Large-scale Data Generation for Benchmarking Data Cleaning Tools

Olga Ovcharenko

Bachelor Thesis

to achieve the university degree of

Bachelor of Science

Bachelor's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dr.-Ing. Matthias Boehm

Institute of Interactive Systems and Data Science

Graz, August 2022

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present bachelor's thesis.

Date	Signature

Abstract

Data preparation is a crucial part of the data science lifecycle. Most state-of-the-art tools and frameworks contain data cleaning functionality, but many of them require manual effort to prepare the input or to define constraints. There is a lack of large real-world datasets with ground truth that can be used for benchmarking data cleaning frameworks, e.g., for evaluation of results. The best existing solution is the generation of synthetic data, but it does not represent the real-world problems and errors. It would be interesting and helpful for the community if one could cheaply generate new large datasets from small samples of dirty and clean data. Since any non-trivial data sizes lead to new challenges in both data cleaning and full Machine Learning (ML) pipelines. Hence, this work focuses on data generation, observations of the original clean and dirty data are used to scale up to arbitrary data sizes. The biggest scaling factor achieved is 65,536x, and the largest data size achieved was 18 GB. These results were achieved under the constraints of preserving data statistics and the error distribution of the original dirty dataset, and new algorithms were applied. The generator can run either locally, or distributed via Apache Spark. As expected, the local execution is faster up to 42x for small scaling factors, while, on the other hand, when scaling to larger sizes the distributed execution shows better performance and scalability for data sizes that would not fit in single node memory. It was challenging to introduce errors preserving the characteristics of the original dataset, and to scale to arbitrary sizes. The proposed framework solves the problem of the large-scale data generation, and provides data generation for benchmarking cleaning for ML.

Contents

1 Introduction **1**

2 Background and Related Work **3**
2.1 Data Cleaning Problems 3
2.2 Data Cleaning Tools 4
2.3 Existing Benchmarks 6
2.4 Limitations of Existing Work 8

3 Data Generation Framework **9**
3.1 Types of Errors 9
3.2 Data Generation 11
3.2.1 Local Data Generation 12
3.2.2 Distributed Data Generation 16
3.3 Validation of Results 18

4 Experiments **21**
4.1 Experimental Setup 21
4.2 Data Characteristics 21
4.2.1 Datasets 21
4.2.2 Data and Error Characteristics 23
4.3 Runtime Experiments 31
4.3.1 Local Experiments 31
4.3.2 Distributed Experiments 31

5 Conclusions **33**

Bibliography **34**

1 Introduction

Data integration, cleaning and preparation are a part of the data science lifecycle. This lifecycle is an exploratory process that takes a lot of effort, time and experiments. Data engineers and data scientists in general spend between 80% and 90% of their time just on finding, collecting, integrating and cleaning datasets [12, 59]. The remaining 10%-20% are spent on model selection, training, scoring, and deployment. Accordingly, data cleaning is a huge challenge in Databases (DB) and Machine Learning (ML) fields. Both communities have been working on solutions of problems related to dirty data [37, 20, 19]. The ML community concentrates on impact of dirty data and noise on ML models [37]. It was studied that noise can have negligible or positive effort on accuracy [2, 39] since noise can act as a regularizer, but ML models can also be sensitive to the dirty data, especially dirty labels [16]. The DB community's focus is on error detection and repairing [24, 13].

Data preparation and its challenges: Data cleaning is the process of replacing or removing faulty values from a dataset. It consists of two steps: error detection and repair. Error detection aims to identify errors, while error repair imputes/fixes flawed values using knowledge gathered from the clean input data. There are many existing techniques [13, 22] and frameworks to clean the data such as HoloDetect/HoloClean [23, 50], Raha/Baran [43, 41], BoostClean [33], AlphaClean [31], ActiveClean [32]. Existing techniques and frameworks are limited by additional user input such as hand-crafted constraints [3], or manual labeling (by users) [43, 41]. For evaluating data cleaning approaches the ground truth (GT) is required, unless other signals such as the performance on an ML application exist. GT is not always available and is time-consuming to collect or create. Sufficient data cleaning often requires an extra user input such as hand-crafted constraints [3]. Additionally, many datasets are private or confidential. Limited or even missing access to the raw data makes construction or acquisition of the ground truth extremely difficult or sometimes even impossible.

Data cleaning in ML pipelines: Data cleaning is essential for the performance of ML models or databases. It is part of the ML pipeline that aims to automate machine learning workflow. ML pipeline consists of several steps: data collection [38], data cleaning [43, 41, 50, 23, 32, 31, 37, 20], feature extraction [56], model training [50], model debugging [53, 30, 36, 28, 27],

model evaluation [17, 52], model visualization [46, 9, 58], and model serving [47, 35, 61]. Data preparation is a key step in ML pipeline and obeys the garbage in, garbage out principle (GIGO). GIGO is the concept that flawed input data produces nonsense output.

Contributions: This paper aims to create a tool for the distributed dirty data generation that can be used by the ML and DB communities. The high level list of contributions is:

- A study of existing data cleaning tools and frameworks.
- Local and Spark distributed scaling that allows generating data at a scale beyond single node memory.
- Techniques for error type detection and error extraction.
- Data generation with configurable errors and error rates.
- Novel techniques to preserve the statistical properties of the input data while generating new data.
- Experimental evaluation of the data generator.

Finally, this data generation framework is a part of a larger project called WashHouse, a data cleaning for ML benchmark, that is designed to solve the problem of evaluating data cleaning approaches, and the problem of the missing ground truth datasets.

2 Background and Related Work

This chapter presents the background and related work. It starts with Section 2.1 describing general problems in the area of data cleaning. In Section 2.2, current state-of-the-art cleaning tools are presented and compared. Although, data cleaning is not the main goal of this work, results of the data generator are intended to be used for the evaluation of data cleaning tools. Section 2.3 and Section 2.4 describe existing benchmarks and limitations of the existing work respectively.

2.1 Data Cleaning Problems

The fundamental problem of the data cleaning is data validity. Data validation refers to the process of ensuring the quality of data, its fairness and accuracy. Invalid, inconsistent data can bring a larger stream of issues - for instance the Garbage in, Garbage out (GIGO) concept. Additionally, dirty data leads to wasted resources, extra cost, and time losses.

Dirty data origin: Data cleaning can be divided into error detection and repair of inconsistencies to improve data quality. Errors in data appear due to:

- Heterogeneous data sources such as federated database systems or data warehouses (update anomalies on denormalized data, inconsistencies, multi-modal data).
- Human errors (errors made during manual data collection and due to manually crafted data extractors, bias, missing or default values).
- Measurement / processing errors that are mostly caused by unreliable tools.

Error detection and repair problems: Common data cleaning problems are presented in Table 2.1. We differentiate between error scopes: attribute/value, observation/record, feature or between multiple features. A huge effort has been made in the research community to automate both error detection in combination with repairing broken records. It can not always be guaranteed that the cleaning performed on the dataset is sufficient. In some cases, there is no sufficient probabilistic evidence that repairing the data is better than

Table 2.1: Common cleaning problems

Problem	**Example**
Duplicates	Name: *Jane Smith* and *Smith Jane*
Uniqueness violation	Name: *Jane Smith*, insurance: *123* Name: *Kate James*, insurance: *123*
Violation of integrity constraints	DoB: 11-77-1975
Typos	Country: *Morrrocco*
Outliers	30 m tall person
Missing values	Missing cell in an observation: NA, NaN, default value

ignoring errors for specific cells [50]. Additionally, for most of the real-world datasets, the ground truth is unknown, and manual effort and domain knowledge are needed to establish it.

