3. **PClean SMC (2 particles) followed by PClean rejuvenation, no subproblem hints** is the same as (1), except we disregard subproblem hints in the PClean program. (The program in question, shown in Appendix B.4.1, has two subproblem hints.) As a result, SMC takes bigger steps, and enumerative proposals take longer to execute (but are higher quality).

4. **PClean SMC (20 particles) followed by PClean rejuvenation** is the same as (1) except with 20 particles, instead of 2.

5. **PClean MCMC** initializes the latent database using ancestral sampling, i.e., from the prior, but modified to use observed values when they are available. It then performs two complete MCMC sweeps, using PClean's block rejuvenation proposals; each sweep performs an MCMC move for each object in the current latent database.

6. **Generic MCMC** initializes the latent database as in (5), and performs ten complete sweeps using *single-site* Metropolis-Hastings (tens of thousands of accept/reject steps). That is, each individual attribute or reference slot is separately updated, using the prior as proposal. When a reference slot is proposed, there is a chance that a new object is also proposed as its target. We note that our implementation is much faster than most PPLs' single-site Metropolis-Hastings implementations, as it re-evaluates only those likelihood terms affected by the proposed single-variable change.

7. **Generic SMC (100 particles) followed by generic PGibbs rejuvenation (100 particles)** initializes the latent database using 100-particle sequential Monte Carlo, using the same sequence of target distributions as in PClean SMC, but with the prior as a proposal. This is followed by three sweeps of Particle Gibbs rejuvenation moves: as in PClean rejuvenation from (1), we perform per-*object* updates, but the proposal is generated not via PClean's enumerative proposal compiler, but rather by using 100-particle conditional sequential Monte Carlo (CSMC) [like a Gibbs move, this proposal is always accepted]. We note that this baseline improves over existing PPLs' support for Particle Gibbs in several ways. First, Particle Gibbs updates only those variables connected to a particular latent object, rather than trying to update the entire model state at once. Second, incremental SMC weights are computed incrementally, evaluating only those

likelihood terms that are necessary. Third, a reweighting (and, based on ESS, possibly resampling) step is triggered whenever a new likelihood term could possibly be evaluated, regardless of how the PClean program is written. However, unlike PClean's rejuvenation moves (but like many other PPL implementations), our "generic PGibbs rejuvenation" uses proposals from the prior for its CSMC sweeps, greatly limiting its effectiveness. (We note that *delayed sampling* [Murray et al., 2018, Wigren et al., 2019] is a sophisticated PPL technique that could provide benefits similar to those provided by PClean's proposal; however, to our knowledge delayed sampling is not implemented in any PPL capable of performing SMC in PClean's model.)

8. **Generic SMC (100 particles) followed by generic rejuvenation** initializes the latent database using 100-particle sequential Monte Carlo, as in (7). It then performs five single-site Metropolis-Hastings rejuvenation sweeps (tens of thousands of accept/reject steps), as described in (6).
9. **Generic SMC (100 particles)** initializes the latent database as in (7) and performs no additional rejuvenation.

For each run of each algorithm, time and accuracy were measured after each SMC step or MCMC transition. Since steps/transitions finished at different timestamps across runs, and because each run of an algorithm lasted a different number of steps (due to the stochastic number of objects in the latent database), we used linear interpolation to approximate a continuous time/accuracy curve for each run. Then, to plot median performance across the five runs, we took the median value across the interpolated curves at a fixed set of times. In all nine algorithms, all five runs ended at roughly the same time; the plotted endpoint for each algorithm was chosen as the time when the *last* run was complete. For any run that finished slightly earlier, the accuracy value was extrapolated as the accuracy at its last timestamp.

B Evaluation on Data Cleaning Benchmarks: Datasets, Systems, and System Configurations

Table 1 of our paper provides evidence of PClean's applicability to data-cleaning problems, by comparing accuracy and runtime for three PClean programs against state-of-the-art data cleaning systems applied to the same benchmark datasets. The table reports F_1 scores, but omits the breakdown in terms of recall ($R = \frac{\text{correct repairs}}{\text{total errors}}$) and precision ($P = \frac{\text{correct repairs}}{\text{total repairs}}$), the metrics from which $F_1 = 2PR/(P+R)$ is derived. The table below presents a fuller picture:

Task	Metric	PClean	HoloClean (Unpublished)	HoloClean	NADEEF	NADEEF + Manual Java Heuristics
Flights	Prec	0.91	0.79	0.39	0.76	0.92
	Rec	0.89	0.55	0.45	0.03	0.88
	F_1	**0.90**	0.64	0.41	0.07	**0.90**
	Time	**3.1s**	45.4s	32.6s	9.1s	14.5s
Hospital	Prec	1.0	0.95	1.0	0.99	0.99
	Rec	0.83	0.85	0.71	0.73	0.73
	F_1	**0.91**	0.90	0.83	0.84	0.84
	Time	**4.5s**	1m 10s	1m 32s	27.6s	22.8s
Rents	Prec	0.68	0.83	0.83	0	0.83
	Rec	0.69	0.34	0.34	0	0.37
	F_1	**0.69**	0.48	0.48	0	0.51
	Time	1m 20s	20m 16s	13m 43s	13s	**7.2s**

The remainder of this appendix describes in detail: each benchmark dataset (Appendix B.1), each baseline system (Appendix B.2), the HoloClean and NADEEF configurations used for each baseline (emphasizing the ways in which we attempted to encode dataset-specific domain knowledge) (Appendix B.3), and the PClean programs we used for each dataset (Appendix B.4).

