In-Database Data Imputation

In-Database Data Imputation

MASSIMO PERINI, University of Edinburgh, UK
MILOS NIKOLIC, University of Edinburgh, UK

Missing data is a widespread problem in many domains, creating challenges in data analysis and decision making. Traditional techniques for dealing with missing data, such as excluding incomplete records or imputing simple estimates (e.g., mean), are computationally efficient but may introduce bias and disrupt variable relationships, leading to inaccurate analyses. Model-based imputation techniques offer a more robust solution that preserves the variability and relationships in the data, but they demand significantly more computation time, limiting their applicability to small datasets.

This work enables efficient, high-quality, and scalable data imputation within a database system using the widely used MICE method. We adapt this method to exploit computation sharing and a ring abstraction for faster model training. To impute both continuous and categorical values, we develop techniques for in-database learning of stochastic linear regression and Gaussian discriminant analysis models. Our MICE implementations in PostgreSQL and DuckDB outperform alternative MICE implementations and model-based imputation techniques by up to two orders of magnitude in terms of computation time, while maintaining high imputation quality.

CCS Concepts: • **Information systems** → **Data cleaning**; **Database query processing**; • **Computing methodologies** → *Supervised learning*.

Additional Key Words and Phrases: missing data; incomplete data; MICE; ring; factorized computation

ACM Reference Format:
Massimo Perini and Milos Nikolic. 2024. In-Database Data Imputation. *Proc. ACM Manag. Data* 2, N1 (SIGMOD), Article 70 (February 2024), 27 pages. https://doi.org/10.1145/3639326

1 INTRODUCTION

Missing data is pervasive in real-world datasets across various domains, posing significant challenges in drawing accurate and reliable conclusions from incomplete information. Missing data may introduce bias, reduce statistical power, and undermine the validity of analyses and predictive models [20, 40, 66]. Moreover, traditional machine learning techniques, typically designed to operate on complete datasets, are ill-equipped to handle missing data effectively [18, 26, 33, 67]. Therefore, handling missing data effectively is essential for the validity and robustness of data analysis.

Various techniques are commonly used to handle datasets with missing information, including disregarding records with missing data, incorporating indicator variables to capture the missing pattern, and imputing alternative values in place of the missing values. Common imputation methods involve mean imputation, which replaces missing values with the mean of the column; last observation carried forward (LOCF), which substitutes missing values with the most recent observed value [36]; and hot deck imputation, which utilizes similarity criteria within the column [4]. Although these techniques are easy to implement and computationally efficient, they offer limited

Authors' addresses: Massimo Perini, massimo.perini@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Milos Nikolic, milos.nikolic@ed.ac.uk, University of Edinburgh, Edinburgh, UK.

ACM 2836-6573/2024/2-ART70
https://doi.org/10.1145/3639326

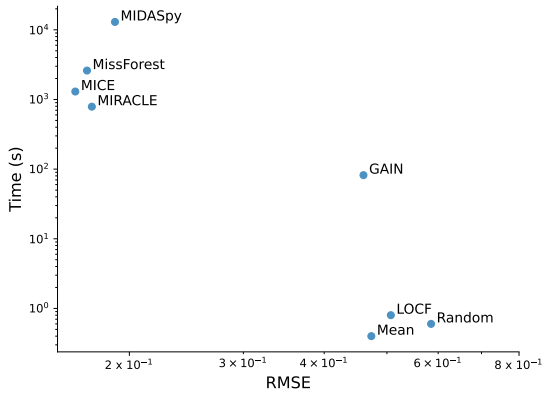


Fig. 1. Imputation quality and runtime of Python-based imputation methods on a flight dataset [51] with 5M rows and 20% missing values. Imputation quality is measured as the root mean square error (RMSE) of the linear regression model trained over an imputed dataset to predict flight duration.

guarantees regarding the quality of the imputed data. They can potentially distort value distributions, underestimate data variance, disrupt variable relationships, and introduce bias in statistical measures such as the mean, leading to subpar analytics and machine learning models over the imputed data [13, 16, 36, 66].

Model-based imputation methods can overcome these problems and capture complex missingness patterns while preserving the underlying data distribution. These methods learn models over observed data to impute missing data. Popular iterative imputation methods include MissForest [65], which utilizes random forests, and Multivariate Imputation by Chained Equations (MICE) [68], which allows using a different model for each attribute with missing values. Recent generative imputation methods exploit generative adversarial networks, like in GAIN [69], and deep learning, like in MIDASpy [37] and MIRACLE [35]. Despite their effectiveness, such model-based methods suffer from high computation costs. Figure 1 illustrates the trade-off between the quality of imputation, measured by the performance of a linear regression model trained on imputed data, and the time needed for imputation using different methods on a real dataset. Compared to model-free imputation methods such as mean and random imputation, complex methods such as MICE and MIRACLE yield models with lower root mean squared errors (RMSE), indicating higher quality, but demand orders of magnitude more time for model training and data imputation.

We next identify the challenges involved in enabling high-quality imputation on large datasets.

Challenge 1: Enabling Model-Based Imputation in DBMSs. Model-based imputation methods are commonly implemented in external tools such as scikit-learn [52] and R [68], operating independently of the DBMS environment. To impute a dataset stored in a DBMS using these methods, we need to export the dataset to the external tool for imputation and import the imputed dataset back into the DBMS for further analyses. These steps can significantly increase the computation time. Moreover, as the external tool typically lacks support for out-of-memory computation, this approach works only with datasets small enough to fit into memory.

Challenge 2: Long Imputation Time. Despite the benefits offered by model-based imputation methods, simpler techniques such as dropping rows with missing values and mean imputation continue to be prevalent in practice and are often the default option in statistical packages such as R and SAS. The reluctance to adopt more complex imputation methods primarily stems from their higher computational time and limited ability to handle large datasets [13, 20, 60]. For instance,

MissForest and MICE impute missing data by employing an iterative process of predictive modeling [20, 65], retraining models from scratch on every iteration. Data practitioners utilizing complex imputation methods may experience extended waiting periods, potentially lasting several hours, for the imputation process to converge [14]. Therefore, these complex methods are typically viable only for small datasets.

Challenge 3: Avoiding Data Explosion. Model-based imputation methods implemented in external tools can only train models on data available in a single table. Therefore, data practitioners need to preprocess the dataset before exporting it to the external tool, which may involve joining relations if the dataset is normalized and performing one-hot encoding of categorical features. These preprocessing steps introduce redundancy in both data and computation, significantly increasing the size of the training dataset and prolonging the overall computation time [15, 63, 64].

Our Goal. This work aims to enable efficient and high-quality data imputation within database systems. By moving imputation closer to data, we want to harness the performance and scalability of database systems to accelerate both model training and data imputation, effectively addressing the above challenges.

This work studies in-database data imputation using the MICE method [13, 68]. This method can impute incomplete multivariate data comprising mixtures of continuous and categorical values. For each attribute with missing values, MICE learns a model – a regressor or a classifier depending on the attribute's type – using all other attributes as predictors. The imputation proceeds one attribute at a time in a round-robin fashion, incorporating previously imputed values into the prediction models for subsequent attributes.

We focus on MICE for three compelling reasons: 1) MICE offers flexibility and can utilize various predictive models, including those that can be efficiently trained inside database systems; 2) In terms of imputation quality, MICE competes with or outperforms other state-of-the-art imputation methods, including recent generative approaches [69, 73]; 3) MICE has been extensively studied in the past, widely used across diverse domains [3, 24, 45, 68], and implemented in popular statistical tools [68, 72]. We leave studying other model-based imputation methods as future work.

Our contributions. We next highlight our main contributions.

To support MICE within a database system (Challenge 1), we introduce techniques for in-database training of two novel models: stochastic linear regression [13], used for imputing continuous values, and Gaussian discriminant analysis, used for imputing categorical values (Section 3). The former builds upon existing work on learning regression models [49, 64] by incorporating random noise into predictions to capture the uncertainty associated with the imputation process. The latter transforms a multinomial classification problem into a database problem, computing the model parameters using a single database query over normalized data, without the need for prior one-hot encoding and materialization of the training dataset. To the best of our knowledge, no prior work has explored classification methods in this setting. Interestingly, both models compute the same set of database aggregates in the training phase, despite their distinct characteristics.

To reduce imputation time (Challenge 2), we adapt the MICE algorithm to exploit computation sharing across iterations while preserving its accuracy, regardless of the proportion of missing data (Section 4). To unlock the sharing potential, we select stochastic linear regression and Gaussian discriminant analysis as the models for imputing continuous and categorical attributes, respectively. As mentioned, training these models within a database system relies on computing the same set of aggregate values. We observe that MICE computes these aggregates over overlapping subsets of records. The extent of overlap increases as the proportion of missing values decreases. For datasets with low missing rates, we propose a rewriting of the MICE algorithm that: 1) performs the most expensive computation over the entire dataset once, outside the iterative loop; and 2) performs less expensive incremental computations over (smaller) incomplete parts of the dataset inside the

iterative loop. This rewriting accelerates model training over datasets with less than 20% missing values, based on our experimental results. We further devise partitioning strategies to minimize redundant computation and avoid repetitive scans of the entire dataset, tailoring to both low and high missing rate scenarios. Our improvements of the MICE algorithm are applicable not only within a database system but also in other tools and libraries implementing MICE.

To avoid materialization of large datasets (Challenge 3), we optimize the computation of MICE aggregates using the mathematical notion of ring[1]. We leverage the cofactor ring from prior work [49, 50] to compactly encode the needed aggregates as ring values and compute them using the sum and product from the ring. When imputing a normalized dataset, we exploit the algebraic properties of the ring to push the aggregate computation past joins, eliminating the need to materialize the joined relation. Furthermore, to avoid the size explosion caused by one-hot encoding of categorical attributes, we utilize the generalized cofactor ring [50] to uniformly encode the needed aggregates over continuous and categorical attributes as generalized multiset relations [31]. This representation accounts for only the interactions between attributes that exist in the dataset, avoiding the sparsity of one-hot encoding. Finally, the ring-based representations is essential for our optimized MICE implementation because it allows the aggregates to be incrementally computed as new values are imputed.

We implemented our data imputation approach in PostgreSQL and DuckDB, including the procedures for in-database learning of ridge linear regression, stochastic linear regression, and Gaussian discriminant analysis models. To efficiently compute the aggregates required for training these models, our implementation harnesses the power of the cofactor ring, representing the first instance of incorporating this abstraction into fully-fledged database systems. Our ring-based implementation improves the performance of computing these aggregates in PostgreSQL and DuckDB by up to 6x over a single table and up to 12x over multiple tables.

Our experiments show that our in-database imputation outperforms the fastest competitor, SystemDS [7, 12], by 3-13x when using PostgreSQL and 86-346x when using DuckDB, depending on the fraction of missing values. For datasets comprising multiple tables, our imputation method can leverage factorized evaluation to bring a 6x improvement compared to imputing over the joined table. Our MICE method is on par with other model-based methods in terms of imputation quality but requires up to two orders of magnitude less time, under various missing rates and patterns.

The rest of this article is organized as follows. Section 2 introduces background material. Section 3 shows how to support stochastic linear regression and Gaussian discriminant analysis within a database system. Section 4 presents our adapted MICE algorithm and its implementation in Section 5. Section 6 gives experimental results, followed by a discussion of related work and conclusion.

Our implementation of the above machine learning and imputation methods in PostgreSQL and DuckDB is publicly available at https://github.com/eddbase/db-imputation.

2 BACKGROUND

We next introduce the MICE algorithm for imputing missing values and review the relevant prior work on in-database learning.

Notation. Consider an incomplete dataset $\mathbf{X}$ consisting of records over attributes $X_1, \ldots, X_m$. We denote by $\tilde{\mathbf{X}}$ a complete version of $\mathbf{X}$ with the missing values replaced by imputed values. We write $\tilde{\mathbf{X}}_{i=miss}$ (or $\tilde{\mathbf{X}}_{i=obs}$) to denote the records from $\tilde{\mathbf{X}}$ for which the value of attribute X_i was originally missing (or observed).

[1]A ring is a set $\mathcal{R}$ with two binary operations, + (addition) and $*$ (multiplication), such that $(\mathcal{R}, +)$ is an abelian group, $(\mathcal{R}, *)$ is a monoid, and $*$ is distributive over +.

Algorithm 1: MICE

Input : $\mathbf{X}$ incomplete dataset with attributes $X_1, ..., X_m$
mattrs indices of incomplete attributes
Output: $\tilde{\mathbf{X}}$ imputed dataset

$\tilde{\mathbf{X}} \leftarrow \mathbf{X}$ with initial imputations for all missing values
repeat
  foreach $i \in mattrs$ **do**
    $\boldsymbol{\theta} \leftarrow \textsc{Train}(data = \tilde{\mathbf{X}}_{i=obs},\ target = X_i)$
    $\tilde{\mathbf{X}} \leftarrow \textsc{Predict}(data = \tilde{\mathbf{X}}_{i=miss},\ target = X_i, model = \boldsymbol{\theta})$
  end
until *stopping_condition*;

2.1 The MICE Algorithm

Multivariate Imputation by Chained Equations (MICE) is an imputation method for handling missing data in multivariate datasets [68]. The method iteratively imputes missing values in each attribute based on the observed values in the other attributes. This imputation process accounts for the patterns present in the data, preserving correlations among attributes and yielding more accurate imputations than model-free methods (e.g., mean imputation).