Principle of minimality: An important part of data cleaning is the principle of minimality [7, 50]. If there are two consistent repairs, we aim to pick the repair with fewer value changes as it's more likely to be correct and less likely to violate integrity constraints. There is a number of mechanisms / approaches to detect and fix these issues [34]. Thus, data should be analyzed, modeled, enriched, validated, and debugged.

2.2 Data Cleaning Tools

A huge effort has been made in the research community to automate data cleaning and validation. Many researchers have injected errors into the data with ground truth to experiment and evaluate their cleaning pipelines. In this section, existing frameworks, their differences, advantages, and disadvantages are discussed. Error detection is typically limited to a specific problem (e.g., duplicates, integrity constraints violation), and data repairing methods are often not introducing correct repairs [50].

HoloClean/HoloDetect: HoloClean [50] is a framework for holistic data repairing driven by probabilistic inference. The framework introduces an approach for combining heterogeneous data cleaning methods. It generates a probabilistic graphical model that uses input denial constraints and matching dependencies, capturing the uncertainty over observations in the input dataset. Then, the probabilistic inference is used to reason about inconsis-

tencies and repair them with most probable values. HoloDetect [23] is the follow up work. HoloDetect is a framework for the error detection. Correct and erroneous observations are generated with data augmentation and few-shot learning, and then used as input for a classifier to detect whether a value is an error.

Raha/Baran: Another error detection system is Raha [43], and Baran [41]. Here errors are detected and corrected with an ML ensemble of existing algorithms, rules, and constraints. Users need to label clusters of noisy data where each cluster represents an implicit type of data errors. These labels are then used to classify dirty and clean values. The main limitation of Raha/Baran is that the user label propagation is required for the classifier. Moreover, the system requires to run many algorithms/configurations that can be very extensive, and even with pruning, there is no guarantee for the best performance.

ActiveClean and BoostClean: ActiveClean [32], a model training framework that allows cleaning for iterative data while preserving monotone convergence guarantee. Samples of data are suggested to clean based on the likelihood of being a dirty value. ActiveClean concentrates on the repair of the specific error types such as domain values violations. But Sympson's Paradox [32] defines that any subset of a dataset is not guaranteed to represent the whole dataset. This means that sample distribution is not always enough to judge about the distribution of the whole data. BoostClean [33] is another framework that automatically selects a from a defined number of cleaning methods, and chooses repairs that are likely to affect the model and show an improvement in accuracy.

Jenga: Jenga [54] is an open source experimentation library that allows to test ML models for robustness against common data errors observed in production scenarios (e.g., missing values, swapped values, noise). It takes raw input data and randomly introduces certain data errors to then evaluate the given as an input ML model. Jenga gives an opportunity to study the impact of introduced errors on the dataset and model prediction, respectively. Jenga uses ML approach (predict - corrupt - evaluate) to validate data and to check the influence of errors.

GouDa and BART: This work addresses the commonly lacking public availability of datasets with ground truth for research and evaluation of ML pipelines, data cleaning approaches. GouDa [51] is a tool for the automated generation of datasets with possibility to create specific error types at arbitrary error range. GouDa consists of two steps: data and error generation. Data generator (part of EvoBench project [45]) takes a schema and feature value ranges as an input and outputs purely synthetic dataset based on them. Error generator introduces variety of error types at configurable error

rate at a single value, attribute, tuple and several tuples levels. Since the data is fully synthetic and generator's inputs are schema and constraints, ground truth is provided. Conversely to GouDa, BART [3], an error-generation tool for data cleaning applications, needs an existing dataset and inserts errors into the actual dataset that is assumed to be clean. It requires relational schema, users specify a set of data quality rules as denial constraints [40] (functional dependencies [5, 49], conditional functional dependencies [14], fixing rules [60]). User can specify error rates but also how detectable and repairable introduced errors are given constraints.

Comparison of existing tools and the new data generator: Most of the mentioned above systems are not supporting distributed execution [43, 41, 50, 23, 51, 54], limiting the applicability to a single machine. Another constraint in existing frameworks are hand-crafted rules/constraints or manual labeling (by users). Importantly, it is difficult to measure the accuracy of the above mentioned tools without ground truth and mostly existing frameworks use manually cleaned datasets, which is not feasible at scale. This thesis solves the dilemma of finding ground truth. Errors are generated not syntactically, but are based on the original data and error distribution. Moreover, data can be generated either locally or in a distributed setting. In comparison to GouDa, where datasets are purely synthetic, our data generator uses real-world datasets, thus error and data distributions are preserved. Additionally, in contrast to GouDa and Jenga, our system supports distributed data and error generation.

2.3 Existing Benchmarks

Currently, there is no standard and generalized way of benchmarking data validation, error detection and repair for ML, but benchmarks for selected related aspects exist.

Sorting Hat project: This project aims to benchmark the automation of the data preparation in AutoML platforms: create benchmarks and labeled datasets and use ML to automate data preparation. In the context of the Sorting Hat project [57], the ML Data Prep Zoo [55] was introduced. ML Data Prep Zoo is a public repository with labeled datasets and pre-trained ML models for data preparation tasks (e.g., schema inference, detection of anomalous categories, category deduplication, feature type inference). The central goal is to create large labeled datasets without time-consuming manual labeling.

OpenML: OpenML [4] is a public platform for sharing datasets, algorithms, and experiments. It also contains a number of benchmark suites. The latest

suite is OpenML-CC18 [4] that contains frequently used ML datasets from 2018.

DataCivilizer: DataCivilizer [6] is an end-to-end big data management system. It builds a linkage graph for the data and utilizes it to identify data relevant to user tasks. It integrates data cleaning process into the query processing and trades-off the query result quality with the cleaning cost. Automatic error detection tools are applied to detect outliers, duplicates and integrity constraints violations.

CleanML: CleanML [37] is a benchmark for joint data cleaning and machine learning. This study experimented with 14 public real-world datasets without ground truth to investigate the impact of data cleaning on ML classification tasks. ML model performance is evaluated on clean/dirty data. The study shows that data cleaning is more applicable solution comparing than specific robust models, and can be valuable in both the model development and deployment steps [37]. Moreover, some errors have less impact on ML models than others (e.g, missing values imputation positively influences ML model performance, while cleaning outliers has more likely insignificant impact).

AutoML benchmarks: There are also AutoML benchmarks that compare diverse AutoML systems. In this context, tools such as AutoWeka [30] and Auto-SKLearn [15] are evaluated. AutoWeka is a platform that focuses mainly on model selection, not data cleaning and preparation like Sorting Hat and Prep Zoo. AutoSKLearn also mainly concentrates on algorithm selection and hyperparameter tuning.

Comparison to this work: Current error detection [1, 37] shows that the percentage of errors found by techniques in real-world datasets is still well below 100%. Many errors can be spotted only by humans, but not by currently existing algorithms, techniques, and tools. Moreover, there is no universal data cleaning tool for all datasets, and thus "composite" strategies are commonly applied [1]. This paper takes a real-world dataset as an input, extracts real error patterns and dependencies (e.g., functional dependencies, value ranges) from the data, and finally, scales the dataset with configurable error rates. Moreover, a user can define own rules, constraints or even methods how to impute the error to a particular feature.