B.1 Description of Benchmarks

The three smaller benchmark datasets are available in the PClean repository; *Physicians* is excluded for size, but is hosted by Medicare.

Hospital is a real-world Medicare dataset, but with artificially introduced typos in approximately 5% of its 19,000 cells (1000 rows, 19 columns). Each row reports the performance of a particular hospital on a particular metric, and it includes metadata such as hospital address and phone number. This leads to a lot of duplicated information, as the same hospital appears multiple times (with different metrics), and the same metrics also appear multiple times (with different hospitals). All this duplication facilitates accurate cleaning even in the presence of typos.

Flights consists of 2,377 rows describing real-world flight, their scheduled departure/arrival times, and their true departure/arrival times, as scraped from the web. These times often conflict between the sources, so the task is to integrate them to form a consistent dataset. We use the version from [Mahdavi et al., 2019].

Rents is a new synthetic dataset of apartment listings that we derived from census and housing statistics [US Census Bureau, 2019]. It contains bedroom size, rent, county, and state. We first generated a clean dataset with 50,000 rows in the following manner:

- The county-state combination is chosen proportionally to its population in the United State
- The size of the apartment is chosen uniformly from studio, 1 bedroom, 2 bedroom, 3 bedroom, 4 bedroom.
- The rent is chosen according to a normal distribution in which the mean is the median rent for an apartment of the chosen size in the chosen country and the standard deviation is chosen to be 10% of the mean

The dataset was then dirtied in the following ways:

- 10% of state names are deleted (many counties exist across multiple states, e.g. 30 states have a Washington County).
- Approximately 1-2% of county names are misspelled
- 10% of apartment sizes are deleted
- 1% of apartment prices are listed in the incorrect units (thousands of dollars, instead of dollars)

B.2 Description of State-of-the-Art Data-Cleaning Systems

HoloClean is a data-cleaning system, which compiles user-provided *integrity constraints* and when available, external ground-truth, into a factor graph with learned weights [Rekatsinas et al., 2017]. These integrity constraints describe cells that should match, conditional on the agreement of other fields, e.g. if zip codes of two rows match, the states in those two rows should match. These constraints can also be made with respect to external data (e.g. if a row's zip code in the table matches a zip code in a gazetteer, the row's state should match the corresponding state in the gazetteer).

NADEEF is a data-cleaning system that leverages user-specified cleaning rules [Dallachiesat et al., 2013]. NADEEF compiles users' rules into a weighted MAX-SAT query and runs it through a solver, then uses the results to clean the data. User-specified rules can either be integrity constraints (as HoloClean) or handcrafted rules. These handcrafted rules take the form of Java classes, in which users write a `detect` function that takes in a pair of tuples and outputs whether one or more violations have been detected, and if so, over which groupings of cells. The user can also optionally write a `repair` function that takes in those detected cells, and returns a fix. That is, unlike in PClean, user-encoded knowledge explicitly describes how to both detect and repair violations.

To our knowledge, neither system comes with special logic for handling text fields, dates, etc. as distinct from general categorical data.

B.3 Settings for Data-Cleaning Systems

Below, we present the integrity constraints we encoded in both HoloClean and NADEEF, as well as the handcrafted Java rules for NADEEF. The integrity constraints are presented as "A determines B", which means that for two rows, if all columns in A match, one should expect all columns in B to also match.

For each NADEEF Java rule, we describe the functionality and report the number of lines of code used to encode it (ignoring imports, boilerplate, and parentheses). All integrity constraints and Java rules can also be found in the PClean repository.

On encoding domain knowledge. Data cleaning is of course easier with accurate domain knowledge about the data and the likely errors. This is one reason we developed PClean: to enable generatively encoded domain knowledge to inform a data cleaning system. This does, however, raise the question of how to compare PClean fairly to other data-cleaning systems: if PClean is more accurate only because it encodes more domain knowledge, it would be misleading to claim that PClean is 'better' in some absolute sense than an existing system. Our evaluation in Section 4 specifically explains that this is not our intention: we just mean to contextualize PClean's accuracy and runtime in the context of other data-cleaning systems, using reasonable configurations for those systems.

That said, we tried our best to encode as much helpful domain knowledge as we could into the configurations for HoloClean and NADEEF. Some of the settings below were chosen in response to direct advice from authors of each system; others were based on existing scripts, written by the system authors, for cleaning these benchmark datasets (some of our benchmarks also appeared in the papers presenting these systems). In addition, we tried tweaking these configurations ourselves, and reported the best numbers we could.

It is likely that the *approaches* that NADEEF and HoloClean take, of using weighted logic and factor graphs, could in principle express richer domain knowledge than our configurations here encode. But to our knowledge, the current *systems* do not expose these capabilities in easy-to-exploit ways.

B.3.1 Hospital

Integrity Constraints

- *Hospital Name* determines *Phone Number*, *City*, *ZIP Code*, *State*, *Address*, *Provider Number*, *County Name*, *Hospital Type*, and *Hospital Owner*.
- *Phone Number* determines *City*, *ZIP Code*, *State*, *Address1*, *Provider Number*, *County Name*, *Hospital Type*, *Hospital Owner*.
- *ZIP Code* determines *City* and *State*.
- *Measure Code* determines *Measure Name* and *Condition*.
- *Measure Code* and *State* together determine *State Average*.