The MICE method, shown in Algorithm 1, starts by imputing the missing data with initial guesses, typically the mean/mode values. For each attribute X_i with missing values, the algorithm fits a model over the observed part $\tilde{\mathbf{X}}_{i=obs}$ with X_i as target and then uses this model to generate new imputations of X_i in the missing part $\tilde{\mathbf{X}}_{i=miss}$. This iterative process continues until a stopping criterion is satisfied or the maximum number of iterations specified by the user is reached. The MICE algorithm offers flexibility by allowing the use of distinct models for different attributes, making it capable of handling both continuous and categorical attributes.

We refer to one round of imputations of all incomplete attributes as one iteration step. While the MICE algorithm offer no convergence guarantees, it generally converges within a small number of iterations in practice, typically between 5 and 20 iterations [13].

Example 2.1. Consider an incomplete flight dataset with three attributes, two continuous (*Distance* and *AirTime*) and one categorical (*Diverted*), each with missing values. The MICE algorithm first imputes the missing values in each attribute with the mean or mode, depending on the attribute's type. It then trains a regression model to predict *Distance* given *AirTime* and *Diverted* over the subset of records with complete *Distance* values. The trained model serves to impute missing *Distance* values. The second regression model predicts *AirTime*, while the third classification model predicts *Diverted*. The algorithm refines the imputed values in a round-robin way until convergence or a fixed number of iterations. □

The MICE algorithm involves retraining models for every incomplete attribute in every iteration, making it potentially prohibitively expensive on large datasets with many incomplete attributes.

2.2 In-Database Linear Regression

Consider a training dataset $\mathbf{X}$ consisting of N training examples with attributes $X_1, \ldots, X_m$. Without loss of generality, assume that $X_1 = 1$ for all examples in $\mathbf{X}$, and X_m is the target attribute. The goal of linear regression is to find the parameters $\boldsymbol{\theta} = [\theta_1 \ldots \theta_m]^T$, with $\theta_m = -1$, that minimize the squared error loss $L = (\mathbf{X}\boldsymbol{\theta})^T \mathbf{X}\boldsymbol{\theta}$.

Batch gradient descent solves this optimization problem by iteratively updating the learned parameters in the opposite direction of the gradient of L until convergence, while θ_m is fixed to -1:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \mathbf{X}^T(\mathbf{X}\boldsymbol{\theta}^{(k)})$$

where α is the learning rate and $\boldsymbol{\theta}^{(k)}$ are the parameter values in iteration k. Repeatedly scanning the complete dataset to compute $\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta}^{(k)})$ can be time-consuming, especially with large datasets.

A more efficient approach precomputes $\mathbf{X}^T\mathbf{X}$ once and reuses it in each iteration, effectively decoupling the computation over the training dataset from the parameter convergence [64]:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha\, \mathbf{C}\, \boldsymbol{\theta}^{(k)}$$

where $\mathbf{C} = \mathbf{X}^T\mathbf{X}$ is the cofactor matrix of size $m \times m$, which quantifies the level of correlation for each combination of attributes. The one-off computation of $\mathbf{X}^T\mathbf{X}$ takes $O(Nm^2)$ time. Using the precomputed cofactor matrix, each iteration now takes time $O(m^2)$ instead of $O(Nm)$, yielding faster convergence as usually $N \gg m$.

Cofactor Matrix Computation. Let us first assume that all attributes are continuous. The cofactor matrix $\mathbf{X}^T\mathbf{X}$ accounts for the interactions $\texttt{SUM(X}_\texttt{i} * \texttt{X}_\texttt{j}\texttt{)}$ of all pairs (X_i, X_j) of attributes. Thus, computing the cofactor matrix amounts to executing a database query with $O(m^2)$ sum aggregates over the training dataset.

When the training dataset $\mathbf{X}$ is the result of a join, a naïve way of computing the cofactor matrix is to first calculate the join result and then calculate the cofactor aggregates. Based on our experimental evaluation, existing query optimizers make no attempts to factorize the evaluation of many aggregates, that is, to perform partial preaggregation by pushing SUMs past joins [38].

Prior work shows how to express the cofactor matrix computation as the computation of one compound aggregate. This aggregate is a triple $(N, \mathbf{s}, \mathbf{Q})$, where N is the size of the dataset, `SUM(1)`; $\mathbf{s}$ is a vector of sums of values for each attribute, $\mathbf{s}_i = \texttt{SUM(X}_\texttt{i}\texttt{)}$; and $\mathbf{Q}$ is a matrix of sums of products of values for any two attributes, $\mathbf{Q}_{(i,j)} = \texttt{SUM(X}_\texttt{i} * \texttt{X}_\texttt{j}\texttt{)}$. The computation over triples is captured by the cofactor (degree-m matrix) ring [49, 50].

The cofactor ring defines the addition and multiplication operations over triples. Let $\mathcal{R}$ be a set of triples $(\mathbb{Z}, \mathbb{R}^m, \mathbb{R}^{m\times m})$, for fixed $m \in \mathbb{N}$. For any $a = (N_a, \mathbf{s}_a, \mathbf{Q}_a) \in \mathcal{R}$ and $b = (N_b, \mathbf{s}_b, \mathbf{Q}_b) \in \mathcal{R}$, the addition and multiplication operations on $\mathcal{R}$ are defined as:

$$a +^{\mathcal{R}} b = (N_a + N_b,\ \mathbf{s}_a + \mathbf{s}_b,\ \mathbf{Q}_a + \mathbf{Q}_b)$$
$$a *^{\mathcal{R}} b = (N_a\, N_b,\ N_b\, \mathbf{s}_a + N_a\, \mathbf{s}_b,\ N_b\, \mathbf{Q}_a + N_a\, \mathbf{Q}_b + \mathbf{s}_a\mathbf{s}_b^T + \mathbf{s}_b\mathbf{s}_a^T)$$

where $+^{\mathcal{R}}$ uses scalar and matrix addition, and $*^{\mathcal{R}}$ uses matrix addition and scalar and matrix multiplication. The additive identity (zero) is $(0, \mathbf{0}_{m\times 1}, \mathbf{0}_{m\times m})$ and the multiplicative identity (one) is $(1, \mathbf{0}_{m\times 1}, \mathbf{0}_{m\times m})$, where $\mathbf{0}_{m\times n}$ is the zero matrix of size $m \times n$.

The function λ_{con} maps values of continuous attribute X to triples from $\mathcal{R}$ such that $\lambda_{\text{con}}(x, i) = (1, \mathbf{s}, \mathbf{Q})$, where i is the index of X in the cofactor matrix, and $\mathbf{s}$ and $\mathbf{Q}$ contain all zeros except $\mathbf{s}_i = x$ and $\mathbf{Q}_{(i,i)} = x^2$. We refer to λ_{con} as a lifting function. For brevity, we omit the index i in λ_{con}, assuming a fixed order of attributes.

Example 2.2. For the flight dataset from Example 2.1, we can compute the cofactor matrix over the attributes *Distance* and *AirTime* using one aggregate query, which returns a triple from the cofactor ring:

```
SELECT SUM(λcon(Distance) * λcon(AirTime)) FROM Flight
```

The function λ_{con} maps a *Distance* value d to $(1, [\, d\ 0\,], [\, d^2\ 0\ ;\ 0\ 0\,])$ and an *AirTime* value a to $(1, [\, 0\ a\,], [\, 0\ 0\ ;\ 0\ a^2\,])$. The triple multiplication yields $(1, [\, d\ a\,], [\, d^2\ da\ ;\ ad\ a^2\,])$. The SUM operator

uses the addition $+^{\mathcal{R}}$ from the cofactor ring. The result encodes the cofactor aggregates: `SUM(1)`, `SUM(D)`, `SUM(A)`, `SUM(D*D)`, `SUM(D*A)`, and `SUM(A*A)`, where `D` and `A` stand for `Distance` and `AirTime`, respectively. □

When computing the cofactor matrix over joins, this ring-based approach allows for the `SUM` to be pushed past the joins, leveraging the distributivity of multiplication over addition in $\mathcal{R}$. This eliminates the need to compute the entire join result in advance and promotes the sharing of aggregate computation.

Handling Categorical Attributes. Most machine learning algorithms require numerical input and cannot work directly with discrete categories. One-hot encoding is often employed to represent categorical attributes as indicator vectors. But this encoding step can cause data explosion as each category yields a new binary attribute.

The cofactor matrix of a one-hot encoded dataset includes the following aggregates: `SUM(`X_i `*` X_j`)` when X_i and X_j are continuous; `SUM(`X_i`) group by` X_j when X_i is continuous and X_j is categorical; and `SUM(1) group by` X_i`,` X_j when X_i and X_j are categorical [64].

To avoid one-hot encoding and operate directly over categorical values, prior work [50] generalizes the cofactor ring with the ring over relations [31] to uniformly treat aggregates with continuous and categorical attributes. The triple structure remains unchanged except that N, $\mathbf{s}$, and $\mathbf{Q}$ contain relations instead of scalars. Each relation is a mapping from tuples to scalars. The operations $+^{\mathcal{R}}$ and $*^{\mathcal{R}}$ remain unchanged except that scalar addition is replaced by union and scalar multiplication is replaced by join. A scalar c is represented as the relation $\{() \mapsto c\}$ mapping the empty tuple to c.

The generalized cofactor ring requires new functions for mapping attribute values to ring values. For a categorical attribute X with index i in the cofactor matrix, the lifting function λ_{cat} maps categories of X to triples such that $\lambda_{\text{cat}}(x) = (\mathbf{1}, \mathbf{s}, \mathbf{Q})$, where $\mathbf{1}$ is $\{() \mapsto 1\}$, and $\mathbf{s}$ and $\mathbf{Q}$ contain all empty relations except $\mathbf{s}_i = \{x \mapsto 1\}$ and $\mathbf{Q}_{(i,i)} = \{x \mapsto 1\}$. The lifting function λ_{con} returns a triple as before but with every scalar c replaced by $\{() \mapsto c\}$.

Example 2.3. For the flight dataset from Example 2.1, we can compute the cofactor matrix over *AirTime* (continuous) and *Diverted* (categorical) using a query over the generalized cofactor ring:

```
SELECT SUM(λcon(AirTime) * λcan(Diverted)) FROM Flight
```

Each tuple (a, d) is mapped to $(N, \mathbf{s}, \mathbf{Q})$, where $N = \{() \mapsto 1\}$, $\mathbf{s} = [\ \{()\mapsto a\}\ \{d\mapsto 1\}\ ]$, and $\mathbf{Q} = [\ \{()\mapsto a^2\}\ \{d\mapsto a\}\ ;\ \{d\mapsto a\}\ \{d\mapsto 1\}\ ]$). The final aggregate $(N, \mathbf{s}, \mathbf{Q})$ encodes: `SUM(1)` in N; `SUM(A)` and `SUM(1) group by D` in $\mathbf{s}$; `SUM(A * A)` and `SUM(A) group by D` in $\mathbf{Q}$, where `A` and `D` stand for `AirTime` and `Diverted`, respectively. □

3 IN-DATABASE IMPUTATION METHODS

This section presents two methods for learning models needed for the imputation of continuous and categorical values, within a database management system. The first method extends the approach for learning regression models from Section 2.2 to incorporate random noise into predictions. The second method is a novel approach for in-database classification using Gaussian discriminant analysis. Seemingly disparate, both methods rely on the aggregates that can be computed using the generalized cofactor ring.

3.1 Stochastic Linear Regression

Using linear regression for data imputation carries the risk of overstating the strength of the relationship between the target and predictor attributes. Stochastic regression imputation [13] is a refinement of regression imputation that attempts to address this correlation bias by adding

noise to the predictions. By doing so, imputed data will randomly deviate from the regression line, capturing the inherent uncertainty of the imputation process.

For convenience, we summarize the setup from Section 2.2. A training dataset $\mathbf{X}$ consists of N training examples over continuous attributes $X_1, \ldots, X_m$, where X_m denotes the target and X_1 is fixed to 1. Linear regression learns the parameters $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_m]^T$, with $\theta_m = -1$, minimizing the squared error loss $L = (\mathbf{X}\boldsymbol{\theta})^T\mathbf{X}\boldsymbol{\theta}$.