2.4 Limitations of Existing Work

Limited coverage: Most of the existing frameworks or systems take into account either a subset of particular error types [33, 43, 41, 54], or use syntactic data as a ground truth to then generate errors [51, 3].

Synthetic data: Synthetically generated errors are not representing real-world errors in the dataset. Existing frameworks support a subset of cleaning tasks and data modalities. For the real-world data the ground truth and errors are unknown. Moreover, many datasets are publicly not available with or without ground truth (out of date links or no source provided).

Manual effort: Hand-crafted rules/constraints [3, 51], manual labeling [43, 41], additional user inputs can also be seen as a restriction since user at least is not always capable to formulate them correctly/fully. Additionally, existing work can be optimized by parallelizing the workflow [43, 41, 50, 23, 51, 54].

No standardization: There is no standardized benchmark for cleaning algorithms and methods for ML pipelines. Currently available frameworks require an additional input to make correct decisions during the cleaning. Manual cleaning to get the ground truth is practically infeasible at scale because it takes a lot of effort. Moreover, manual cleaning may lead to biased results if the expert knowledge is not sufficient enough. Thus, there is a need for a benchmark that standardizes the comparison and evaluation of data cleaning for ML, and aims to solve the problem of the missing ground truth or publicly not available datasets.

3 Data Generation Framework

In this chapter, the framework and its components are described: error types (Section 3.1), data and error generation (Section 3.2), and validation of the results (Section 3.3). Unlike other frameworks discussed in Section 2, this framework uses only clean and dirty versions of real datasets to scale them up while maintaining data characteristics, without taking any additional hand-crafted input or constraints.

3.1 Types of Errors

Data errors are values that differ from the ground truth. In this context, we differentiate between data with and without errors by calling it clean and dirty. Completely clean data is ground truth (GT). In the context of this benchmark, we have a clean ground-truth dataset and its dirty version with earlier introduced errors. Leveraging this, faulty values can be found by locating the cells that differ between dirty and clean. This information is helpful to detect errors, their distribution, and frequencies of specific features being error prone. Moreover, it is useful for classification of error types.

The five error types chosen are common in the real-world datasets. They include missing values, typos, outliers, replacements, and swaps.

Missing values: The simplest type of error that is frequently seen in our experiments are missing values. Generally, there are three sub categories of missing values: Missing Completely at Random (MCAR), Missing at Random (MAR), Missing Not at Random (MNAR). MCAR is a value that is missing completely independently from other values and itself. Missing values have no dependencies or ties to any other values in the dataset. MAR occurs when the missing value is random, but is related to the part of the observed data and thus can be reconstructed, an instance of this is where in datasets with personal weight, females are less inclined to include their weight. MNAR means that the missing value of an observation depends on its values, for instance, using the weight example again, weight is not reported for obese individuals. MNAR analysis is problematic because the

distribution of the missing observations depends on both observed and unobserved values. Missing data introduce various problems. First, the absence of data reduces statistical power of the dirty dataset, moving its statistical properties further from the ground truth. Second, lost data can cause bias in the estimation of parameters. The previous example of people not reporting if they are obese could be a cause of bias. Third, the representativeness of the samples can influence the set of distinct values, frequencies, and ratios between them. Our data generator supports MCAR. Infinity, Not a Number, NA, maximum and minimum floating point numbers are classified as missing values that can be added. New missing values are generated using unique missing values of the original dirty dataset.

Typos: Typo is a typographical error that typically is introduced by misspelling. Common instances of typos are Morrocco and Lost Angeles. Introduction of typos can violate the set of distinct values, since new distinct values are be inserted. This can be detrimental for techniques such as one-hot-encoding that is highly sensitive to the number of distinct values. In the benchmark, the typo distribution and the distinct value set are used to estimate the number of new unique typos to add to the generated output. The new distinct typos are generated by modifying existing values multiple times. Following modifications are randomly performed: add existing character, add new character, remove existing character, and swap two existing characters. Modifications are controlled by measuring Levenshtein distance between the clean and modified values.

Outliers: A data point that significantly differs from a data distribution is called an outlier. An example of this could be human adult height equal to 3m, 20cm, or a negative value such as -1.5m. Since outliers largely effect certain loss functions (L2 loss) and distances, they decrease statistical power. In our benchmark we are introducing outliers while preserving dirty data statistics. New outlier values are created using the correction, in Equation 3.1. Correction is used to shift actual mean of the scaled data to the desired mean of the dirty data. Correction increases interquartile range. To introduce outliers preserving the statistics, every new outlier should be balanced by another outlier. Thus, outliers are created in pairs of values that balance the outliers. In essence the balanced outlier values utilize the concept of reflection symmetry around mean. We generate initial outliers using interquartile range, scaling by orders of magnitudes, distribution, or minimum and maximum of the clean dataset.

$$\text{CORRECTION} = \frac{\left(\text{MEAN\_DIRTY} - \text{MEAN\_GENERATED}\right) \cdot \text{NROW\_GENERATED}}{\text{NUM\_OUTLIERS}} \tag{3.1}$$

Replacements: Replacement is a flipped value that was chosen from the existing set of valid values. For example, in feature with distinct set of values

{A, B, C, D}, A can be replaced by D. Replacement does not introduce new distinct values, but changes the frequencies of the different distinct values. Replacement reduces statistical power of the dirty dataset and moves its statistical properties further from the ground truth. In this thesis, correspondence pairs of replacements and their frequencies are detected, and scaled accordingly.

Swaps: An exchange of pair of values within an observation is a swap. Mostly, they occur when data is misplaced while entering. For instance, a DoB value 01.01.2001 is mistakenly filled into Surname, and Surname value Smith - into DoB. Swaps introduce new irrelevant distinct items or outliers if two numerical values are swapped. In the benchmark, swaps are generated based on their distribution in the dirty dataset. Swaps are detected and counted, then the number of swaps is scaled. This approach allows to maintain the data distribution while generating the new dataset, without introducing violating distinct value set of the dirty data.

Error generation properties: Single source datasets are used to introduce errors. Thus, multiple source datasets and errors that occur during schema integration are not considered in the framework. The error generation includes properties such as error distribution and reproducibility. Respective errors are introduced to the scaled dataset according to the dirty data error distribution. The whole data generation is seeded, and errors origin can be tracked.

Bias of introduced errors: Bias in the data generation is unavoidable, especially under the constraints of preserving data distribution, statistics, and error fractions. Missing values are introduced at random. This leads to potential replacing of never missing values in original data. Moreover, for small scales (e.g., 2x) this random application can effect statistics, such as the mean, significantly. Typos are generated by modification of existing distinct values, thus, fully new relevant typos can not be guaranteed. Outliers are biased by the statistics that the introduction of outliers tries to fix. Additionally, similar to missing values, outliers are introduced at random. Replacements and swaps are not biased and reproduce the original dirty data via the counts of the specific swaps/replacements of 'a' and 'b'. Both these types do not introduce new values and only modify existing dirty-clean pairs seen in the original dataset.