Java Rules

The *State Average* field is a concatenation of the *Measure Code* and *State* fields. For any row, we raise a violation if the concatenation does not hold over those three cells. We do not provide a repair, since it's unclear from that row alone which of the three cells is the incorrect one. This took 9 lines of Java code.

B.3.2 Flights

Integrity Constraints

- *Flight* number determines both the *Scheduled Departure Time* and the *Actual Departure Time*
- *Flight* number determines both the *Scheduled Departure Time* and the *Actual Departure Time*

Java Rules

For a pair of rows, if both flights have the same flight number, a violation is already raised by the existing integrity constraints if the departure or arrival time does not match. The source corresponding to the flight's airline tends to more correct than third-party sources. Therefore, when applicable over a pair of rows, we provided the suggested repair of choosing the time from the website of the airline. This took 52 lines of Java code.

B.3.3 Rent

Integrity Constraints

County determines *State*.

Java Rules

If a state was missing for a rental listing, we suggested that NADEEF choose the repair of the most common state corresponding to a given county (which it would not otherwise do), requiring 48 lines of Java.

Additionally, if a rent was below a certain fixed threshold, the program would flag as a violation, and multiply by the correct factor for a unit conversion. This second rule required 12 lines of Java.

B.3.4 Physician

Integrity Constraints

- The *National Provider Identifier (NPI)* determines the *PAC ID* and vice versa.
- The *National Provider Identifier (NPI)* determines *First Name*, *Last Name*, *Medical School Name*, and *Graduation Year*.
- The *Group Practice ID* determines the *Organization name*.
- The *Zip Code* determines the *City* and *State*.

B.4 PClean Programs

In this section, we present the PClean programs we used to clean each benchmark dataset. This is the closest analogue to a 'configuration' of an automated data-cleaning system. But rather than encode rules for detecting and repairing errors, PClean programs encode generative models of relational databases and of the process by which they are corrupted, filtered, and joined to yield flat, dirty, denormalized datasets.

B.4.1 Hospital

The *Hospital* dataset is modeled with seven classes: `Record`s reflect typo'd attributes of `Hospital`s and the `Measure`s by which they are evaluated; `Hospital`s have `HospitalType`s and are located in `Place`s; `Place`s belong to `County` objects; and each `Measure` is related to some `Condition`. Typos are modeled as independently introduced for each cell of the dataset. Some fields are modeled as draws from broad priors over strings, whereas others are modeled as categorical draws whose domain is the set of unique observed values in the relevant column (some of which are in fact typos).

Inference hints are used to focus proposals for `string_prior` choices on the set of strings that have actually been observed in a given column, and also to set a custom subproblem decomposition for the `Record` class (all other classes use the default decomposition).

```
latent class County
  parameter state_proportions ~ dirichlet(ones(num_states))
  state ~ discrete(observed_values[:State], state_proportions)
  county ~ string_prior(3, 30) preferring observed_values[:CountyName]
end
latent class Place
  county ~ County
  city ~ string_prior(3, 30) preferring observed_values[:City]
end
latent class Condition
  desc ~ string_prior(5, 35) preferring observed_values[:Condition]
end
latent class Measure
  code ~ uniform(observed_values[:MeasureCode])
  name ~ uniform(observed_values[:MeasureName])
  condition ~ Condition
end
latent class HospitalType
  desc ~ string_prior(10, 30) preferring observed_values[:HospitalType]
end
```

```
latent class Hospital
  parameter owner_dist ~ dirichlet(ones(num_owners))
  parameter service_dist ~ dirichlet(ones(num_services))
  loc ~ Place
  type ~ HospitalType
  id ~ uniform(observed_values[:ProviderNumber])
  name ~ string_prior(3, 50) preferring observed_values[:HospitalName]
  addr ~ string_prior(10, 30) preferring observed_values[:Address1]
  phone ~ string_prior(10, 10) preferring observed_values[:PhoneNumber]
  owner ~ discrete(observed_values[:HospitalOwner], owner_dist)
  zip ~ uniform(observed_values[:ZipCode])
  service ~ discrete(observed_values[:EmergencyService], service_dist)
end
latent class Record
  subproblem begin
    hosp   ~ Hospital;                     service ~ typos(hosp.service)
    id     ~ typos(hosp.id);               name    ~ typos(hosp.name)
    addr   ~ typos(hosp.addr);             city    ~ typos(hosp.loc.city)
    state  ~ typos(hosp.loc.county.state); zip     ~ typos(hosp.zip)
    county ~ typos(hosp.loc.county.county); phone  ~ typos(hosp.phone)
    type   ~ typos(hosp.type.desc);        owner   ~ typos(hosp.owner)
  end
  subproblem begin
    metric ~ Measure
    code ~ typos(metric.code); mname ~ typos(metric.name);
    condition ~ typos(metric.condition.desc)
    stateavg = "$(hosp.loc.county.state)_$(metric.code)"
    stateavg_obs ~ typos(stateavg)
  end
end
```