Stochastic linear regression estimates the parameters $\boldsymbol{\theta}$ under the linear model but adds random noise to predictions: $f(\mathbf{x}) = \mathbf{x}^T\boldsymbol{\theta'} + \epsilon$, where $\mathbf{x} \in \mathbb{R}^{m-1}$, $\boldsymbol{\theta'} = [\theta_1 \ldots \theta_{m-1}]^T$, and $\epsilon \sim \mathcal{N}(0, \sigma^2)$ with the variance σ^2 calculated from the vector of residuals, $\mathbf{r} = \mathbf{X}\boldsymbol{\theta}$. Since the mean of residuals is zero in linear regression, we can compute the residual variance as: $\sigma^2 = \frac{\mathbf{r}^T\mathbf{r}}{N} = \frac{\boldsymbol{\theta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}}{N}$, where N, $\boldsymbol{\theta}$, and $\mathbf{X}^T\mathbf{X}$ are already calculated in the training phase.

Database perspective. We compute the aggregates for training and prediction using one query over the generalized cofactor ring:

```
SELECT SUM(λcon(X1) * ... * λcon(Xm)) FROM Xdataset
```

where λ_{con} is the lifting function for the generalized cofactor ring (cf. Section 2.2). Here, we assume that all attributes are continuous. When an attribute X_i is categorical, we use the lifting function λ_{cat} to map X_i-categories into ring values. In the presence of categorical attributes, the computed aggregate is a compact representation of the cofactor matrix computed over the one-hot encoded dataset.

We utilize user-defined functions to unpack the computed aggregate into a real-valued matrix, learn parameters $\boldsymbol{\theta}$, and compute variance σ^2. We generate predictions for a given test dataset as:

```
SELECT (θ1 * X1 + ... + θm-1 * Xm-1 + ϵ) AS prediction FROM Xtest
```

where ϵ is a sample from $\mathcal{N}(0, \sigma^2)$ computed via the Box-Muller transform: $\epsilon = \sqrt{-2\ln U_1}\cos(2\pi U_2)\cdot\sigma$, where U_1 and U_2 are independent samples from the uniform distribution.

3.2 Gaussian Discriminant Analysis

We consider Gaussian discriminant analysis (GDA) for categorical data imputation. It is a type of generative classifier that models the distribution of the input features for each class as a multivariate Gaussian distribution. To classify a new instance, GDA estimates the posterior probability of each class given the input features using Bayes' rule and chooses the class with the highest probability.

We focus here on Linear Discriminant Analysis (LDA), a variant of GDA where all classes share the same covariance matrix, thus yielding linear decision boundaries among classes. The following discussion can serve as a blueprint for other classifiers such as Quadratic Discriminant Analysis, another variant of GDA that uses class-specific covariance matrices, and Naïve Bayes classifiers.

Setup. Consider a training dataset $\mathbf{X}$ consisting of N training examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, where the features $\mathbf{x}_i \in \mathbb{R}^m$ and the targets y_i take on values from a set of classes $\{1, \ldots, C\}$[2]. GDA assumes the class-conditional densities are normally distributed:

$$\Pr(\mathbf{x} \mid y = c) = \frac{1}{(2\pi)^{m/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_c)^T\Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}_c)\right)$$

[2]Without loss of generality, we assume that classes are encoded as integers.

where m is the dimension of the features, $\boldsymbol{\mu}_c$ is the class-specific mean vector, and Σ is the shared covariance matrix for all the classes. Using Bayes' rule, we can compute the class posterior as:

$$\Pr(y = c \mid \mathbf{x}) = \frac{\Pr(\mathbf{x} \mid y = c) \Pr(y = c)}{\sum_{k=1}^{C} \Pr(\mathbf{x} \mid y = k) \Pr(y = k)}$$

We then classify a sample $\mathbf{x}$ into class: $\operatorname{argmax}_c \Pr(y = c \mid \mathbf{x})$.

Training. We estimate the class-specific mean and shared covariance matrix from the training data. Let $\mathbf{1}_{y_i=c}$ denote an indicator function that returns 1 if the i-th training example belongs to class c and 0 otherwise. Let π_c be the prior $\Pr(y = c)$ for class c. To simplify notation, let $\boldsymbol{\theta}$ denote the parameters $(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_C, \Sigma, \pi_1, \ldots, \pi_C)$.

The likelihood of data is given as:

$$L(\boldsymbol{\theta}) = \Pr(\mathbf{X}; \boldsymbol{\theta}) = \prod_{i=1}^{N} \prod_{c=1}^{C} \Pr(\mathbf{x}_i | y_i = c)^{\mathbf{1}_{y_i=c}} \Pr(y_i = c)^{\mathbf{1}_{y_i=c}}$$

By maximizing L with respect to the parameters $\boldsymbol{\theta}$, we find the maximum likelihood estimate of the parameters as:

$$\pi_c = \frac{\sum_{i=1}^{N} \mathbf{1}_{y_i=c}}{N} \qquad \boldsymbol{\mu}_c = \frac{\sum_{i=1}^{N} \mathbf{1}_{y_i=c} \mathbf{x}_i}{\sum_{i=1}^{N} \mathbf{1}_{y_i=c}} \tag{1}$$

$$\Sigma = \frac{1}{N} \sum_{c=1}^{C} \sum_{i=1}^{N} \mathbf{1}_{y_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c)(\mathbf{x}_i - \boldsymbol{\mu}_c)^T \tag{2}$$

The class prior π_c is the proportion of training examples that belong to the class c. The class-specific mean vector $\boldsymbol{\mu}_c$ is the mean of the features of the class c. The shared covariance matrix Σ_c is the weighted average of the covariance matrix of every class.

Prediction. We can classify a sample $\mathbf{x}$ by finding the class c that maximizes the class posterior $\Pr(y = c \mid \mathbf{x})$. After maximizing the log-posterior and dropping terms common to all classes, we obtain the classification function:

$$f(\mathbf{x}) = \operatorname*{argmax}_c \ \ln \pi_c - \frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_c)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_c)$$

We can simplify f by expanding the second argmax term, exploiting the symmetry of Σ, and dropping terms common to all classes:

$$f(\mathbf{x}) = \operatorname*{argmax}_c \ \mathbf{a}_c^T \mathbf{x} + b_c \tag{3}$$

where $\mathbf{a}_c = \Sigma^{-1} \boldsymbol{\mu}_c$ and $b_c = \ln \pi_c - \frac{1}{2} \boldsymbol{\mu}_c^T \Sigma^{-1} \boldsymbol{\mu}_c$.

Database Perspective. We next show how to compute the aggregates needed to estimate the LDA parameters: π_c, $\boldsymbol{\mu}_c$, and Σ. Let N_c denote the number of training examples with class c, that is, $N_c = \sum_{i=1}^{N} \mathbf{1}_{y_i=c}$. The `SUM(1)` aggregate counts the number of examples in the training dataset, while `SUM(1) group by Y` counts the number of examples per class. These aggregates suffice to calculate the prior $\pi_c = \frac{N_c}{N}$ for each class c. To compute the class-specific mean vectors $\boldsymbol{\mu}_c$, we also need a batch of aggregates `SUM(X`$_i$`) group by Y` for each feature attribute X_i, where $i \in [m]$.

To compute the shared covariance matrix, we first rewrite the expression from Equation (2) considering that $\sum_{i=1}^{N} \mathbf{1}_{y_i=c}\mathbf{x}_i = N_c\boldsymbol{\mu}_c$:

$$\Sigma = \frac{1}{N}\sum_{c=1}^{C}\sum_{i=1}^{N} \mathbf{1}_{y_i=c}\mathbf{x}_i\mathbf{x}_i^T - \frac{1}{N}\sum_{c=1}^{C} N_c\boldsymbol{\mu}_c\boldsymbol{\mu}_c^T$$

The first term sums up the class-specific cofactor matrices computed over disjoint subsets of the training datasets. This summation computes the normalized cofactor matrix $\frac{1}{N}(\mathbf{X}_{\text{-}Y})^T\mathbf{X}_{\text{-}Y}$, where $\mathbf{X}_{\text{-}Y}$ is the projection of $\mathbf{X}$ without the target attribute Y.

We can compute all the required aggregates using one database aggregate query over the generalized covariance ring:

```
SELECT SUM(λcon(X1) * ... * λcon(Xm) * λcat(Y)) FROM Xdataset
```

where λ_{con} and λ_{cat} are the lifting functions for the generalized covariance ring. Here, we assume that all input features are continuous. The query returns a triple $(N, \mathbf{s}, \mathbf{Q})$ of aggregates. The matrix $\mathbf{Q}$ of size $(m+1)\times(m+1)$ encodes the following aggregates as relations (we omit the symmetric lower part of $\mathbf{Q}$):

$$\begin{bmatrix} \texttt{SUM(X1*X1)} & \cdots & \texttt{SUM(X1*Xm)} & \texttt{SUM(X1) group by Y} \\ & \ddots & \vdots & \vdots \\ & & \texttt{SUM(Xm*Xm)} & \texttt{SUM(Xm) group by Y} \\ & & & \texttt{SUM(1) group by Y} \end{bmatrix}$$

Using the computed triple of aggregates, we can now calculate the maximum likelihood estimates of the LDA parameters as:

$$N_c = \mathbf{Q}_{(m+1,m+1)}(c) \qquad \boldsymbol{\mu}_c = \frac{[\mathbf{Q}_{(1,m+1)}(c) \ \cdots \ \mathbf{Q}_{(m,m+1)}(c)]^T}{N_c}$$

$$\pi_c = \frac{N_c}{N} \qquad \Sigma = \frac{1}{N}\mathbf{Q}_{(1\ldots m,\,1\ldots m)} - \frac{1}{N}\sum_{c=1}^{C} N_c\boldsymbol{\mu}_c\boldsymbol{\mu}_c^T$$

where $\mathbf{Q}_{(1\ldots m,\,1\ldots m)}$ is the upper-left submatrix of $\mathbf{Q}$ of size $m\times m$.

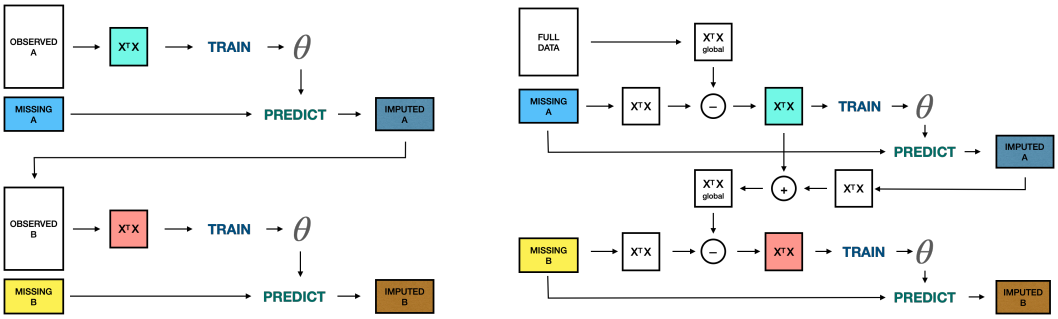
When a feature attribute X_i is categorical, we use $\lambda_{\text{cat}}(\texttt{X}_\texttt{i})$ in the query computing the cofactor aggregate. A user-defined function extracts the computed parameters into real-valued matrices and evaluates the classification function from Equation (3).

4 MICE WITH COMPUTATION SHARING

The MICE algorithm iteratively trains models one after the other. During each iteration, for each attribute with missing data, the algorithm trains a model over the subset of data where the target attribute is not missing. The MICE algorithm from Section 2 retrains models from scratch, discarding previous computations. We next present our improvements over this approach.

In-Database ML. We start by incorporating in-database learning into the MICE algorithm. We opt for stochastic linear regression and linear discriminant analysis as the models for imputing continuous and categorical data. These models are efficiently trainable and compute the cofactor matrix $\mathbf{X}^T\mathbf{X}$ during training, allowing for further optimizations. Figure 2a visualizes this approach. After the initial imputation, for the attribute A with missing data, we compute the cofactor matrix over the subset of records with observed (non-missing) A values, train a model, and impute the missing A values. The same steps are repeated for other incomplete attributes.

This approach benefits from faster training of individual models but still computes the cofactor matrix over the observed data for each incomplete attribute, in each iteration. When the fraction of



(a) MICE with in-database ML *(b)* MICE with in-database ML and computation sharing

Fig. 2. Two improvements of the MICE algorithm. Colored blocks with the same color in both figures contain the same data.

missing values is low, as often the case in practice, recomputing the cofactor matrix due to small changes is unnecessarily expensive.

Incremental Maintenance of Cofactor Matrices. We can improve the previous approach using incremental computation. We precompute the cofactor matrix over the observed data for each incomplete attribute once. When new imputations are created, we maintain each cofactor matrix by adjusting the contribution of the affected records before and after the imputation. We exploit here that the cofactor aggregates are ring values supporting + and −.

The drawback of this approach is that it stores and maintains multiple cofactor matrices. Every time imputed values are updated for one attribute, each of the cofactor matrices might be affected due to the overlap among the datasets used for their computation.