3.2 Data Generation

For our data generation framework, we introduce two implementations to generate scaled up datasets: Local and distributed. Both implementations

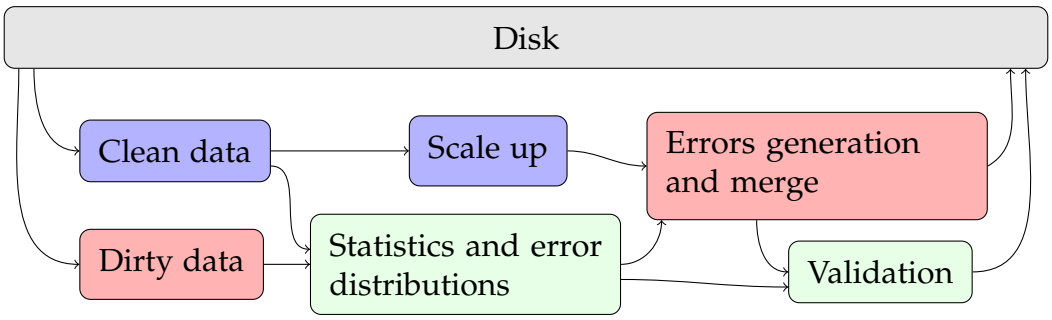


Figure 3.1: Local data generator

maintain data and error statistics. The framework contains full pipelines consisting of the following generalized steps:

- Read clean and dirty data.
- Data analysis and error classification.
- Scale up the clean data by replication.
- Generate errors and merge with the scaled clean data.
- Save generated dirty dataset.
- Generate validation report.

3.2.1 Local Data Generation

First, for scaling to in-memory sizes there is a local data generator execution that consists of the following steps that are also shown in Figure 3.1:

Read: Reading of clean and dirty datasets is done via Pandas DataFrames. Pandas is chosen for its flexibility, rich API and simplicity. An important part of the reading is detecting the schema. Pandas has a schema detection that allows reading dirty data while handling features with different data types. Python in general supports collections that may contain heterogeneous data types. These properties together allow the creation of a save schema for the dirty data. On the other hand, the automatic schema detection is not always precise, and integer cases can be detected as floats. Therefore, for the clean data, custom schema detection is applied. A sample of the data is extracted and analyzed for fine-grained data types.

Error analysis: Error analysis is performed on the clean and dirty datasets together. First, distinct values of actual data are aggregated, and a binary mask (of dirty dataset dimension) of differences between clean and dirty datasets is created. The mask allows the framework to know the ratio and exact number of errors introduced in each feature, and to track detected and undetected errors. Errors are analyzed in a particular sequence, since if a cell was classified as error of one type, it can not contain any other

error types, this is based on the simplifying assumption that there are no compound errors. First, missing values are detected using the Pandas ISNULL() method. Second, outliers are detected by interquartile range (IQR) and by distribution. Formula 3.3 uses IQR, Formula 3.2, to define lower and upper limits, while Formula 3.5 relies on standard deviation, Formula 3.4. Lower and upper limits are stored, and then reused during error generation. Third, dirty data is inspected for replacements. Replacements are cells that differ from the clean dataset, but still belong to a clean distinct values set. Fourth, to find swaps between features, the framework computes row sums utilizing the binary difference mask. For rows with more than 2 errors, faulty cells are checked for swaps using an all-pairs comparison of dirty cells with clean values. For simplicity during error generation, swaps are divided into numerical (between 2 numerical features) and mixed (between string and any other feature), since swaps between string and numerical features violate schema integrity. Finally, all remaining erroneous *string* feature values are classified as a typo.

$$\text{IQR} = \text{Q}(0.75) - \text{Q}(0.25) \tag{3.2}$$

$$\text{IQRO}(v) = v < \text{Q}(0.25) - 1.5 \cdot \text{IQR} \ \cup \ v > \text{Q}(0.75) + 1.5 \cdot \text{IQR} \tag{3.3}$$

$$\text{STD}(x) = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}} \tag{3.4}$$

$$\text{STDO(v)} = v < \overline{a} - k \cdot \text{STD} \ \cup \ v > \overline{a} + k \cdot \text{STD} \tag{3.5}$$

Analysis of statistical properties: Statistics analysis is performed by computing univariate statistics such as min, max, mean, variance, kurtosis and skeweness are computed for both clean and dirty datasets. When scaling the dataset, the framework strives towards maintaining these statistics at two steps. First, when scaling the clean dataset, the statistics of the clean dataset are maintained. Second, when introducing errors, the detected errors and their distribution are preserved. To reflect a realistic dataset, Schlosser [21] and Hass Stokes [18] distinct item estimators are used. The dirty dataset is used as a sample of the generated. It is assumed that depending on the distribution of the distinct errors in the dirty dataset, more or an equivalent number of distinct values are introduced when scaling the data.

Scaling up: Scaling up clean data while preserving statistics is challenging, especially if applied at random. For instance, in a column with all unique

Algorithm 1 Local Error Generation Algorithm

Input: Error distribution **err_dist**, scaled dataset **data**, scaling factor *scaling_factor*, schema of the dirty dataset *dirty_schema*
Output: New error distribution **new_err_dist**, scaled dataset **data**

```
1: // a) create new error distribution
2: new_err_dist ← ErrorDistribution(dirty_schema, data.row, data.col)
3: // b) create a pool to introduce mv, typos, replacements, and indices for outliers
4: pool ← Pool(#CPU)
5: for col in data.columns do
6:     kwargs ← [data[col], err_dist, new_err_dist, scaling_factor, col]
7:     tasks.append(pool.apply_async(GET_ERRORS_IN_COL, kwargs))
8: // c) execute tasks and modify new error distribution
9: for task in tasks do
10:    dist_changes, col_name, updated_col ← task.get()
11:    new_err_dist.UPDATE(dist_changes, col_name)
12:    data[col] ← updated_col
13: // d) introduce numerical swaps
14: data, new_err_dist ← ADD_SWAPS_NUM(data, err_dist, new_err_dist)
15: // e) introduce outliers based on indices created earlier
16: for col in data.columns do
17:    kwargs ← [data[col], err_dist, new_err_dist.outlier_indices, col]
18:    tasks.append(pool.apply_async(ADD_OUTLIERS, kwargs))
19: for task in tasks do
20:    dist_changes, col_name, updated_col ← task.get()
21:    new_err_dist.UPDATE(dist_changes, col_name)
22:    data[col] ← updated_col
23: // f) introduce mixed swaps
24: data, new_err_dist ← ADD_SWAPS(data, err_dist, new_err_dist)
25: return new_err_dist, data
```

values, the probability of selecting all values at least once is low when scaling factor is small. It is similar to the rolling dice with 6 unique sides. Scaling to 2x is equivalent to rolling 6 unique sides at least once with 12 trials. To avoid this inconsistency, we chose to scale up by replication. If the scaling factor is a floating point number, then the integer part is replicated, and the remaining floating part is sampled randomly from the clean data.