B.4.2 Flights

The model for *Flights* uses three classes: each observed `Record` comes from a `TrackingWebsite` and is about a `Flight`:

```
latent class TrackingWebsite
  name ~ string_prior(2, 30) preferring observed_values[:website]
end
latent class Flight
  flight_id ~ string_prior(10, 20) preferring flight_ids; index on flight_id
  sdt ~ time_prior() preferring observed_values["$flight_id-sched_dep_time"]
  sat ~ time_prior() preferring observed_values["$flight_id-sched_arr_time"]
  adt ~ time_prior() preferring observed_values["$flight_id-act_dep_time"]
  aat ~ time_prior() preferring observed_values["$flight_id-act_arr_time"]
end
latent class Record
  parameter error_probs[_] ~ beta(10, 50)
  flight ~ Flight; src ~ TrackingWebsite
  error_prob = lowercase(src.name) == lowercase(flight.flight_id[1:2]) ? 1e-5 : error_probs[src.name]
  sdt ~ maybe_swap(flight.sdt, observed_values["$(flight.flight_id)-sched_dep_time"], error_prob)
  sat ~ maybe_swap(flight.sat, observed_values["$(flight.flight_id)-sched_arr_time"], error_prob)
  adt ~ maybe_swap(flight.adt, observed_values["$(flight.flight_id)-act_dep_time"],   error_prob)
  aat ~ maybe_swap(flight.aat, observed_values["$(flight.flight_id)-act_arr_time"],   error_prob)
end
```

In the **parameter** declaration for *error_probs*, we use the syntax `error_probs[_] ~ beta(10, 50)` to introduce a *collection* of parameters; the declared variable becomes a dictionary, and each time it is used with a new index, a new parameter is instantiated. We use this to learn a different `error_prob` parameter for each tracking website. We could alternatively declare `error_prob` as an attribute of the `TrackingWebsite` class. However, PClean's inference engine uses smarter proposals for declared **parameter**s (taking advantage of conjugacy relationships), so for our experiments, we use the **parameter** declaration instead. We hope to extend automatic conjugacy detection to all attributes, not just parameters, in the near future.

As in *Hospital*, we use `observed_values` to provide inference hints to the broad `time_prior`; this expresses a belief that the true timestamp for a certain field is likely one of the timestamps that has actually been observed, in the dirty dataset, with the given flight ID.

B.4.3 Rents

The program we use for *Rents* contains two classes: `Listing`s are for apartments in some `County`:

```
data_table.block = map(x -> "$(x[1])$(x[end])", data_table.County)
units = [Transformation(identity, identity, x -> 1.0),
```

```
        Transformation(x -> x/1000.0, x -> x*1000.0, x -> 1/1000.0)]
latent class County
  parameter state_pops ~ dirichlet(ones(num_states))
  block ~ unmodeled(); index by block
  name ~ string_prior(10, 35) preferring observed_values[block]
  state ~ discrete(states, state_pops)
end
latent class Listing
  parameter avg_rent[_] ~ normal(1500, 1000)
  subproblem begin
    county ~ County
    county_name ~ typos(county.name, 2)
    br ~ uniform(room_types)
    unit ~ uniform(units)
    rent_base = avg_rent["$(county.state)_$(county.name)_$(br)"]
    observed_rent ~ transformed_normal(rent_base, 150.0, unit)
  end
  rent = round(unit.backward(observed_rent))
end
```

We model the fact that the rent may be in *grand* instead of *dollars*, as well as that the county name may contain typos. We introduce an artificial field, *block*, consisting of the first and last letters of the observed (possibly erroneous) County field, and use it to inform an inference hint: we hint that posterior mass for a county's name concentrates on those strings observed somewhere in the dataset that share a first and last letter in common with the observed county name for this row. Without this approximation, inference is much slower (but potentially more accurate).

B.4.4 Physicians

The model for *Physicians* contains five classes: `Record`s reference `Practice`s and `Physician`s; each `Physician` attended some medical `School`; and each `Practice` is in a `City`:

```
latent class School
  name ~ unmodeled(); index by name
end

latent class Physician
  parameter error_prob ~ beta(1.0, 1000.0)
  parameter degree_proportions[_] ~ dirichlet(3 * ones(num_degrees))
  parameter specialty_proportions[_] ~ dirichlet(3 * ones(num_specialties))
  npi ~ number_code_prior()
  school ~ School
  subproblem begin
    degree ~ discrete(observed_values[:Credential], degree_proportions[school.name])
    specialty ~ discrete(observed_values[Symbol("Primary specialty")], specialty_proportions[degree])
    degree_obs ~ maybe_swap(degree, observed_values[:Credential], error_prob)
  end
end

latent class City
  c2z3 ~ unmodeled(); index by c2z3
  name ~ string_prior(3, 30) preferring cities[c2z3]
end

latent class Practice
  addr ~ unmodeled(); index by addr
  addr2 ~ unmodeled(); index by addr2
  zip ~ string_prior(3, 10); index by zip
  legal_name ~ unmodeled(); index by legal_name
  subproblem begin
    city ~ City
    city_name ~ typos(city.name)
  end
end

latent class Record
  physician ~ Physician
  address ~ Practice
end
```

Many columns are not modeled. Similar to *Rents*, we use a **parameter** in the `Physician` class for `degree_probs`, although it might seem more natural to use an attribute of the `School` class; the resulting model is the same, but using **parameter** allows PClean to exploit conjugacy.