Shared Computation of Cofactor Matrices. We next present two optimizations designed to boost the performance of MICE by enabling computation sharing across iteration. The first optimization targets datasets with low missing rates, while the second optimization applies to both low and high missing rate datasets.

(1) Shared Computation with Low Missing Rates. In practice, the observed data is often much larger than the missing data. We leverage this observation to compute one global cofactor matrix over the entire (initially imputed) data once, and use it to derive the cofactor matrix for each attribute on-the-fly, by scanning only the missing data. This approach effectively moves the expensive computation over the observed data outside the iteration loop.

Algorithm 2 shows our MICE algorithm with computation sharing. After the initial imputation, we compute the global cofactor matrix over the imputed dataset $\tilde{\mathbf{X}}$ (Line 2). To compute the cofactor matrix for an attribute X_i, we remove the contribution of the records with missing X_i-values, $\tilde{\mathbf{X}}_{i=miss}$, from the global cofactor matrix since these records are not used for training (Lines 5-6). After training the model and imputing new values, we update the global matrix to account for the new imputations (Lines 9-10), making it ready for the next iteration. Figure 2b visualizes this approach. The colored blocks denote the same data in the two approaches, with and without sharing the cofactor matrix computation.

(2) Shared Computation with Data Partitioning. We can further accelerate the cofactor matrix computation by partitioning the data into partitions based on the number of missing values in each record. We start with two observations. First, the records containing no missing values are part of every training dataset and contribute equally to the cofactor matrix computed in each iteration. Thus, we can form a partition of such records, precompute their contribution once, and reuse it in

Algorithm 2: MICE with computation sharing

Input : $\mathbf{X}$ incomplete dataset with attributes $X_1, ..., X_m$
mattrs indices of incomplete attributes
Output: $\tilde{\mathbf{X}}$ imputed dataset

$\tilde{\mathbf{X}} \leftarrow \mathbf{X}$ with initial imputations for all missing values
$\mathbf{C} \leftarrow \tilde{\mathbf{X}}^{\mathrm{T}}\tilde{\mathbf{X}}$
repeat
  foreach $i \in mattrs$ **do**
    $\Delta\mathbf{C} \leftarrow (\tilde{\mathbf{X}}_{i=miss})^{\mathrm{T}}\tilde{\mathbf{X}}_{i=miss}$
    $\mathbf{C}_{train} \leftarrow \mathbf{C} - \Delta\mathbf{C}$
    $\boldsymbol{\theta} \leftarrow \textsc{Train}(cofactor = \mathbf{C}_{train},\ target = X_i)$
    $\tilde{\mathbf{X}} \leftarrow \textsc{Predict}(data = \tilde{\mathbf{X}}_{i=miss},\ target = X_i, model = \boldsymbol{\theta})$
    $\Delta\mathbf{C} \leftarrow (\tilde{\mathbf{X}}_{i=miss})^{\mathrm{T}}\tilde{\mathbf{X}}_{i=miss}$
    $\mathbf{C} \leftarrow \mathbf{C}_{train} + \Delta\mathbf{C}$
  end
until *stopping_condition*;

every iteration. Second, the records containing missing values in all incomplete attributes are not part of any training dataset, thus we can skip such records during model training and only impute them at the end of each iteration.

For datasets with low missing rates, we want to ensure fast access to (small) incomplete data, $\tilde{\mathbf{X}}_{i=miss}$, needed by Algorithm 2. Before starting the imputation, we split each dataset into four partitions: one stores records without missing values, one stores records with only missing values, one stores records with exactly one missing value, and one stores records with at least two missing values. We further recursively partition the third partition containing records with one missing values into subpartitions, one for each incomplete attribute. Then, accessing the records with missing values in a given attribute requires scanning two partitions: the subpartition of the given attribute and the third ('overflow') partition. When the fraction of missing values is low, both of these partitions tend to be small, allowing fast access to the needed records.

For datasets with high missing rates, we want to ensure fast access to (small) observed data, $\tilde{\mathbf{X}}_{i=obs}$, to speed up the model training in Algorithm 1. The partitioning strategy in this case is the complete opposite of that in the case with low missing rates: partitioning uses the same criteria but based on the number of observed (not missing) values of incomplete attributes in each record. The partition containing records with only observed values is used to precompute a partial cofactor aggregate once, outside the iteration loop. The rest of the training dataset for an incomplete attribute contains records from two partitions: one subpartition storing records with exactly one observed value for the given attribute and one partition storing records with at least two observed values. The model training in each iteration now needs to scan only these two partitions, often much smaller than the entire dataset when its missing rate is high.

5 IMPLEMENTATION

We next discuss how to implement our data imputation methods in existing DBMSs. We opt for an in-database implementation instead of building a specialized tool for several reasons:

- Providing a solution integrated with widely-used DBMSs facilitates its adoption, eliminating the need for adding yet another tool to already complex data pipelines.

- DBMSs are mature systems with robust mechanisms for handling large data and recovering from failures, in contrast to data imputation implementations in tools such as R.
- DBMSs offer highly-optimized techniques for query evaluation, including parallel execution, allowing us to focus on more high-level aspects of the imputation process.

We implemented our imputation methods in PostgreSQL and DuckDB, two open-source database systems with row-oriented and column-oriented storage models, respectively. Our approach can be implemented in any other database system that supports defining custom data types of variable size and aggregate functions operating over values of these types. Our implementation assumes that categorical values are encoded as integers; if they are not, we can map categorical values to integers in a preprocessing step.

5.1 In-Database ML Implementation

We provide libraries in PostgreSQL and DuckDB for in-database training of ridge linear regression, stochastic linear regression, and LDA models, and for generating predictions under each model. The core library component is the implementation of the generalized cofactor ring. We refer to this data structure as Triple.

In PostgreSQL, Triple is a custom data type of variable size. A Triple value is a struct that comprises an array storing the numerical aggregates over continuous attributes, followed by an array storing the relational aggregates over continuous and categorical attributes. The struct occupies a contiguous memory chunk with no external pointers to allow for fast allocation/de-allocation.

DuckDB, instead, offers native support for nested data types such as arrays and structs, allowing for a simpler Triple implementation based on using an array of numerical values and an array of key-value structs.

In both cases, we implement the ring operations, addition, subtraction, and multiplication, over Triple values, as user-defined functions. Instead of the lifting functions λ_{con} and λ_{cat}, we provide a more efficient bulk version λ that takes as input a list of continuous attributes and a list of categorical attributes and maps their values to a Triple aggregate at once, avoiding triple multiplication.

```
SELECT SUM(λ([X_{i_1},...,X_{i_k}], [X_{j_1},...,X_{j_1}])) FROM X_dataset
```

DuckDB supports a faster way of computing Triples via a custom aggregate operator that operates directly over attribute values:

```
SELECT SUM_TRIPLE(X_{i_1},...,X_{i_k}, X_{j_1},...,X_{j_1}) FROM X_dataset
```

In the implementation, the aggregate operator `SUM_TRIPLE` takes as input a list of value vectors of a fixed size, together with their type (continuous or categorical), and aggregates these values to a Triple in bulk, rather than one record at a time.

Factorized Computation of the Cofactor Matrix. When computing the cofactor matrix over joins, we can exploit the algebraic properties of the cofactor ring to achieve factorized evaluation, that is, compute partial Triple values over individual tables and then combine these triples to produce the final result.

Example 5.1. Consider relations $R(A, B)$ and $S(B, C, D)$, where C is categorical and the others are continuous. We can compute the cofactor triple for A, C, and D over the join of R and S as:

```
SELECT SUM(λ([A,D], [C])) FROM R, S WHERE R.B = S.B
```

The factorized query computes partial triples before the join:

```
SELECT SUM(t1.T * t2.T) FROM
  (SELECT B, SUM(λ([A], [ ])) AS T FROM R GROUP BY B) AS t1,
```

```
  (SELECT B, SUM(λ([D],[C])) AS T FROM S GROUP BY B) AS t2
WHERE t1.B = t2.B
```

This optimization often pays off when the domain of B is small. □

In the current implementation, we refactor input queries like in Example 5.1 before passing them to the query optimizer. As future work, we aim to enable this optimization in the query optimizer.

The Triple value computed over the training dataset is passed to the functions for model training and prediction, along with other parameters such the index of the target attribute, learning rate, and regularization factor. LDA relies on LAPACK routines [5] to solve systems of linear equations and perform matrix operations, necessary in model training and prediction.

5.2 In-Database MICE Implementation

We implemented our imputation methods as driver functions in PL/pgSQL for PostgreSQL and in C++ for DuckDB. We provide three functionally-equivalent implementations of the MICE algorithm:

(1) Baseline implements the logic from Algorithm 1 using the generalized cofactor ring, without any partitioning strategy;
(2) Low implements the shared cofactor computation from Algorithm 2 using the partitioning strategy for datasets with low missing rates;
(3) High implements the shared cofactor computation using the partitioning strategy for datasets with high missing rates.

Each implementation creates a copy of the dataset that needs to be imputed in the preprocessing step, outside the iteration loop. The Baseline version repeatedly scans the entire copy to train models over subsets of records with complete values for different incomplete attribute. The other two versions create a partitioned copy and compute partial cofactor aggregates in the preprocessing step, as discussed in Section 4. This partitioning can accelerate the retrieval of matching records, reducing the per-iteration cost at the expense of increasing the preprocessing cost.

Reducing Update Overhead. Both PostgreSQL and DuckDB use multi-version concurrency control (MVCC) to maintain data consistency. When imputing new values using an `UPDATE` command, MVCC creates new versions of the affected objects. As the algorithm proceeds iteratively, updating imputed values leads to increasingly many obsolete versions, causing significant overheads and eventually becoming the bottleneck in our implementation.

A potential solution is to store each attribute's imputed values in another table, re-created every time these values are updated. The main drawback of this solution, however, is that it requires joining all tables of imputed values every time a model is trained.

We reduce the update overhead with the following optimizations:

- *Swapping columns (DuckDB).* We changed the internals of DuckDB to support pointer-based column swaps between tables (70 LOC), inspired by a similar approach from prior work [23]. We create a temporary table with one column containing new imputed values and then move this column in place of an existing one in the corresponding table partition.
- *Recreating subpartitions (DuckDB & PostgreSQL).* The subpartitions storing records with one missing value are completely changed in each iteration. In PostgreSQL, we recreate such subpartitions with new imputations but update in-place other affected partitions. In DuckDB, we always swap a fresh column with new imputations into affected partitions.
- *Enabling heap-only-tuples (PostgreSQL).* This optimization in PostgreSQL aims to store multiple versions of one row on the same page, reducing update overheads. For partitions updated in-place (e.g., those storing records with at least two missing values), we reduce their page fill factor to 75%.

- *Using shorter transactions (PostgreSQL).* The vacuum processes in PostgreSQL can only remove row versions that are older than any currently active transaction. Repeatedly updating the same rows in a single transaction leads to accumulating obsolete versions and degrading performance. Thus, we start a new transaction on imputing each attribute.

6 EXPERIMENTS

We compare our techniques for in-database learning and data imputation in DuckDB and PostgreSQL against the following systems[3]: Apache SystemDS [7], a data science platform for large-scale data analysis and machine learning; Apache MADlib [6], a PostgreSQL library for in-database machine learning; MindsDB [47], a layer on top of database systems for training machine learning models; and five alternative imputation methods implemented in Python.

Our experimental findings can be summarized as follows:

- Using the cofactor ring for computing cofactor aggregates improves the training performance by up to 6x, regardless of the dataset and DBMS. Factorized computation of the cofactor aggregates can further improve the performance up to 12x if the denormalized database is highly redundant.
- When imputing a single table, our DuckDB implementation outperforms the fastest competitor, SystemDS, in terms of per-iteration cost by 86x to 346x as the rate of missing values ranges between 5% and 80%. The imputation time scales linearly with the number of incomplete attributes.
- When imputing missing values over a normalized dataset, employing factorized evaluation can be faster than denormalizing the dataset before imputation by up to 6 times in PostgreSQL and up to 1.7 times in DuckDB.
- Our MICE implementations are competitive with or outperform state-of-the-art imputation methods in terms of imputation quality, while offering up to two orders of magnitude faster imputation, under various missing rates and patterns.

Experimental Setup. We run all experiments on a server with 2 x AMD EPYC 7302 16-Core Processor, 64 threads, 512 GB RAM running Ubuntu 20.04. We use PostgreSQL 12.12 and DuckDB 0.8.1. We run each experiment 3 times with a timeout of 200 minutes and report averaged results, unless stated otherwise.