Error generation and introduction: Error generation and introduction is done column wise, and is presented in Algorithm 1. Since the main workload of the framework is in this phase, each column is processed in parallel, with the exception of multi column errors such as swaps. Typos, missing values and replacements are added to the scaled dataset in the first iteration as shown in Algorithm 2. For missing values and typos the distinct item

Algorithm 2 GET_ERRORS_IN_COL ALGORITHM

Input: Error distribution **err_dist**, new error distribution **new_err_dist**, column data **data_c**, scaling factor *scaling_factor*
Output: New error distribution **new_err_dist**, scaled dataset **data**

```
// a) get typos values
new_err_dist.typo_values, num_typos ← GET_TYPOS(err_dist)
// b) get missing values
new_err_dist.mv_values, num_mv ← GET_MV(err_dist)
// c) introduce replacements
new_err_dist.replacements_indices ←
            ADD_REPLACEMENTS(data_c, err_dist)
// d) count number of outliers
new_err_dist.num_outliers ← err_dist · scaling_factor
// e) get random indices
size ← new_err_dist.num_mv + new_err_dist.num_outliers + new_err_dist.num_typos
ind ← RANDOM_SAMPLE_EXCEPT(
            data_c.nrow, size, new_err_dist.replacements_indices)
// f) modify column with errors
data_c ← ADD_ERRORS(data_c, ind, new_err_dist.replacements_indices)
return data_c, new_err_dist
```

sets and frequencies are gathered from the error analysis and statistics phases. To maintain the frequencies of missing values and typos, existing missing values and typos are scaled up, and number of new distinct missing values and typos is estimated by Schlosser [21] and Hasso Stokes [18] estimation algorithms. Existing missing value and typo frequencies are scaled by the scaling factor. Additionally, a specific number of new unique values is generated, this number is estimated as described in the previous paragraph. The number of occurrences of a new unseen unique value is computed similar to n-gram smoothing techniques. The new frequency of unseen values is defined by Laplace smoothing "add-one" to avoid the zero-frequency problem. The zero-frequency problem: If an individual class label is missing, then the frequency-based probability estimate will be zero. To introduce the errors into the data, a vector with shuffled random indices is allocated and then utilized. For each unique error type a random sample of indices is extracted and then applied to the scaled dataset. Replacements are added using earlier created dictionaries of original and replaced values, and their frequencies. Cells to modify are filtered, then a sub-fraction is randomly sampled and corrupted. Since replacements do not introduce new distinct items, no additional computations are necessary. Similarly, swaps are done by filtering two columns and randomly choosing rows to

exchange. The number of rows to modify is computed from the original number of swaps and scaling factor. To choose not already modified values for replacements and swaps, indices of modified cells are stored and then RANDOM_EXCEPT method is used. This method returns random row indices that are not in the given list of indices. The last step is the addition of outliers. They are applied at the end of the sequence because, if applied correctly, they can scale the mean of the new scaled dataset to the desired dirty mean through the technique described in Section 3.1. Outliers are also introduced to columns in a parallel manner, Algorithm 1.

Write: The generated dataset is saved immediately after the error generation. It is written to the disk in comma-separated values (CSV) format. The dataset is saved before validation to ensure that we have the generated data, even if validation fails or concludes invalid generation.

Validation: The last stage of the framework is the validation of the generated dataset. It is done by comparing the statistics of the original dirty and generated datasets, the details are described in Section 3.3. The error distribution of the generated dataset is stored while generating and introducing the errors. The univariate statistics are computed after the full generation is finished. Both univariate statistics and error distribution are used for the validation. Additionally, all statistics are saved to files.

Limitations of the local execution: The Pandas DataFrames are relatively slow. Pandas is written in C, and switching between C and Python takes time, especially for mathematical computations and data retrieval. Another limitation is the in-memory computation. This means that data can not be scaled above memory of a single machine. Third, there is potential for more parallelization of different parts of the local execution, but the parts already parallelized include the critical parts.

3.2.2 Distributed Data Generation

The distributed generator is for scaling to larger sizes of data that exceeds local memory restrictions. It consists of the following steps that are also shown in Figure 3.2:

Read: Reading of clean and dirty datasets is similar to local execution. To compute statistics and detect schema, local Pandas DataFrames are used. On the other hand, the lazily evaluated Spark DataFrame is defined for further scaling up. The schema applied to the Spark DataFrame is obtained from a schema detection of the dirty data in Spark. This is important to avoid schema violations while introducing different from data types into clean feature. For instance, swaps can be introduced between numerical and

Algorithm 3 Distributed Error Generation Algorithm

Input: Error distribution **err_dist**, scaled dataset **data**, scaling factor *scaling_factor*, schema of the dirty dataset *dirty_schema*
Output: New error distribution **new_err_dist**, scaled dataset **data**

```
1: new_err_dist ← ErrorDistribution(dirty_schema, data.row, data.col)
2: // a) introduce replacements
3: pool ← Pool(#CPU)
4: for col in data.columns do
5:     new_err_dist.rep_indices, new_err_dist ← CREATE_REPLACEMENTS_WRITE_AND_JOIN(data, err_dist, new_err_dist)
6: // b) create a pool to write files with mv, typos, and outliers
7: pool ← Pool(#CPU)
8: for col in data.columns do
9:     kwargs ← [err_dist, new_err_dist, scaling_factor, col]
10:    tasks.append(pool.apply_async(GET_ERRORS_IN_COL_DIST, kwargs))
11: // c) execute tasks and modify new error distribution
12: for task in tasks do
13:    dist_changes, col_name ← task.get()
14:    new_err_dist.UPDATE(dist_changes, col_name)
15: // d) read files with errors and join into data
16: READ_ERRORS_AND_JOIN(data, new_err_dist)
17: // e) introduce swaps
18: data, new_err_dist ← CREATE_SWAPS_WRITE_AND_JOIN(data, err_dist, new_err_dist)
19: return new_err_dist, data
```

string columns.

Error analysis and statistics: This step is equivalent to the local execution.

Distributed scaling up: Scaling of the dataset in distributed mode is achieved via three different phases. Since Spark DataFrame is not indexed per default, to scale it up, one of the possibilities is to append DataFrame to itself, similarly to the local. But because of Spark usability and API, it is possible to avoid concatenation by using a combination of Spark user-defined function (UDF) and EXPLODE. First, we define a column with sequential indices. Then, this column is used by a UDF to map each existing row index to a list of new sequential row indices. Finally, Spark EXPLODE utilizes these lists to "explode" (replicate) rows with a new index.

Error generation: Erroneous values are generated locally using error characteristics collected from the original dirty dataset. The values are combined with row indices to create new error locations, Algorithm 3. Indices are chosen at random for missing values, typos and outliers, Algorithm 4, or based

Algorithm 4 GET_ERRORS_IN_COL_DIST ALGORITHM

Input: Error distribution **err_dist**, new error distribution **new_err_dist**, scaling factor *scaling_factor*
Output: New error distribution **new_err_dist**, column name *col*

```
1: // a) get typos values
2: new_err_dist.typo_values, num_typos ← GET_TYPOS(err_dist)
3: // b) get missing values
4: new_err_dist.mv_values, num_mv ← GET_MV(err_dist)
5: // d) count number of outliers
6: new_err_dist.outliers_values, num_outliers ← GET_OUTLIERS(err_dist)
7: // e) get random indices
8: size ← new_err_dist.num_mv + new_err_dist.num_outliers +
   new_err_dist.num_typos
9: ind ← RANDOM_SAMPLE_EXCEPT(
10:           data_c.nrow, size, new_err_dist.replacements_indices)
11: // e) write to file tuples of indices and errors
12: WRITE_IND_ERR_TO_FILE(ind, new_err_dist)
13: return new_err_dist, col
```

on the filtering of specific values for replacements and swaps. A challenging part here is to guarantee that indices are non-intersecting. To achieve this property, replacements are performed first, and afterwards random indices for other errors are chosen, avoiding already used indices. This process is parallelized to write buffered strings of error tuples to HDFS, materializing and persisting the errors introduced in the scaled dataset. These files can also be seen as meta data. Introducing the errors into the actual distributed scaled up dataset is done by reading errors into Spark DataFrame from HDFS, and joining the errors with the replicated clean data on synthetic index.

Write to HFDS: Writing to HDFS was initially done using csv format, similar to the local data generation. However, utilizing Spark compression and parquet format reduced runtime significantly.

Validation: Results validation is the same as local, except Spark functions are used to aggregate statistics for comparison of the generated and original dirty datasets.