B.5 Effect of Additional Domain Knowledge

The quality of PClean's inference depends on the PClean program one uses to model the data. To demonstrate this, we apply four different PClean programs on *Flights*. In our baseline (16 lines of code), we assume all sources are equally reliable and achieve an F1 score of 0.56. By additionally modeling the timestamp format, we achieve an F1 of 0.60. If we program PClean to learn a per-source reliability (one extra line of code), F1 climbs to 0.69. Finally, if we provide our program that the airline's own website is likely to be the most reliable for a given flight (one additional line of code for a total of 18), F1 jumps to 0.90.

We also implemented a user-defined cleaning rule in NADEEF, manually specifying a repair procedure for flight times that searched for a reported time from the flight's airline, and used that if available. This rule enabled NADEEF to clean the *Flights* data, but required 52 lines of Java (beyond the boilerplate required for every NADEEF rule). Furthermore, as Table 1 of the paper shows, even encoding manual Java rules is, for some datasets, not enough to yield accurate cleaning.

C Additional Model Details

C.1 Discrete Random Measure representation

Our non-parametric structure prior $p(\mathbf{S})$ is described by Section 2 of the paper in terms of the two-parameter Chinese Restaurant Process. It is also possible to represent the generative process encoded by a PClean program using the Pitman-Yor process:

GenerateDataset():
  for latent class $C \in$ TopologicalSort($\mathcal{C}$) **do**
    $\theta_C \sim p_{\theta_C}()$
    $G_C \sim$ GenerateCollection$(C, \theta_C, \{G_{C'}\}_{C' \in Pa(C)})$
  $\theta_{\mathbf{Obs}} \sim p_{\theta_{\mathbf{Obs}}}()$
  for $i \in \{1, \ldots, n\}$ **do**
    $r_i \sim$ GenerateObject$(C_{Obs}, \theta_{C_{Obs}}, \{G_C\}_{C \in Pa(C_{Obs})})$

GenerateCollection$(C, \theta_C, \{G_{C'}\}_{C' \in Pa(C)})$:
  $s_C \sim Gamma(1, 1)$
  $d_C \sim Beta(1, 1)$
  $G_C \sim PY(s_C, d_C,$ GenerateObject$(C, \theta_C, \{G_{C'}\}_{C' \in Pa(C)}))$

GenerateObject$(C, \theta_C, \{G_{C'}\}_{C' \in Pa(C)})$:
  for reference slot $Y \in \mathcal{R}(C)$ **do**
    $r.Y \sim G_{T(C.Y)}$
  for attribute $X \in \mathcal{A}(C)$ **do**
    $r.X \sim \phi_{C.X}(\theta_C, \{r.\tau\}_{\tau \in Pa(C.X)})$

We process classes one at a time, in topological order. For each latent class, we (1) generate class-wide hyperparameters θ_C from their corresponding hyperpriors, and (2) generate an *infinite weighted collection* of *objects* of class C. In this setting, an *object r of class C* is an assignment of each attribute $C.X$ to a value $r.X$ and of each reference slot $C.Y$ to an object $r.Y$ of class $T(C.Y)$. An infinite collection of latent objects is generated via a Pitman-Yor Process [Teh and Jordan, 2011]:

$$G_C \sim PY(s_C, d_C, \text{GenerateObject}(C, \theta_C, \{G_{C'}\}_{C' \in Pa(C)}))$$

The Pitman-Yor Process is a discrete random measure that generalizes the Dirichlet Process. It can be understood as first sampling an infinite vector of probabilities $\rho \sim GEM(s_C, d_C)$ from a two-parameter GEM distribution, then setting $G_C = \sum_{i=1}^{\infty} \rho_i \delta_{r_i^C}$, where each of the infinitely many objects r_i^C is distributed according to GenerateObject$(C, \theta_C, \{G_{C'}\}_{C' \in Pa(C)})$. This itself is a distribution over *objects*, which first samples reference slots and then attributes.

To generate the objects of the observation class, which will be translated by the program's query into the flat dataset $\mathbf{D}$, we sample $\theta_{C_{Obs}}$ from its prior distribution, then, for $i \in \{1, \ldots, n\}$, generate the i^{th} observed entry: $r_i \sim \textsc{GenerateObject}(C_{Obs}, \theta_{C_{Obs}}, \{G_C\}_{C \in Pa(C_{Obs})})$.

C.2 Description of primitive distributions

Our models for particular datasets make use of PClean's built-in probability distributions, which include not just the common distributions for categorical and numerical data, but also several domain-specific distributions useful for modeling strings and random errors. We briefly summarize several of PClean's built-in distributions here, before showing how to compose them into short PClean programs:

- `string_prior(min, max)` encodes a prior over strings between *min* and *max* characters long. The length is uniformly distributed within that range, and characters follow a Markov model based on relative character bigram frequencies in English.
- `typos(str)` is a distribution over strings centered at `str`. The generative process it represents is to sample a number of typos from a negative binomial distribution whose number-of-trials parameter depends on the length of `str`. That many typos (random insertions, deletions, substitutions, or transpositions) are then performed. The likelihood is computed approximately using dynamic programming.
- `maybe_swap(x, ys, p)` returns a true value `x` with probability $1 - \texttt{p}$, but chooses a replacement uniformly from `ys` otherwise.
- `transformed_normal(mean, std, bijection)` samples a real number from a Gaussian distribution with the given mean and standard deviation, but then applies a transformation (the bijection). We use this distribution to model unit errors.