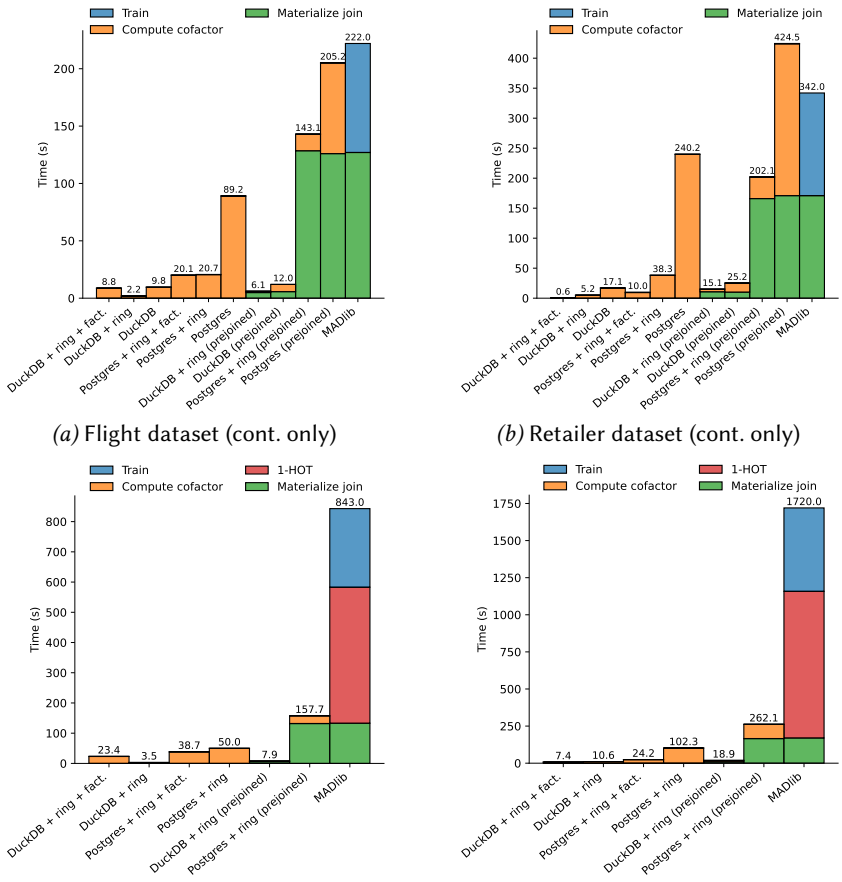
Datasets. We consider three real-world datasets: (1) *Flight Delays and Cancellations* [51] contains information about U.S. flights. It consists of 3 tables, 60M rows and 31 columns, of which 5 are categorical. (2) *Retailer* [64] contains historical inventory data of stores at different locations. It consists of 5 tables arranged in a snowflake schema, 84M rows and 25 columns, of which 4 are categorical. (3) *Taiwan's Air Quality* [2] contains information about Taiwan's air quality in the years 2016-2021. It is a single table with 3.5M rows, 11 numerical columns, and 6% missing values.

The Flight and Retailer datasets are complete, containing no missing values. These datasets serve as the foundation for our benchmarks, where we randomly remove different quantities of values to evaluate performance under various scenarios.

6.1 In-Database Learning

We evaluate the performance of training a linear regression model inside DuckDB and PostgreSQL when the training dataset is formed by joining the input tables. We consider three different evaluation approaches for DuckDB and PostgreSQL. The baseline approach uses standard SQL with scalar `SUM` aggregates to compute the cofactor matrix, the second approach adopts the cofactor ring (`ring`), and the third approach additionally includes factorized evaluation (`ring + fact`). We also compare

[3]Our code is available at https://github.com/eddbase/db-imputation.



(a) Flight dataset (cont. only) *(b)* Retailer dataset (cont. only)

(c) Flight dataset (cont. & cat.) *(d)* Retailer dataset (cont. & cat.)

Fig. 3. Breakdown of execution time for training a linear regression model in DuckDB, PostgreSQL, and MADlib over the join of the input tables from the Retailer and Flight datasets, with continuous only and continuous + categorical attributes. The `ring` variants use the covariance ring, `fact` uses factorized evaluation, and `prejoined` materializes the join result first.

our library with MADlib. We run this experiment over the Flight and Retailer datasets considering only continuous and both continuous and categorical attributes. MADlib assumes a single table is available, therefore, we need to precompute and materialize the joined relation. We are unable to compute the cofactor aggregates with a standard SQL query over categorical attributes, because the number of aggregates after one-hot encoding exceeds the limits in PostgreSQL and DuckDB.

Figure 3 shows the performance of these approaches. In the case of continuous-only attributes, the adoption of the ring structure leads to 4 to 6 times higher performance, regardless of the DBMS used due to the compact representation of the cofactor aggregates. While the addition of categorical attributes slows down the computation, the performance improvement over MADlib is even higher, as one-hot encoding in MADlib takes 450s for Flight and 980s for Retailer, in addition to materialization of the joined result.

Computing the cofactor aggregates using factorized evaluation affects the performance differently according to the dataset used. In DuckDB, training a model over the normalized Retailer dataset

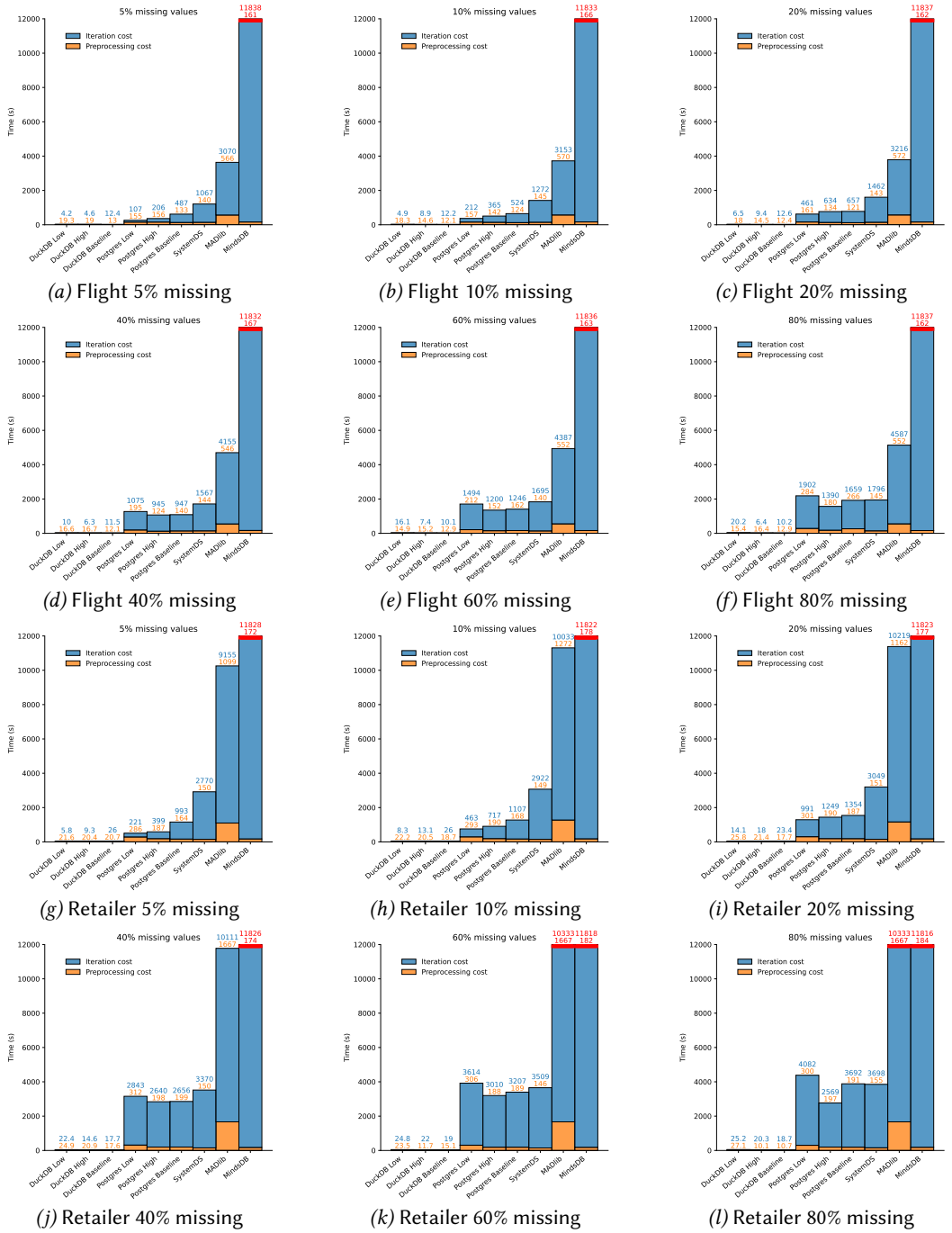
improves the performance by 8.7x with continuous-only attributes and by 1.4x with mixed attributes compared with non-factorized evaluation. The smaller speedup with categorical attributes is due to increased memory management pressure caused by resizing the data structures that store categorical values. With the Flight dataset, factorized evaluation leads to a higher runtime because joining the input tables does not produce many redundant values: the fact table already contains most of the data, with the other tables being 15% of its size. In Retailer, joining the input tables brings more redundancy as the fact table has only 4 attributes, with the other tables being less than 1% of its size. The computation saved by factorized evaluation is therefore more pronounced in Retailer than Flight. Our experiments with PostgreSQL show similar results.

6.2 Single-Table Imputation

We compare our three MICE implementations from Section 5.2 – Baseline, Low, and High – in PostgreSQL and DuckDB against SystemDS, MADlib, and MindsDB, measuring the time required to execute one round of the MICE algorithm, that is, impute values in each incomplete attribute once. Since every competitor except ours assumes a single table as input, we precompute the join result for the Flight and Retailer datasets. We randomly remove different quantities of missing values from 7 columns in each dataset to measure their impact on the runtime. SystemDS and MADlib implement MICE with linear regression and logistic regression. Both implementations produce predictions similar to ours obtained using stochastic linear regression and LDA; the difference of RMSE on Flight and Retailer is less than 1% after convergence. MindsDB uses gradient boosting decision trees (LightGBM) for MICE.

Figure 4 reports the time required to run a single round of MICE over 7 columns (in blue) and the preprocessing time (in orange) over Flight and Retailer when the percentage of missing values varies between 5% and 80%. Our baseline implementations in DuckDB and PostgreSQL achieve lower per-iteration costs than all other competitors on both datasets, regardless of the fraction of missing values. Compared to the leading competitor, SystemDS, our DuckDB baseline is increasingly faster per iteration as the missing rate increases, ranging from 86x to 176x on Flight and from 106x to 346x on Retailer. The performance improvement is due to several reasons. All other competitors perform one-hot encoding in the preprocessing stage, increasing the size of the training dataset. SystemDS and MADlib use the direct solve method for linear regression, computing the cofactor matrix and its inverse. MindsDB executes expensive training of decision trees for every column. Our methods, instead, use gradient descent for regression, computing one compound aggregate in one pass over the dataset, without the need for prior one-hot encoding. DuckDB further benefits from columnar-vectorized aggregation and inexpensive updates via column swapping. The performance advantage of our PostgreSQL baseline over SystemDS decreases with more missing values due to increased update overheads. SystemDS and DuckDB exploit the full parallelism of 64 threads, while PostgreSQL does sequential updates and limits its scan parallelism to 8 threads for the two datasets.

Figure 4 also shows the effectiveness of our computation sharing techniques (cf. Section 4). The Low implementation, tailored to datasets with low missing rates, reduces the per-iteration baseline cost by 3x in DuckDB and 4.5x in PostgreSQL on the Flight dataset with 5% of missing values, at the expense of increasing the one-off preprocessing costs; similar holds on Retailer (cf. Figures 4a and 4g). The results evidence that the Low implementation pays off on datasets containing up to 20% missing values. The High implementation, although initially designed for datasets with high missing rates, outperforms the baseline in both low and high missing rate scenarios: in the former, this is mainly due to precomputing partial cofactor aggregates over complete records outside the iteration loop, which reduces the iteration cost by 2.7x in DuckDB and 2.4x in PostgreSQL on Flight, and similarly on Retailer; in the latter, this is also due to processing data partitions that are smaller with more missing values, which makes the model training less expensive.



(a) Flight 5% missing *(b)* Flight 10% missing *(c)* Flight 20% missing

(d) Flight 40% missing *(e)* Flight 60% missing *(f)* Flight 80% missing

(g) Retailer 5% missing *(h)* Retailer 10% missing *(i)* Retailer 20% missing

(j) Retailer 40% missing *(k)* Retailer 60% missing *(l)* Retailer 80% missing

Fig. 4. Single-table imputation using MICE. The time needed to run a single round of MICE over Flight and Retailer with different percentages of missing values, divided into preprocessing time (done once) and iteration time (repeated every round). The baseline, low, and high versions denote our implementations described in Section 5.2. The red labels indicate a timeout.