3.3 Validation of Results

After the generation, the new dataset is validated to analyze and check if statistics are preserved. The data and error characteristics of the generated

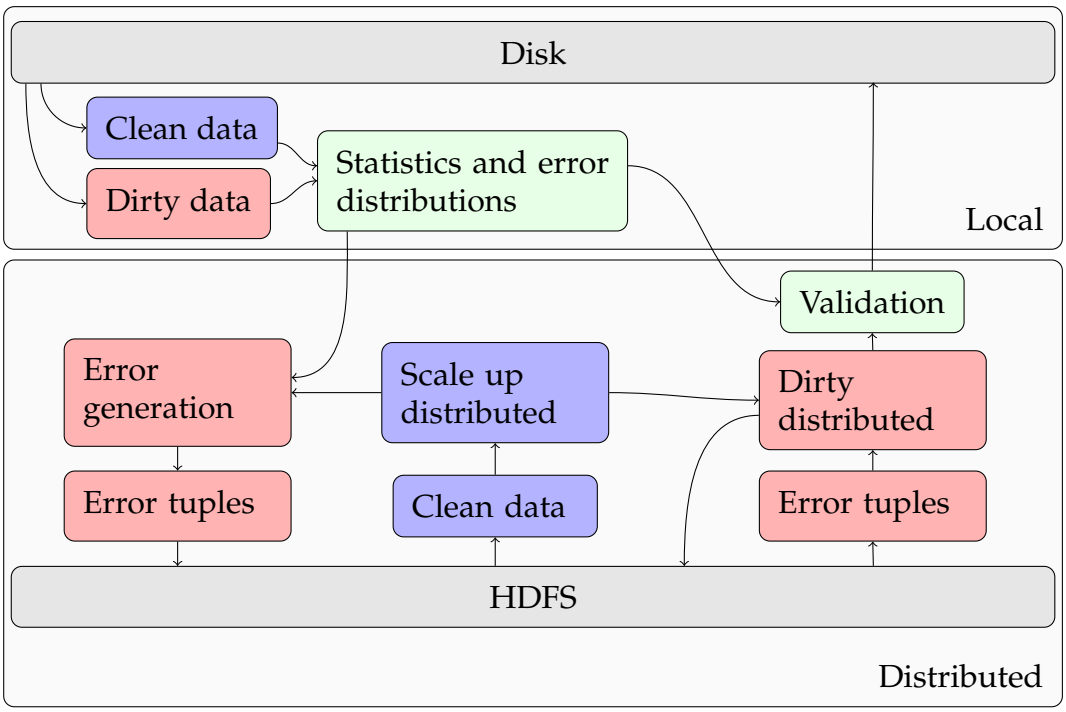


Figure 3.2: Distributed data generator

dataset are compared against characteristics of the original dirty dataset. The following soft constraints are used to validate the generated dataset:

- The mean difference of the generated and the original dirty datasets should be within 5%.
- The variance difference of the generated and the original dirty datasets should be within 5%.
- The upper and lower quantiles of the generated dataset should be within
  $[Q_{dirty}(0.25) - 1.5 \cdot IQR, Q_{dirty}(0.75) + 1.5 \cdot IQR]$.
- Min and max are compared, and should be equivalent.
- The number of distinct values in generated dataset and estimated number of distinct values by Equation 3.6 should be within 5%.
- The number of generated missing values, outliers, typos, replacements, and swaps differentiates by maximum of 10% from the original dirty numbers multiplied with the scaling factor.
- The number of unique values for each error is compared to the estimated, similar to the number of distinct items above.

$$d_{generated} = d_{clean} + d_{mv} + d_{typos} + d_{outliers} + d_{swaps} \tag{3.6}$$

If any of constraints are violated, the user is informed about these specific

violations. Additionally, characteristics of the generated dataset are stored into separate logging files. They contain information such as statistics, error distribution, and binary masks. All this information can be useful since recomputing it can be time consuming, especially at large scales.

Local validation: To compute statistical properties of the freshly generated data, Pandas API is used.

Distributed validation: In distributed data generation, Spark SQL aggregation functions are used to accumulate statistics. Interestingly, the performance of aggregating statistics impacts execution time. Thus, Spark approximate algorithms were tried out, such as APPROX_COUNT_DISTINCT. But for small scaling factors these were inaccurate even with 10% precision, estimating the number of distinct items less in original dirty dataset. Consequently, estimations and in-expectation statistics are not sufficient for this framework. Error distribution of the generated dataset is computed while error generation same as in the local execution.

4 Experiments

This section contains the experimental evaluation of the framework. First, the experimental setup is described in Section 4.1. Second, in Section 4.2 different datasets and their metrics are described. Third, runtime of both local and distributed are analyzed and compared, in Section 4.3.

4.1 Experimental Setup

All experiments were conducted on a Hadoop cluster using 3 worker nodes. Each node has a single AMD EPYC 7302 CPUs @ 3.0-3.3 GHz (16 physical/32 virtual cores, and 128 GB DDR4 RAM (peak performance is 768 GFLOP/s, 183.2 GB/s). The software stack comprises Ubuntu 20.04.1, Python 3.8.10, Apache Hadoop 3.3.1, Apache Spark 3.2, PySpark 3.2, Pandas 1.4.3, and NumPy 1.23.1.

Apache Spark with 3 executor nodes is using OpenJDK 11.0.13 with both drivers and local set to 100 GB memory. Further settings are: executor cores=32, deploy mode is client, heartbeat interval=50 sec, network timeout=100000 sec. The default heartbeat and timeout lead to termination before the long running job is done, therefore the settings are manually set to increased values.

4.2 Data Characteristics

This section describes datasets used for the experiments. Additionally, statistics and error characteristics of datasets are discussed.

4.2.1 Datasets

To evaluate the performance of the benchmark, six public datasets were used from Mahdavi et al. [44] and Mahdavi and Abedjan [42]. They are available on GitHub[1]. All datasets have a clean and dirty version. In Table 4.1 datasets

[1] `https://github.com/BigDaMa/raha/tree/master/datasets`

Table 4.1: Datasets

	#rows	#cols	Clean	Dirty	#Cat	#Numeric
beers	2,410	11	233 KB	255 KB	8	3
flights	2,376	7	173 KB	155 KB	7	0
hospital	1,000	20	303 KB	303 KB	20	0
movies	7,390	17	4.4 MB	4.6 MB	14	3
rayyan	1,000	11	273 KB	273 KB	9	2
tax	200,000	15	14.6 MB	14.6 MB	10	5

and their main characteristics are presented.

Beers is a real-world dataset. It has been collected by web scraping, and was cleaned manually by owners. It is used in Mahdavi et al. [44] and Mahdavi and Abedjan [42], and the source can be be tracked to Hould [26, 25]. The dataset contains information about different beer sorts, respective bottles and their producers.

Flights is a real-world dataset that originally was collected by Li et al. [38]. It is used in many projects [23, 43, 38]. This dataset contains information on arrival and departure of flights.

Hospital data is taken from US Department of Health & Human Services [2]. It is frequently used [51, 8, 10, 23, 44, 42, 50]. The dataset contains information about providers, their addresses, contacts and measurements they do. In the experiments, a fraction of the whole dataset was used. The fraction is published in GitHub [3]. Both *flights* and *hospital*, were obtained along with ground truth from Heidari et al. [23].

Movies is a dataset that is available in the Magellan repository [11, 29]. It was collected by web scraping Rotten Tomatoes [4] and IMBD [5]. The dataset describes movies, their casts and ratings. Interestingly, one of the columns DESCRIPTION is free text.