C.3 Discussion of expressiveness of PClean

PClean imposes restrictions relative to universal PPLs, which helped us to develop an inference algorithm that, for many PClean programs, produces results quickly and scales to large datasets. In this section, we discuss these restrictions and their implications for cleaning dirty data using PClean.

Our non-parametric prior vs. explicit user-specified priors over number of objects and link structure. A primary difference between general-purpose open-universe languages, like BLOG, and PClean's modeling language is that PClean does not give the user control over the prior distribution over the number of objects of each class, or which objects of particular classes are related to one another.[1] Instead, it imposes a domain-general non-parametric prior. This limitation might be mitigated by (1) the use of strength and discount hyperparameters of the Chinese Restaurant Process to control the prior expected size of each class (for a particular amount of data), and (2) the fact that in many data-cleaning applications, accurate prior knowledge about the number of objects may not be unavailable, or else is not a deciding factor in making cleaning judgments.

Of course, there are exceptions. As an interesting example, consider the Hospital dataset: if we knew the population of each city, we may have been able to specify accurate priors over the number of distinct hospitals in each city, allowing us to resolve co-reference questions differently in small cities (where it is more likely that two hospitals reported with similar names are in fact the same hospital) and large cities (where it may be more plausible that two hospitals exist with very similar names). However, this factor is likely to be decisive only in high-uncertainty regimes (where the data entries themselves do not help much to resolve the co-reference

[1] However, note that BLOG also has limitations when it comes to expressing priors over link structure. It allows users to specify predicates that the targets of a reference slot must satisfy, and the choice is then assumed to be uniform among all objects satisfying the predicate. Thus, BLOG cannot express that certain objects are more "popular" targets of reference slots than others—an assumption that is built in to PClean's Pitman-Yor-based model. We also note that by introducing additional classes, PClean can represent more interesting priors over link structure. For example, suppose $A.Y$ and $B.Y$ are two reference slots to objects from C, and we wish each reference slot to be filled using different distributions over the objects in C. We can create dummy classes for each reference slot, AC and BC, each with a single reference slot ($AC.Y$ and $BC.Y$) to the target class C. We then have the reference slots $A.Y$ and $B.Y$ target AC and BC respectively, instead of directly targeting C. This implements a hierarchical Pitman-Yor process; by analogy with the HDP-LDA topic model, objects of A and B play the role of words from two different documents, and objects of class C are the topics.

question), and it is unclear whether a data-cleaning system should trust such high-uncertainty answers (vs. reporting 'I don't know'—see Appendix **??**). If the use case is such that it *is* desirable to represent such priors, similar logic might be encoded in PClean by creating two different classes for hospitals in large and small cities, and allowing their strength and discount parameters to vary independently.

On schemas with cyclic vs. acyclic class dependency graphs. PClean requires that the schema of the latent database have an acyclic class dependency graph: there cannot be a chain of reference slots K such that $T(C.K) = C$. Although, generally speaking, many relational modeling and inference tasks may be well-served by cyclic class dependencies, we found during literature review that none of the benchmark data-cleaning problems in [Abedjan et al., 2016, Dallachiesat et al., 2013, Rekatsinas et al., 2017, Heidari et al., 2019, Hu et al., 2012, Mahdavi et al., 2019] were naturally modeled using cyclic class dependencies. In addition, [Pasula et al., 2003, Milch and Russell, 2006], who use BLOG for deduplication, do not use its support for reference cycles. There are, of course, some tasks for which cyclic references may be a natural fit, e.g. denoising genealogical data, where we may want to model that people have parents, who are other people, with many attributes inherited from one's ancestors. One could still model such datasets using coarser PClean models, e.g., by clustering people into families without modeling parent/child relationships explicitly. More generally, when we wish to model objects of the same class C (e.g. *Person*) as related via some chain of reference slots, we can often instead introduce an additional class C' (e.g. *Family*), and model any related objects of class C as referring to a shared object of class C'.

D Additional Inference Details

D.1 Object-wise rejuvenation moves

In sequential Monte Carlo, rejuvenation moves are transition kernels that preserve the current target distribution π_i, similar to the kernels used in Markov chain Monte Carlo algorithms. But we do not run them until convergence, instead using them to "rejuvenate" past decisions within SMC, in light of new data.

Any valid MCMC kernel for our model is also a valid rejuvenation kernel, and in particular, Gibbs kernels—which update a single variable in the latent state according to its full conditional distribution, keeping the rest of the state fixed—are a natural choice. However, variables in a model are often correlated, and it can be difficult to escape local modes by updating them one at a time. PClean uses *object-wise blocked rejuvenation* to address this challenge. Object-wise rejuvenation moves update all attributes and reference slots of a single object r in the latent database instance $\mathbf{R}$. In doing so, these moves may also lead to the "garbage collection" of objects that are no longer connected to the observed dataset, or to the insertion of new objects as targets of r's reference slots.