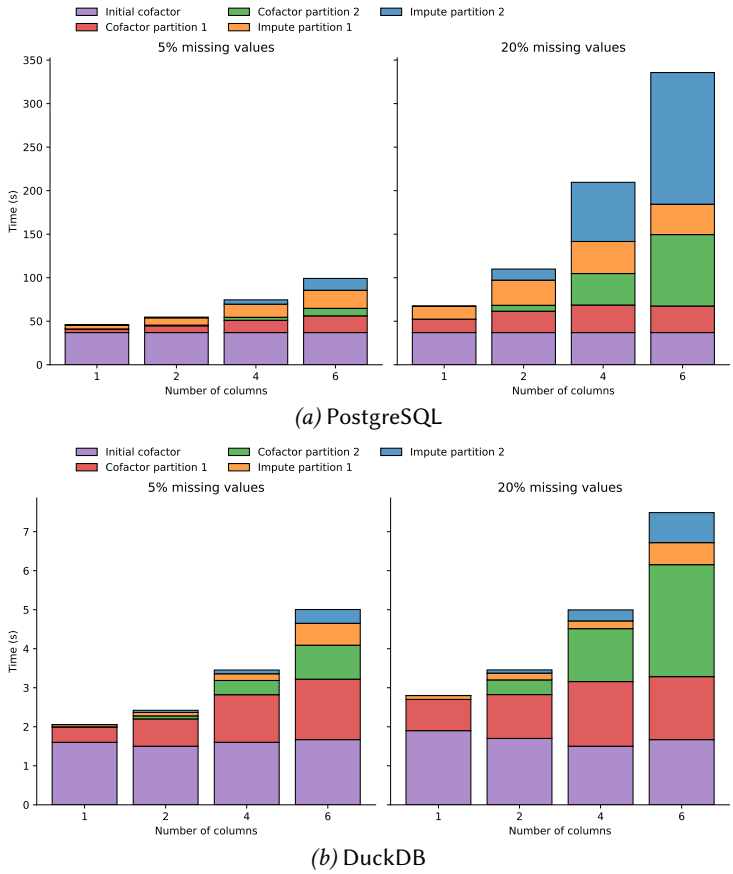
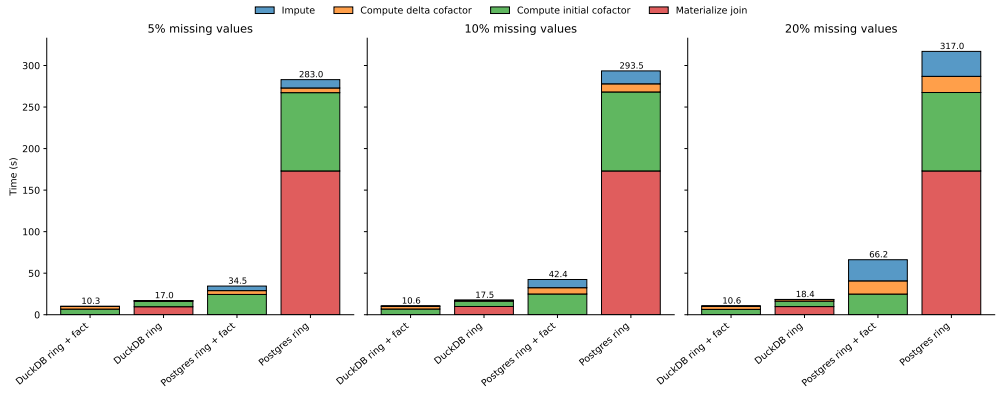


Fig. 5. Single-table imputation using the Low implementations with varying numbers of incomplete attributes. The runtime for a single round of MICE on the Flight dataset with randomly generated 5% and 20% of missing values in each column.

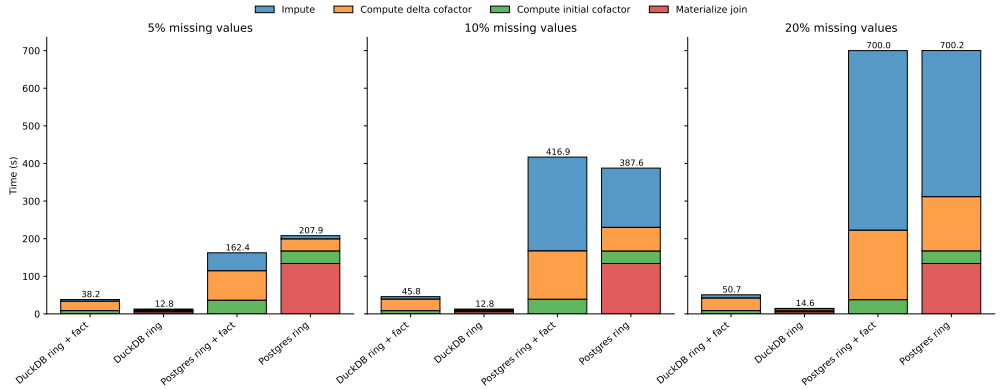
Varying the Number of Incomplete Attributes. We measure the impact of the number of attributes with missing values on the performance of our Low implementations. Figure 5 reports the breakdown of the runtime for both PostgreSQL and DuckDB over the Flight dataset as the number of incomplete attributes varies between 1 and 6. The runtime is split into five components: the initial time to compute the cofactor aggregates over the entire table, and, for each of the two affected partitions with missing values, the time needed to compute the cofactor aggregates in that partition, and the time needed to update the imputed values in that partition.

For both DBMSs, the figure shows a linear increase in the runtime with respect to the number of attributes with missing values, where the missingness ratio dictates the increase quantity. A higher number of columns with missing values also increases the size of the partition containing rows with at least two missing values, prolonging also the cofactor computation and imputation time.

PostgreSQL and DuckDB have different bottlenecks: in the former, updating imputed values is the slowest phase, while in the latter, it is the computation of cofactor aggregates. This is due to their different architectures. PostgreSQL uses a row-based engine with MVCC, which despite our optimizations, still causes significant overheads during the execution of update queries. DuckDB,



(a) Retailer dataset



(b) Flight dataset

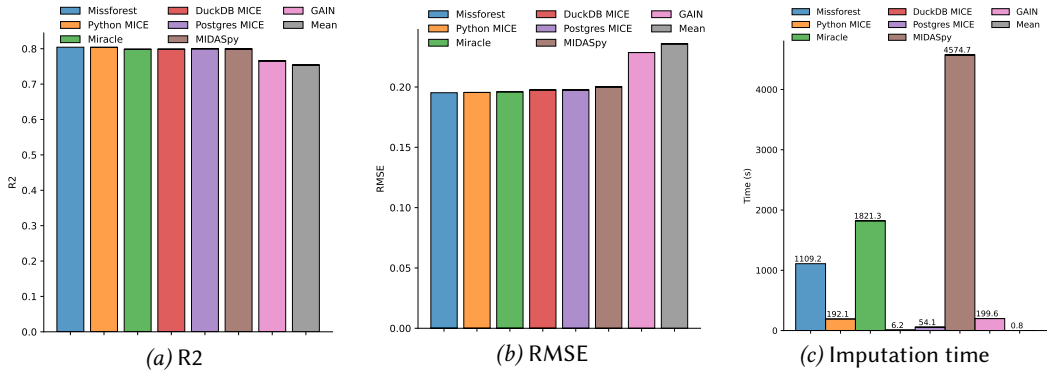
Fig. 6. Imputation over normalized data using the Low implementations with materialization of the join result and with factorized evaluation that avoids the join materialization. The runtime for a single round of MICE with DuckDB and PostgreSQL over Retailer and Flight with different percentages of missing values, randomly generated in the fact table of each dataset.

instead, adopts a column-oriented approach, which we exploit to implement lightweight column swapping to instantiate new imputed values.

6.3 Imputation over Normalized Data

We now consider the case when the dataset is normalized, and the imputation takes into account the attributes from all tables. We compare the performance of our Low implementations running over the materialized join result and its factorized version that pushes the aggregate computation past the joins. We use the Flight and Retailer datasets as they consist of multiple tables, and randomly generate missing values in the fact table only so that in both cases the algorithm generates the same imputations; otherwise, joining incomplete tables can create copies of the same missing value, which might be imputed differently by the two approaches.

Figure 6 shows the performance of the two approaches over a single MICE round for Retailer and Flight, consisting of 1 and 7 attributes with missing values, respectively. As previously discussed, in the Flight dataset, joining its tables does not introduce many redundant values, thus imputing values over a normalized database is slower than using the joined table. On the Retailer dataset,



(a) R2 *(b)* RMSE *(c)* Imputation time

Fig. 7. Imputation quality on the Air Quality dataset. The quality of a regression model that predicts the air quality index given attributes with missing values, measured by R2 (higher is better) and RMSE (lower is better). The time needed to impute the dataset using different imputation approaches.

instead, we get the opposite result. As the joined result is more redundant, imputing values directly over normalized relations in PostgreSQL is from 4.8 to 8.2x faster than using the joined result, mainly because of the long time to materialize the joined table and compute the cofactor aggregates over redundant values. The imputation time is also faster: PostgreSQL writes a full row when a new value is imputed, and updating a table with just 4 attributes is faster than updating 25 attributes of the joined table. DuckDB also imputes faster over the normalized dataset than over the joined dataset, albeit with a smaller improvement of 1.7x than PostgreSQL, mainly due to faster in-memory join materialization and aggregation.

6.4 Imputation Quality

We compare different imputation methods in terms of imputation quality, measured as the performance of a linear regression model trained over imputed data. We consider the following methods: MICE DuckDB and MICE PostgreSQL, our MICE implementations (Low versions); MICE Python, the MICE implementation from scikit-learn using linear and logistic regression; MissForest [65], which uses random forests; GAIN [69], which uses generative adversarial networks; MIRACLE [35], which uses causally-aware refinements, MIDASpy [37], which uses autoencoders; and mean/mode imputation. We use the Python implementation of the competitors from HyperImpute [25]. We use 5 iterations for the MICE methods.

Figure 7 shows the imputation quality over the Taiwan's Air Quality dataset, where the task is to predict the air quality index given its pollutants. Although the dataset contains only 6% of missing values, mean imputation performs significantly worse than the other methods, reducing the R2 score of the regression model by 0.05 and increasing its root mean squared error (RMSE) by 0.05. MICE outperforms GAIN and achieves similar performance to MissForest, MIDASpy, and MIRACLE. Figure 7c reports the imputation time of each method. MICE DuckDB is faster than GAIN by 32x, MICE Python by 38x, MIDASpy by 778x, and only 5.4 seconds slower than mean imputation in Python.

Missing Patterns. We analyze imputation quality under three widely-used mechanisms for injecting missing values: missing completely at random (MCAR), missing at random (MAR), and missing not at random (MNAR) [40]. To allow all competitors to finish within a 30-minute timeout, we restrict the Flight dataset to year 2015 (5M rows) and the Retailer dataset to two dates with ids 501 and 508 (1M rows). For the former, the task is to predict flight duration given predictors (e.g.,

Pattern	Method	Missing Rate						Imp. Time	
		0.05	0.1	0.2	0.4	0.6	0.8	secs	norm
MCAR	MICE DuckDB	0.180 ± 0.009	0.181 ± 0.002	0.161 ± 0.005	0.175 ± 0.016	0.173 ± 0.012	0.175 ± 0.021	5.4	1.0
	MICE Python	0.180 ± 0.005	0.183 ± 0.002	**0.158** ± 0.003	0.174 ± 0.010	0.172 ± 0.012	0.173 ± 0.018	189.6	34.9
	Mean	0.286 ± 0.009	0.359 ± 0.009	0.473 ± 0.008	0.642 ± 0.025	0.774 ± 0.012	0.889 ± 0.012	0.4	0.1
	MissForest	0.235 ± 0.004	0.275 ± 0.011	0.333 ± 0.005	0.431 ± 0.021	0.476 ± 0.005	0.461 ± 0.070	407.5	75.0
	GAIN	0.286 ± 0.038	0.350 ± 0.001	0.460 ± 0.015	0.633 ± 0.019	0.756 ± 0.020	0.864 ± 0.129	82.0	15.1
	MIRACLE	**0.179** ± 0.003	**0.180** ± 0.001	0.175 ± 0.003	**0.169** ± 0.002	**0.166** ± 0.001	**0.168** ± 0.002	788.4	145.1
MAR	MICE DuckDB	0.176 ± 0.021	0.175 ± 0.043	0.268 ± 0.013	**0.336** ± 0.078	**0.331** ± 0.009	0.344 ± 0.068	5.5	1.0
	MICE Python	**0.174** ± 0.016	**0.172** ± 0.039	**0.266** ± 0.012	0.337 ± 0.071	0.339 ± 0.012	**0.342** ± 0.038	186.6	34.2
	Mean	0.487 ± 0.035	0.597 ± 0.020	0.723 ± 0.001	0.845 ± 0.005	0.913 ± 0.002	0.958 ± 0.004	0.4	0.1
	MissForest	0.367 ± 0.003	0.431 ± 0.013	0.510 ± 0.001	0.557 ± 0.095	0.499 ± 0.006	0.486 ± 0.081	415.5	76.1
	GAIN	0.384 ± 0.137	0.485 ± 0.092	0.674 ± 0.129	0.868 ± 0.133	0.917 ± 0.179	0.923 ± 0.001	81.0	14.8
	MIRACLE	0.191 ± 0.025	0.217 ± 0.036	0.257 ± 0.007	0.397 ± 0.077	0.538 ± 0.044	0.632 ± 0.536	785.6	143.9
MNAR	MICE DuckDB	0.194 ± 0.011	0.193 ± 0.018	0.199 ± 0.037	0.280 ± 0.023	0.203 ± 0.041	0.231 ± 0.068	5.5	1.0
	MICE Python	0.192 ± 0.008	0.195 ± 0.021	0.184 ± 0.029	0.279 ± 0.021	0.210 ± 0.043	0.229 ± 0.053	186.2	33.9
	Mean	0.289 ± 0.012	0.365 ± 0.034	0.475 ± 0.004	0.651 ± 0.065	0.779 ± 0.039	0.893 ± 0.039	0.4	0.1
	MissForest	0.239 ± 0.014	0.280 ± 0.010	0.347 ± 0.073	0.436 ± 0.017	0.487 ± 0.045	0.490 ± 0.086	553.6	100.8
	GAIN	0.284 ± 0.039	0.355 ± 0.089	0.463 ± 0.048	0.633 ± 0.058	0.759 ± 0.110	0.898 ± 0.110	79.4	14.5
	MIRACLE	**0.186** ± 0.007	**0.191** ± 0.037	**0.182** ± 0.009	**0.185** ± 0.099	**0.200** ± 0.099	**0.223** ± 0.102	795.4	144.8