Rayyan is also a real-world dataset. It was cleaned by the owners [48] themselves, and is used in Mahdavi et al. [44] and Mahdavi and Abedjan [42]. The dataset summarizes articles, as well as their characteristics such as publisher, volume, and language.

Tax is a large synthetically created dataset from the BART [3] repository. It summarizes information about tax payers and their personal information

[2] http://www.hospitalcompare.hhs.gov

[3] https://github.com/BigDaMa/raha/tree/master/datasets

[4] https://www.rottentomatoes.com/

[5] https://www.imdb.com/

that is important for the tax estimation. Interestingly, *tax* is the largest dataset, its clean and dirty versions are 14.6 MB each.

4.2.2 Data and Error Characteristics

Main goals during data generation is to maintain error characteristics and statistics of the original dirty dataset in the generated dirty dataset. In this Section, the characteristics of the original dirty input and the clean data are discussed and explained.

Preserved characteristics: The main objective is preservation of distinct values, min, max, mean, and distributions of typos, missing values, outliers, replacements and swaps. These aggregate statistics are important because they contain information about every value. For instance, the mean influences the detection of outliers or missing values. Similarly, maintenance of frequencies of distinct values allows one to preserve error and data distribution while scaling.

Error distribution: In Table 4.2, the error distribution of the different datasets is presented. Graphical representations of the errors include: The percentages of different error types in each dataset, shown in Figure 4.1, and frequencies, shown in Figure 4.2. In the *beers* dataset only two types of errors are present: Typos and missing values. Interestingly, one column OUNCES was classified as fully missing because it was transferred from integer to string, e.g. 12 was turned into 12 OZ or 12 OUNCE, and there are many missing values in the IBU column. The *flights* dataset contains the biggest count of typos over all datasets. Most of them are in SCHED_DEP_TIME column, e.g. 6:55 A.M. is replaced by 12/02/2011 6:55 A.M.. The *hospital* data has the smallest number of errors. Typos are in the ADDRESS_1 column, while missing values are present in ADDRESS_1, ADDRESS_2, ADDRESS_3. The *movies* dataset contains the most text columns such as NAME, DESCRIPTION, DIRECTOR, CREATOR, RELEASE_DATE, all of them contain typos. Additionally, in the column RATING_COUNT 5,737 values are missing. The *tax* dataset contains all five error types. Together with typos, missing values and replacements, there are outliers, e.g. SALARY of 100,000,000, and swaps between F_NAME & L_NAME, SALARY & RATE, and L_NAME & CITY. There are 8 swapped values (4 pairs) in total. In general, there are not many outliers detected. First, the chosen datasets contain mostly text features what means that there are no outliers in the columns. Second, for outlier detection only basic techniques such as IQR or by standard deviation were used. This means that contextual and collective outliers are not detected, but they are treated as different error type, because all differences between clean and dirty datasets are classified as faulty cells. There are also not many swaps in the given data. Although,

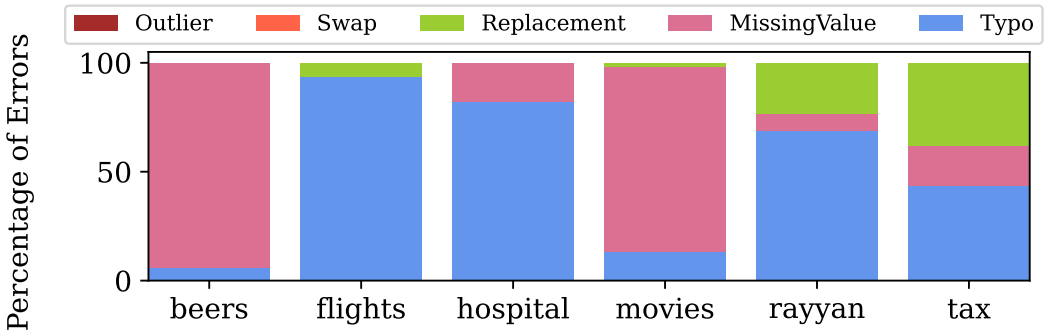


Figure 4.1: Percentage of errors

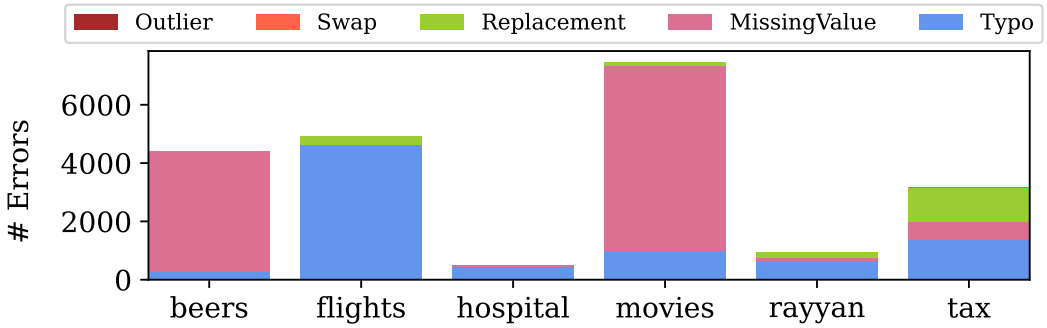


Figure 4.2: Frequency of errors

Table 4.2: Dirty Dataset Error Characteristics

	Outliers	**Typos**	**MV**	**Replacements**	**Swaps**
beers	0	254	4,170	0	0
flights	0	4,606	0	314	0
hospital	0	417	92	0	0
movies	0	982	6,346	141	0
rayyan	0	649	75	224	0
tax	0	1,367	588	1,200	4

this is a common mistake in real-world datasets.

Distinct values: Distinct values are also interesting in terms of data distribution, since errors impact the distinct items set. Figure 4.3 shows how the distinct value set in columns changes after error introduction. The x-axis represents each feature of the dataset, while the y-axis is log-scaled and shows the number of distinct values. The blue color represents clean data while the red color stands for dirty. It is noticeable that in the *beers*, as stated earlier, one column is treated as completely missing. On the other hand, in

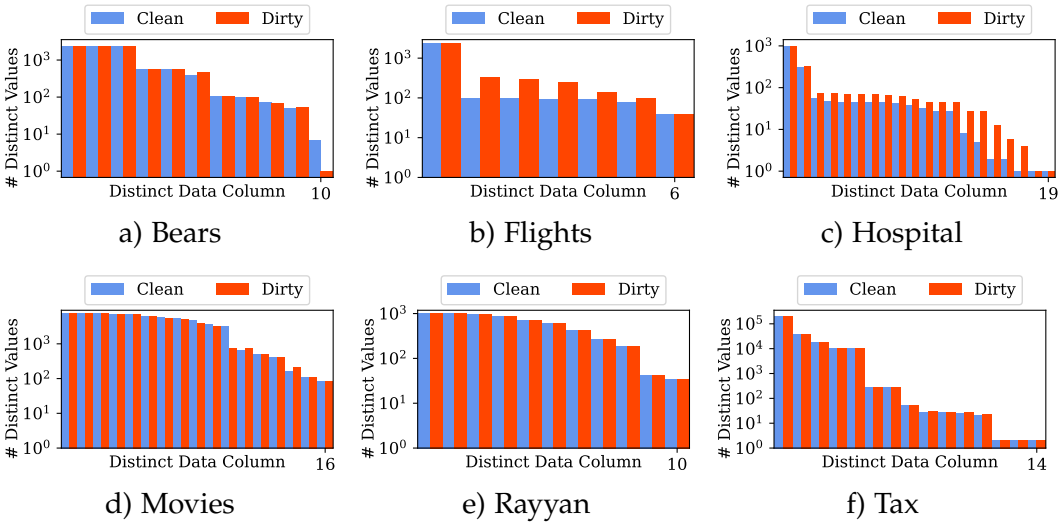


Figure 4.3: Distinct values distribution of clean and dirty datasets

the *hospitals* and *flights* the number of distinct items increases. This happens because of many typos in both datasets.