Let $r \in \mathbf{R}$ be any object in a relational database instance $\mathbf{R}$. Then we define $\mathbf{R}^{-r}$, $\mathbf{D}^{-r}$, $\Delta_r^{\mathbf{R}}$, $\mathbf{R}^r$, and $\mathbf{D}^r$ as follows:

- $\mathbf{R}^{-r}$ is the partial instance obtained by erasing from $\mathbf{R}$: (1) all attribute values and reference slot assignments for the object r; (2) all attribute values of objects r' that depend on r; and (3) any objects r' only accessible from C_{obs} via slot chains that pass through r;
- $\mathbf{D}^{-r}$ is the partial dataset obtained from $\mathbf{D}$ by erasing any attribute values whose distributions depend on values no longer specified within $\mathbf{R}^{-r}$;
- $\Delta_r^{\mathbf{R}}$ is the partial instance specifying: (1) all attribute values and reference slot assignments for the object r; and (2) all objects r' not in $\mathbf{R}^{-r}$ (accessible from C_{obs} only via slot chains that pass through r), along with their attributes and reference slots;
- $\mathbf{R}^r$ is the partial instance assigning values to all object attributes that depend on r's attributes or reference slots as parents; and
- $\mathbf{D}^r$ is the partial dataset assigning any attributes of observation objects that depend on on r's attributes or reference slots as parents.

The model density then factorizes as:

$$p(\mathbf{R}, \mathbf{D}) = p(\mathbf{R}^{-r}, \mathbf{D}^{-r}) p(\Delta_r^{\mathbf{R}} \mid \mathbf{R}^{-r}) p(\mathbf{R}^r, \mathbf{D}^r \mid \Delta_r, \mathbf{R}^{-r}, \mathbf{D}^{-r}),$$

A *blocked Gibbs sweep* loops through each object $r \in \mathbf{R}$ and updates it:

$$\Delta_r^{\mathbf{R}} \sim p(\Delta_r^{\mathbf{R}} \mid \mathbf{R}^{-r}, \mathbf{D}, \mathbf{R}^r).$$

Because resimulating $\Delta_r^{\mathbf{R}}$ may delete objects from classes that are reachable from r via reference slots, we perform this sweep in reverse topological order, starting with the objects that have no reference slots, and working our way up to the observation objects. If computing the blocked Gibbs distribution is intractable, then we can further divide $\Delta_r^{\mathbf{R}}$ according to user-specified subproblem decompositions for $\mathbf{Class}(r)$, as discussed in Section 3.3 of the paper. As the user subproblems get smaller in size, the algorithm approaches ordinary one-variable-at-a-time Gibbs sampling; thus, choosing subproblems is a simple way that users can trade off between runtime and accuracy, based both on the needs of their application and the specific properties of their models or datasets.

Our rejuvenation kernels are compiled using PClean's proposal compiler, and as such, also benefit from (1) efficient enumeration strategies that take advantage of conditional independence in the variables being updated, and (2) user-specified 'preferred values' inference hints (see Section 3.3). The paper's Algorithm 1 can be adapted for rejuvenation by adding observed variables to the Bayesian network for each attribute value specified in $\mathbf{R}^r$ (that is, each attribute value that, given the current link structure, depends on a latent variable being updated). Some of the variables within $\Delta_r^{\mathbf{R}}$ may be constrained by the observed dataset $\mathbf{D}$; this will depend on the patterns of missingness in the observations that, under the current link structure, are connected in some way to the object being updated. PClean recognizes when these patterns of missingness change (due to link structure changing), and compiles new proposals as necessary.

D.2 Continuous variables and parameters

PClean allows users to include continuous variables in their models, either as parameters or attributes in class declarations. To handle these, we augment the inference algorithm in three additional ways:

1. **Gibbs rejuvenation for parameter values.** Continuous parameters θ are updated during SMC via separate Gibbs rejuvenation moves. PClean recognizes certain conjugate relationships between parameter hyperpriors and the attribute statements that use the parameters (e.g., Normal/Normal, Beta/Bernoulli, and Dirichlet/Categorical), and automatically exploits these for efficient and rejuvenation moves informed by all the relevant data. The inference engine tracks the relevant sufficient statistics as inference progresses, so these updates need not perform costly counts or summations.
2. **Mixing with the prior for proposals of continuous attributes.** Continuous attributes are handled as though they are discrete variables with 'preferred values' set to $\emptyset$. The effect of this is that the locally optimal proposal for discrete variables is first derived without regard for the latent continuous attributes being proposed as part of the same subproblem (meaning that any likelihoods that depend on latent continuous attributes are not included during enumeration); then, once discrete values have been sampled, continuous values are sampled from their prior CPDs given any of their parent values (which may have been more intelligently proposed).
3. **Particle Gibbs object-wise rejuvenation.** Because the proposals generated by technique (2) for continuous variables may be poor, Metropolis-Hastings may often reject. To improve chances of acceptance, users can enable Particle Gibbs rejuvenation, which, in order to propose an update $\Delta_r^{\mathbf{R}}$ to an object r of class C, runs *conditional SMC* on the sequence of user-defined subproblems within class C. Using Particle Gibbs, PClean can compensate for poorer proposals by sampling many weighted particles for each subproblem, which are combined into a joint proposal for the object. Note that without continuous variables, Metropolis-Hastings is generally preferred.

D.3 Optimality conditions for proposal compiler

The proposal compiler produces smart proposals by efficiently enumerating discrete variables (exploiting conditional independence) and computing only those likelihood terms that are necessary for a particular SMC or MCMC update. When all latent variables within a subproblem have finite discrete domains, and no variables have preferred values hints specified, the proposals PClean produces are locally optimal SMC proposals, as defined in [Naesseth et al., 2019], or, for MCMC, exact blocked Gibbs rejuvenation kernels. However, introducing preferred-values hints that do not completely cover the posterior mass, or using continuous attributes within the subproblem, will lead to suboptimal (but faster-to-compute) proposals.