(a) Flight

Pattern	Method	Missing Rate						Imp. Time	
		0.05	0.1	0.2	0.4	0.6	0.8	secs	norm
MCAR	MICE DuckDB	**0.228** ± 0.013	0.241 ± 0.024	0.249 ± 0.029	0.254 ± 0.009	0.283 ± 0.036	0.296 ± 0.013	5.7	1.0
	MICE Python	0.233 ± 0.009	**0.237** ± 0.016	**0.248** ± 0.026	**0.252** ± 0.004	**0.282** ± 0.038	**0.294** ± 0.015	128.3	22.6
	Mean	0.316 ± 0.015	0.382 ± 0.045	0.491 ± 0.015	0.645 ± 0.010	0.755 ± 0.022	0.841 ± 0.011	0.8	0.1
	MissForest	0.267 ± 0.011	0.304 ± 0.002	0.361 ± 0.028	0.446 ± 0.001	0.519 ± 0.046	0.558 ± 0.010	212.7	37.5
	GAIN	0.294 ± 0.003	0.357 ± 0.149	0.449 ± 0.103	0.571 ± 0.128	0.673 ± 0.253	0.735 ± 0.010	81.0	14.3
	MIRACLE	0.268 ± 0.044	0.320 ± 0.102	0.378 ± 0.113	0.611 ± 0.133	0.749 ± 0.215	0.791 ± 0.271	995.1	174.5
MAR	MICE DuckDB	**0.226** ± 0.006	0.242 ± 0.009	**0.245** ± 0.023	0.261 ± 0.019	0.277 ± 0.016	**0.296** ± 0.028	5.7	1.0
	MICE Python	0.234 ± 0.005	**0.239** ± 0.008	0.247 ± 0.019	**0.260** ± 0.019	**0.275** ± 0.013	0.299 ± 0.029	125.7	22.1
	Mean	0.316 ± 0.022	0.384 ± 0.057	0.489 ± 0.026	0.638 ± 0.007	0.740 ± 0.028	0.841 ± 0.027	0.8	0.1
	MissForest	0.268 ± 0.008	0.303 ± 0.006	0.360 ± 0.026	0.449 ± 0.029	0.520 ± 0.009	0.551 ± 0.031	227.1	39.9
	GAIN	0.294 ± 0.019	0.352 ± 0.073	0.459 ± 0.025	0.595 ± 0.171	0.670 ± 0.054	0.744 ± 0.031	78.4	13.8
	MIRACLE	0.251 ± 0.075	0.378 ± 0.167	0.342 ± 0.021	0.595 ± 0.219	0.783 ± 0.013	0.823 ± 0.219	984.1	173.8
MNAR	MICE DuckDB	**0.227** ± 0.061	0.239 ± 0.073	**0.245** ± 0.061	0.258 ± 0.069	**0.279** ± 0.018	0.296 ± 0.037	5.8	1.0
	MICE Python	0.232 ± 0.056	**0.238** ± 0.069	0.249 ± 0.065	**0.256** ± 0.064	**0.279** ± 0.015	**0.294** ± 0.034	127.0	22.0
	Mean	0.309 ± 0.149	0.365 ± 0.176	0.468 ± 0.152	0.615 ± 0.184	0.698 ± 0.136	0.824 ± 0.191	0.8	0.1
	MissForest	0.263 ± 0.081	0.296 ± 0.133	0.350 ± 0.165	0.429 ± 0.167	0.487 ± 0.021	0.539 ± 0.166	255.4	44.3
	GAIN	0.283 ± 0.053	0.337 ± 0.056	0.423 ± 0.229	0.552 ± 0.280	0.656 ± 0.041	0.732 ± 0.018	78.7	13.6
	MIRACLE	0.228 ± 0.075	0.256 ± 0.039	0.441 ± 0.104	0.601 ± 0.161	0.670 ± 0.159	0.850 ± 0.156	997.1	172.3

(b) Retailer

Fig. 8. Imputation quality and runtime for different missing rates and patterns on the restricted Flight and Retailer datasets. The quality is measured by the RMSE of a regression model predicting flight duration for Flight and inventory stock for Retailer from the imputed data. The imputation times (absolute and normalized to MICE DuckDB) are for the missing rate of 20%. The procedure of injecting and imputing missing values is repeated three times, with the 95% confidence intervals for RMSE being reported.

distance between airports); for the latter, the task is to predict inventory stock given predictors (e.g., population in the area).

We use the generator from HyperImpute [25] to inject missing values in 7 attributes in each dataset. With MCAR, we generate missing values with a uniform distribution. With MAR, the fraction of missing values depends on flight duration in Flight and inventory stock in Retailer. With MNAR, we generate missing values taking all 7 incomplete attributes as input for each dataset.

Figure 8 reports the performance of these imputation methods on the two restricted datasets. We repeat the procedure of injecting and imputing missing values three times and report the 95% confidence interval for RMSE. On the Flight dataset, the MICE methods and MIRACLE outperform the other methods in terms of quality, achieving the lowest RMSEs under all three patterns and all missing rates. But in terms of imputation time, MICE DuckDB is 145x faster than MIRACLE and 35x faster than MICE Python. GAIN, a deep learning approach, generates increasingly worse imputations as the fraction of missing values increases. Mean imputation performs the worst in terms of imputation quality in all scenarios. We observe similar trends on the Retailer dataset in terms of MICE's runtime performance and imputation quality.

7 RELATED WORK

Data Imputation. Simple techniques for handling missing data, such as removing tuples with missing values, mean/mode imputation, indicator imputation, and Last Observation Carried Forward [36], are fast but only work under restrictive assumptions, produce biased analytics, and might distort the value distribution.

Model-based imputation aims to address these issues. This includes approaches that learn the joint distribution of data based on matrix completion [44], generative adversarial networks [69, 70], diffusion models [73], autoencoders [43, 53], and the EM algorithm [40]. Discriminative approaches to data imputation include deep denoising autoencoders [37], graph neural networks [71], and conditional modeling strategies such as MICE [68] and MissForest [65]. Specific imputation methods are designed for particular domains, such as time-series [8, 17, 29, 30] and non-numerical data [11]. Imputation methods can also be based on multiple models: HoloClean [57] does broader data cleaning using statistical analysis, integrity constraints, and external data, while MIRACLE [35] learns both imputation function and missingness graph.

While these methods offer high imputation quality, they also require external tools, can only impute values in a single table, and have a long imputation time, which hinders their applicability on large datasets: most of them consider datasets with at most few hundreds of thousands records, while their imputation times range from 1.7K to 96K seconds on a 1M dataset [46]. Some systems try to avoid lengthy imputation. SampleClean [32] estimates the result of a query over dirty data by cleaning only a sample of the dataset. ImputeDB [14] incorporates the imputation process into the query optimizer, imputing only the relevant data; this can generate biased models. EDIT [46] addresses the high cost of training deep learning models by reducing the training dataset, estimating the importance of each sample. Such importance estimates, however, are often erroneous for deep networks [9].

Compared to simple data imputation techniques, our approach offers a similar runtime while providing significantly better imputation quality. With respect to model-based imputation tools, our approach avoids the problem of moving data between different systems, imputes data directly over multiple tables, and significantly improves the imputation time. Compared to hybrid systems such as SampleClean, ImputeDB and EDIT, our approach does not reduce the dataset size to improve performance and does not require preprocessing such as joining relations or one-hot encoding.

Systems for Treating Missing Values. There are systems that try to adapt tasks to work over incomplete databases, such as entity resolution [58], discovery of functional dependencies [10], or machine learning training [27, 41, 55]. Other systems try to repair missing values using different approaches, such as human supervision [42, 48], functional dependencies [59], ensemble of tools [1],

by generating interpretable data repair rules [54], or synthesizing completely missing tuples over database schema [22]. Missing values might also not be clearly recognizable, prompting systems for the discovery of hidden missing values [56] and the evaluation of the quality of incomplete databases [39, 61].

In-Database Learning. This line of work aims to enable machine learning tasks within a database system, delivering faster training and prediction times, lower data movement and storage costs, and better data security. MADLib [21] and Bismarck [19] implement ML tasks inside user-defined functions, thus removing the need for data movementm, but without exploiting the structure of the data to improve learning performance. LMFAO [63] and AC/DC [28] support learning various ML models over normalized tables by sharing computation, Orion [34] supports generalized linear models, and Morpheus [15] supports linear algebra operations over a normalized database. F-IVM [49, 50] introduces the cofactor ring to support incremental computation of ML models.

Compared to this prior work, we make several novel contributions: we present a new method for in-database training of linear discriminant analysis models, we adapt the widely-used MICE algorithm to exploit computation sharing, and instead of developing specialized tools, we implement our approach in two popular database systems, offering high data imputation quality at low cost.

8 CONCLUSION

This article introduces a novel and efficient method for generating high-quality data imputation. We show how to enhance the widely-used MICE algorithm to exploit computation sharing and scalable execution, tailoring it specifically for the implementation within a database system. We implement our imputation methods in DuckDB and PostgreSQL and demonstrate their superiority when compared to other methods, resulting in faster imputation time without compromising the quality of imputed data.

Looking ahead, we plan to enrich our open-source library with new methods for in-database learning and data cleaning, offering data practitioners a powerful tool to effectively address the challenges presented by missing data. Our library already supports other ML algorithms not elaborated in this article, such as Naïve Bayes and Quadratic Discriminant Analysis classifiers. We believe it is possible to further empower in-database data cleaning tasks with learning algorithms such as decision trees, k-means, random forests, and gradient boosting [15, 23, 62], including their integration into MICE, albeit the opportunities for computation sharing across models and iterations might not always exist. In that context, we want to pursue support for shared and incremental learning of models in iterative database tasks, like in this work. Going further, deep learning and deep generative models pose a significant challenge at the moment for a fully in-database implementation due to their non-linearity, thus efficiently training such high-capacity models over relational databases stands as an open problem.

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where Are We and What Needs to Be Done? *Proc. VLDB Endow.* 9, 12 (2016), 993–1004. https://doi.org/10.14778/2994509.2994518

[2] Taiwan Environmental Protection Administration. 2016 - 2021. Taiwan Air Quality Data. https://data.gov.tw/dataset/40448

[3] Gareth Ambler, Rumana Z Omar, and Patrick Royston. 2007. A Comparison of Imputation Techniques for Handling Missing Predictor Values in a Risk Model with a Binary Outcome. *Statistical Methods in Medical Research* 16, 3 (2007), 277–298. https://doi.org/10.1177/0962280206074466

[4] Rebecca R. Andridge and Roderick J. A. Little. 2010. A Review of Hot Deck Imputation for Survey Non-response. *International Statistical Review* 78, 1 (2010), 40–64. https://doi.org/10.1111/j.1751-5823.2010.00103.x

[5] E. Angerson, D. Sorensen, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, and C. Bischof. 1990. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Supercomputing*.