D.4 Observation hashing

Preferred values hints can help to limit the number of possibilities enumeration must consider for attribute values, but reference slots can also pose a problem: as the sequential Monte Carlo algorithm progresses, the latent database fills up with objects that could serve as possible targets, and considering each of them can be expensive.

In many models, however, the value of a reference slot is highly constrained by observations in $\mathbf{D}$. Consider an object r of class C with reference slot Y, and let $\mathcal{W} = \{W \mid T(C_{obs}.W) = C\}$ be the set of slot chains connecting observation objects to objects of class C. Given a query map $\mathbf{Q}$, we can check if there exist any observed attributes $x \in \mathcal{A}(\mathbf{D})$ that $\mathbf{Q}$ maps to a slot chain beginning $W.Y$. For each $W \in \mathcal{W}$, let $\mathcal{K}_{W,C.Y} = \{(U, x) \mid x \in \mathcal{A}(\mathbf{D}) \wedge \mathbf{Q}(x) = W.Y.U\}$. Then the only objects of the target class $T(C.Y)$ that $r.Y$ can possibly point to are

$$\bigcap_{W \in \mathcal{W}} \bigcap_{\{i \text{ s.t. } r_i^{obs}.W = r\}} \bigcap_{(U,x) \in \mathcal{K}_W} \{r' \in \mathbf{R}_{T(C.Y)} \mid d_i.x = r'.U\}.$$

PClean can maintain, for each class, an index that maps values v_x to sets of objects r' such that $r'.U = v_x$. PClean also maintains back-pointers from objects r to the observation objects that reference them, and stores with each object r the observed attribute values $d_i.x$ that constrain it. This allows PClean to compute the set of legal target objects for a given reference slot in $O(|\mathcal{W}|)$ time, which is constant in the number of latent objects for many models. (Indexing does require memory. Users can optionally control which U values are indexed on by including **index on** U statements within class declarations.) Of course, in some models and datasets, the size of the computed set of possible target objects may still be large, necessitating enumeration. But in common cases where the vast majority of possible targets have zero likelihood, this indexing plays a key role in helping PClean to scale to large datasets.

References

[Abedjan et al., 2016] Abedjan, Z., Chu, X., Deng, D., Fernandez, R. C., Ilyas, I. F., Ouzzani, M., Papotti, P., Stonebraker, M., and Tang, N. (2016). Detecting data errors: Where are we and what needs to be done? In Proceedings of the VLDB Endowment.

[Dallachiesat et al., 2013] Dallachiesat, M., Ebaid, A., Eldawy, A., Elmagarmid, A., Ilyas, I. F., Ouzzani, M., and Tang, N. (2013). NADEEF: A commodity data cleaning system. In Proceedings of the ACM SIGMOD International Conference on Management of Data.

[Heidari et al., 2019] Heidari, A., McGrath, J., Ilyas, I. F., and Rekatsinas, T. (2019). HoloDetect: Few-shot learning for error detection. In Proceedings of the ACM SIGMOD International Conference on Management of Data.

[Hu et al., 2012] Hu, Y., De, S., Chen, Y., and Kambhampati, S. (2012). Bayesian Data Cleaning for Web Data.

[Mahdavi et al., 2019] Mahdavi, M., Madden, S., Abedjan, Z., Ouzzani, M., Tang, N., Fernandez, R. C., and Stonebraker, M. (2019). Raha: A configuration-free error detection system. In Proceedings of the ACM SIGMOD International Conference on Management of Data.

[Milch and Russell, 2006] Milch, B. and Russell, S. (2006). General-purpose MCMC inference over relational structures. Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence, UAI 2006, pages 349–358.

[Murray et al., 2018] Murray, L., Lundén, D., Kudlicka, J., Broman, D., and Schön, T. (2018). Delayed sampling and automatic rao-blackwellization of probabilistic programs. In International Conference on Artificial Intelligence and Statistics, pages 1037–1046.

[Naesseth et al., 2019] Naesseth, C. A., Lindsten, F., and Schön, T. B. (2019). Elements of sequential monte carlo. arXiv preprint arXiv:1903.04797.

[Pasula et al., 2003] Pasula, H., Marthi, B., Milch, B., Russell, S., and Shpitser, I. (2003). Identity uncertainty and citation matching. Advances in Neural Information Processing Systems.

[Rekatsinas et al., 2017] Rekatsinas, T., Chuy, X., Ilyasy, I. F., and Ré, C. (2017). HoloClean: Holistic data repairs with probabilistic inference. In Proceedings of the VLDB Endowment.

[Teh and Jordan, 2011] Teh, Y. W. and Jordan, M. I. (2011). Hierarchical Bayesian nonparametric models with applications. Bayesian Nonparametrics, pages 158–207.

[US Census Bureau, 2019] US Census Bureau (2019). County Population Totals: 2010-2019.

[Wigren et al., 2019] Wigren, A., Risuleo, R. S., Murray, L., and Lindsten, F. (2019). Parameter elimination in particle gibbs sampling. In Advances in Neural Information Processing Systems, pages 8918–8929.