2–11. https://doi.org/10.1109/SUPERC.1990.129995
[6] Apache MADlib. 2023. https://madlib.apache.org.
[7] Apache SystemDS. 2023. https://systemds.apache.org.
[8] Parikshit Bansal, Prathamesh Deshpande, and Sunita Sarawagi. 2021. Missing Value Imputation on Multidimensional Time Series. *Proc. VLDB Endow.* 14, 11 (2021), 2533–2545. https://doi.org/10.14778/3476249.3476300
[9] Samyadeep Basu, Phillip Pope, and Soheil Feizi. 2021. Influence Functions in Deep Learning Are Fragile. In *ICLR*.
[10] Laure Berti-Équille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Saravanan Thirumuruganathan. 2018. Discovery of Genuine Functional Dependencies from Relational Data with Missing Values. *Proc. VLDB Endow.* 11, 8 (2018), 880–892. https://doi.org/10.14778/3204028.3204032
[11] Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. 2018. "Deep" Learning for Missing Value Imputationin Tables with Non-Numerical Data. In *CIKM*. 2017–2025. https://doi.org/10.1145/3269206.3272005
[12] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (2016), 1425–1436. https://doi.org/10.14778/3007263.3007279
[13] Stef van Buuren. 2018. *Flexible Imputation of Missing Data* (2nd ed.). CRC Press. https://doi.org/10.1201/9780429492259
[14] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. 2017. Query Optimization for Dynamic Imputation. *Proc. VLDB Endow.* 10, 11 (2017), 1310–1321. https://doi.org/10.14778/3137628.3137641
[15] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (2017), 1214–1225. https://doi.org/10.14778/3137628.3137633
[16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *SIGMOD*. 2201–2206. https://doi.org/10.1145/2882903.2912574
[17] Xiaoou Ding, Hongzhi Wang, Jiaxuan Su, Zijue Li, Jianzhong Li, and Hong Gao. 2019. Cleanits: A Data Cleaning System for Industrial Time Series. *Proc. VLDB Endow.* 12, 12 (2019), 1786–1789. https://doi.org/10.14778/3352063.3352066
[18] Tlamelo Emmanuel, Thabiso Maupong, Dimane Mpoeleng, Thabo Semong, Banyatsang Mphago, and Oteng Tabona. 2021. A Survey on Missing Data in Machine Learning. *Journal of Big Data* 8, 1 (2021), 1–37. https://doi.org/10.1186/s40537-021-00516-9
[19] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for In-RDBMS Analytics. In *SIGMOD*. 325–336.
[20] John W. Graham. 2009. Missing Data Analysis: Making It Work in the Real World. *Annual Review of Psychology* 60, 1 (2009), 549–576. https://doi.org/10.1146/annurev.psych.58.110405.085530
[21] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510
[22] Benjamin Hilprecht and Carsten Binnig. 2021. ReStore – Neural Data Completion for Relational Databases. In *SIGMOD*. 710–722. https://doi.org/10.1145/3448016.3457264
[23] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees Over Normalized Data Using Only SQL. arXiv:2307.00422 [cs.DB]
[24] Md Hamidul Huque, John B. Carlin, Julie A. Simpson, and Katherine J. Lee. 2018. A Comparison of Multiple Imputation Methods for Missing Data in Longitudinal Studies. *BMC Medical Research Methodology* 18, 1 (2018), 168. https://doi.org/10.1186/s12874-018-0615-6
[25] Daniel Jarrett, Bogdan Cebere, Tennison Liu, Alicia Curth, and Mihaela van der Schaar. 2022. HyperImpute: Generalized Iterative Imputation with Automatic Model Selection. In *ICML*. 9916–9937.
[26] Sebastian Jäger, Arndt Allhorn, and Felix Bießmann. 2021. A Benchmark for Data Imputation Methods. *Frontiers in Big Data* 4 (2021), 1–16. https://doi.org/10.3389/fdata.2021.693674
[27] Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *Proc. VLDB Endow.* 14, 3 (2020), 255–267. https://doi.org/10.14778/3430915.3430917
[28] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *DEEM*. 1–10. https://doi.org/10.1145/3209889.3209896
[29] Mourad Khayati, Ines Arous, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. ORBITS: Online Recovery of Missing Values in Multiple Time Series Streams. *Proc. VLDB Endow.* 14, 3 (2020), 294–306. https://doi.org/10.14778/3430915.3430920
[30] Mourad Khayati, Alberto Lerner, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. Mind the Gap: An Experimental Evaluation of Imputation of Missing Values Techniques in Time Series. *Proc. VLDB Endow.* 13, 5 (2020), 768–782. https://doi.org/10.14778/3377369.3377383

[31] Christoph Koch. 2010. Incremental Query Evaluation in a Ring of Databases. In *PODS*. 87–98. https://doi.org/10.1145/1807085.1807100
[32] S. Krishnan, Jiannan Wang, M. Franklin, Ken Goldberg, T. Kraska, T. Milo, and E. Wu. 2015. SampleClean: Fast and Reliable Analytics on Dirty Data. *IEEE Data Eng. Bull.* 38 (2015), 59–75.
[33] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*. 1717–1722. https://doi.org/10.1145/3035918.3054775
[34] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. , 1969-1984 pages. https://doi.org/10.1145/2723372.2723713
[35] Trent Kyono, Yao Zhang, Alexis Bellot, and Mihaela van der Schaar. 2021. MIRACLE: Causally-Aware Imputation via Learning Missing Data Mechanisms. *Advances in Neural Information Processing Systems* 34 (2021), 23806–23817.
[36] John M Lachin. 2016. Fallacies of Last Observation Carried Forward Analyses. *Clinical Trials* 13, 2 (2016), 161–168. https://doi.org/10.1177/1740774515602688
[37] Ranjit Lall and Thomas Robinson. 2022. The MIDAS Touch: Accurate and Scalable Missing-Data Imputation with Deep Learning. *Political Analysis* 30, 2 (2022), 179–196. https://doi.org/10.1017/pan.2020.49
[38] P.-A. Larson. 2002. Data Reduction by Partial Preaggregation. In *ICDT*. 706–715. https://doi.org/10.1109/ICDE.2002.994787
[39] Xi Liang, Zechao Shang, Sanjay Krishnan, Aaron J. Elmore, and Michael J. Franklin. 2020. Fast and Reliable Missing Data Contingency Analysis with Predicate-Constraints. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 285–295. https://doi.org/10.1145/3318464.3389785
[40] Roderick J. A. Little and Donald B. Rubin. 2019. *Statistical Analysis with Missing Data* (3rd ed.). Wiley. https://doi.org/10.1002/9781119482260
[41] Tongyu Liu, Ju Fan, Yinqing Luo, Nan Tang, Guoliang Li, and Xiaoyong Du. 2021. Adaptive Data Augmentation for Supervised Learning over Missing Data. *Proc. VLDB Endow.* 14, 7 (2021), 1202–1214. https://doi.org/10.14778/3450980.3450989
[42] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *Proc. VLDB Endow.* 13, 12 (2020), 1948–1961. https://doi.org/10.14778/3407790.3407801
[43] Pierre-Alexandre Mattei and Jes Frellsen. 2019. MIWAE: Deep Generative Modelling and Imputation of Incomplete Data Sets. In *ICML*. 4413–4423.
[44] Rahul Mazumder, Trevor Hastie, and Robert Tibshirani. 2010. Spectral Regularization Algorithms for Learning Large Incomplete Matrices. *Journal of Machine Learning Research* 11, 80 (2010), 2287–2322.
[45] Maritza Mera-Gaona, Ursula Neumann, Rubiel Vargas-Canas, and Diego M. López. 2021. Evaluating the Impact of Multivariate Imputation by MICE in Feature Selection. *PLOS ONE* 16, 7 (2021), 1–28. https://doi.org/10.1371/journal.pone.0254720
[46] Xiaoye Miao, Yangyang Wu, Lu Chen, Yunjun Gao, Jun Wang, and Jianwei Yin. 2021. Efficient and Effective Data Imputation with Influence Functions. *Proc. VLDB Endow.* 15, 3 (2021), 624–632. https://doi.org/10.14778/3494124.3494143
[47] MindsDB. 2023. https://mindsdb.com/.
[48] Mashaal Musleh, Mourad Ouzzani, Nan Tang, and AnHai Doan. 2020. CoClean: Collaborative Data Cleaning. In *SIGMOD*. 2757–2760. https://doi.org/10.1145/3318464.3384698
[49] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*. 365–380. https://doi.org/10.1145/3183713.3183758
[50] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. 2020. F-IVM: Learning over Fast-Evolving Relational Data. In *SIGMOD*. 2773–2776. https://doi.org/10.1145/3318464.3384702
[51] U.S. Department of Transportation. 2015. Flight Delays and Cancellations. https://www.kaggle.com/datasets/usdot/flight-delays
[52] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, null (2011), 2825–2830.
[53] Ignacio Peis, Chao Ma, and José Miguel Hernández-Lobato. 2022. Missing Data Imputation and Acquisition with Deep Hierarchical Models and Hamiltonian Monte Carlo. *Advances in Neural Information Processing Systems* 35 (2022), 35839–35851.
[54] Jinfeng Peng, Derong Shen, Nan Tang, Tieying Liu, Yue Kou, Tiezheng Nie, Hang Cui, and Ge Yu. 2022. Self-Supervised and Interpretable Data Cleaning with Sequence Generative Adversarial Networks. *Proc. VLDB Endow.* 16, 3 (2022), 433–446. https://doi.org/10.14778/3570690.3570694
[55] Jose Picado, John Davis, Arash Termehchy, and Ga Young Lee. 2020. Learning Over Dirty Data Without Cleaning. In *SIGMOD*. 1301–1316. https://doi.org/10.1145/3318464.3389708

[56] Abdulhakim A. Qahtan, Ahmed Elmagarmid, Raul Castro Fernandez, Mourad Ouzzani, and Nan Tang. 2018. FAHES: A Robust Disguised Missing Values Detector. In *KDD*. 2100–2109. https://doi.org/10.1145/3219819.3220109
[57] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631
[58] Weilong Ren, Xiang Lian, and Kambiz Ghazinour. 2021. Online Topic-Aware Entity Resolution Over Incomplete Data Streams. In *SIGMOD*. 1478–1490.
[59] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-Driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (2021), 2546–2554. https://doi.org/10.14778/3476249.3476301
[60] Joseph L. Schafer and John W. Graham. 2002. Missing Data: Our View of the State of the Art. *Psychological Methods* 7, 2 (2002), 147–177. https://doi.org/10.1037/1082-989X.7.2.147
[61] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (2018), 1781–1794. https://doi.org/10.14778/3229863.3229867
[62] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An Engine for Batches of Group-by Aggregates: Layered Multiple Functional Aggregate Optimization. *Proc. VLDB Endow.* 13, 12 (2020), 2945–2948. https://doi.org/10.14778/3415478.3415515
[63] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD*. 1642–1659. https://doi.org/10.1145/3299869.3324961
[64] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*. 3–18. https://doi.org/10.1145/2882903.2882939
[65] Daniel J. Stekhoven and Peter Bühlmann. 2012. MissForest – Non-Parametric Missing Value Imputation for Mixed-Type Data. *Bioinformatics* 28, 1 (2012), 112–118. https://doi.org/10.1093/bioinformatics/btr597
[66] Julia Stoyanovich, Bill Howe, and H. V. Jagadish. 2020. Responsible Data Management. *Proc. VLDB Endow.* 13, 12 (2020), 3474–3488. https://doi.org/10.14778/3415478.3415570
[67] Etienne Toussaint, Paolo Guagliardo, Leonid Libkin, and Juan Sequeda. 2022. Troubles with Nulls, Views from the Users. *Proc. VLDB Endow.* 15, 11 (2022), 2613–2625. https://doi.org/10.14778/3551793.3551818
[68] Stef van Buuren and Karin Groothuis-Oudshoorn. 2011. mice: Multivariate Imputation by Chained Equations in R. *Journal of Statistical Software* 45, 3 (2011), 1–67. https://doi.org/10.18637/jss.v045.i03
[69] Jinsung Yoon, James Jordon, and Mihaela van der Schaar. 2018. GAIN: Missing Data Imputation using Generative Adversarial Nets. In *ICML*, Vol. 80. 5689–5698.
[70] Seongwook Yoon and Sanghoon Sull. 2020. GAMIN: Generative Adversarial Multiple Imputation Network for Highly Missing Data. In *CVPR*. 8453–8461.
[71] Jiaxuan You, Xiaobai Ma, Yi Ding, Mykel J Kochenderfer, and Jure Leskovec. 2020. Handling Missing Data with Graph Representation Learning. *Advances in Neural Information Processing Systems* 33 (2020), 19075–19087.
[72] Yang C Yuan. 2010. Multiple Imputation for Missing Data: Concepts and New Development (Version 9.0). *SAS Institute Inc, Rockville, MD* 49, 1-11 (2010), 12.
[73] Shuhan Zheng and Nontawat Charoenphakdee. 2022. Diffusion Models for Missing Value Imputation in Tabular Data. In *NeurIPS 2022 First Table Representation Workshop*.

Received July 2023; revised October 2023; accepted November 2023