Table 4.3: Local error distribution in beers

Scale	Outliers	Typos	MV	Replacements	Swaps
1	0	254	4,170	0	0
2	0	508	8,340	0	0
4	0	101	16,680	0	0
8	0	203	33,360	0	0
16	0	406	66,720	0	0
32	0	812	133,440	0	0
64	0	162	266,880	0	0
128	0	325	533,760	0	0
256	0	650	1,067,520	0	0

Scaled error distribution: Table 4.8 shows the scaled error distribution for the *tax* dataset. In the Table row 1 is the error distribution of original dirty data. It is guaranteed that errors scale linearly within 10% of original number of errors. It can be observed that there is no necessity in 10% bound for local execution since number of errors is perfectly matching the expected. Other datasets are described in Table 4.3 - *beers*, Table 4.5 - *hospital*, Table 4.4 - *flights*, Table 4.3 - *movies*, and Table 4.7 - *rayyan*.

Distinct values: Scaling distinct values is more challenging. Table 4.13 shows how the number of distinct values of each error type changes for different

Table 4.4: Local error distribution in flights

Scale	Outliers	Typos	MV	Rep	Swaps
1	0	4,606	0	314	0
2	0	9,212	0	628	0
4	0	18,424	0	1,256	0
8	0	36,848	0	2,512	0
16	0	73,696	0	5,024	0
32	0	147,392	0	10,048	0
64	0	294,784	0	20,096	0
128	0	589,568	0	40,192	0
256	0	1,179,136	0	80,384	0

Table 4.5: Local error distribution in hospital

Scale	Outliers	Typos	MV	Rep	Swaps
1	0	417	92	0	0
2	0	834	184	0	0
4	0	1,668	368	0	0
8	0	3,336	736	0	0
16	0	6,672	1,472	0	0
32	0	13,344	2,944	0	0
64	0	26,688	5,888	0	0
128	0	53,376	11,776	0	0
256	0	106,752	23,552	0	0

Table 4.6: Local error distribution in movies

Scale	Outliers	Typos	MV	Rep	Swaps
1	0	982	6,346	142	0
2	0	1,964	12,692	284	0
4	0	3,928	25,384	568	0
8	0	7,856	50,768	1,136	0
16	0	15,712	101,536	2,272	0
32	0	31,424	203,072	4,544	0
64	0	62,848	406,144	9,088	0
128	0	125,696	812,288	18,176	0
256	0	251,392	1,624,576	36,352	0

scaling factors. The number of distinct typos, distinct missing values and distinct items are represented by estimated (*Est*) and actual (*Act*) number of

Table 4.7: Local error distribution in rayyan

Scale	Outliers	Typos	MV	Rep	Swaps
1	0	649	75	224	0
2	0	1,298	150	448	0
4	0	2,596	300	896	0
8	0	5,192	600	1,792	0
16	0	10,384	1,200	3,584	0
32	0	20,768	2,400	7,168	0
64	0	41,536	4,800	14,336	0
128	0	83,072	9,600	28,672	0
256	0	166,144	19,200	57,344	0

Table 4.8: Local error distribution in tax

Scale	Outliers	Typos	MV	Rep	Swaps
1	2	1,367	588	1,200	4
2	4	2,734	1,176	2,400	8
4	8	5,468	2,352	4,800	16
8	16	10,936	4,704	9,600	32
16	32	21,842	9,408	19,200	64
32	64	43,744	18,816	38,400	128
64	128	87,488	37,632	76,800	256
128	256	174,976	75,264	153,600	512
256	512	349,952	150,528	307,200	1,024

distinct values respectively. It is noticeable that typos are scaled according to the estimation. Missing values differ from the estimation because the set of new distinct missing values, that can be added randomly, is limited. For example, infinity and Not a Number can be new distinct missing values that are added to the existing distinct set of missing values. It is not possible to define new distinct missing values since values that can be classified as missing values are fix. There is no estimation of distinct outlier values since the original outlier distribution is fully reused, and newly generated values are used instead of original. Column *Distinct* represents number of estimated and observed number of distinct values in the whole dataset. Since some errors are introduced at random, it is expected that estimation slightly differs from the actual value.

Statistics of generated dataset: After generation, statistics are expected to be within 5% of original statistics. Figure 4.4 shows that mean, variance, min, and max are preserved. In all figures, x-axis represents the scaling factor

Table 4.9: Local errors distincts in beer

	Typos		MV		Distinct	
Scale	**Est**	**Act**	**Est**	**Act**	**Est**	**Act**
1	97	97	3	6	9,038	9,038
2	167	167	3	6	9,128	9,116
4	290	290	3	6	9,251	9,241
8	526	526	3	6	9,487	9,474
16	994	994	3	6	9,955	9,938
32	1,926	1,923	3	6	10,887	10,866
64	3,789	3,786	3	6	12,750	12,699
128	7,514	7,405	3	6	16,475	16,290
256	14,964	14,575	3	6	23,925	23,420

Table 4.10: Local errors distincts in flights

	Typos		MV		Distinct	
Scale	**Est**	**Act**	**Est**	**Act**	**Est**	**Act**
1	651	651	0	0	3,512	3,512
2	808	808	0	0	3,682	3,682
4	1,012	1,012	0	0	3,886	3,883
8	1,334	1,334	0	0	4,208	4,204
16	1,940	1,937	0	0	4,814	4,796
32	3,136	3,126	0	0	6,010	5,974
64	5,521	5,473	0	0	8,395	8,309
128	10,151	10,012	0	0	13,025	12,816
256	17,301	16,968	0	0	20,175	19,728

and y-axis is the percentage difference of the original dirty and generated mean, variance, min, and max respectively. Figure 4.4 contains only plots of the statistics with percentage difference greater than 0 because y-axis is log-scaled. The mean percentage difference is less than 1% for all datasets except *movies*. Since the *movies* dataset contains one fully missing column, as mentioned before, and the statistical properties are not preserved. The *tax* dataset variance also differs from the original variance because of the introduced outliers. Outliers are values that significantly differ from the clean dataset values. The min and max plots are shown in Figure 4.4c and Figure 4.4d respectively. Since outliers were added to the *tax*, min and max are different from original and are not within 5%. In some cases, original statistics are not maintained because of amount of errors and their nature, e.g., fully missing column. All above mentioned data characteristics are

Table 4.11: Local errors distincts in hospital

Scale	Typos		MV		Distinct	
	Est	Act	Est	Act	Est	Act
1	311	311	3	3	2,094	2,094
2	568	568	3	6	2,357	2,355
4	1,060	1,058	3	6	2,849	2,841
8	2,039	2,037	3	6	3,828	3,809
16	3,992	3,977	3	6	5,781	5,735
32	7,899	7,839	3	6	9,688	9,571
64	15,711	15,493	3	6	17,200	17,181
128	31,336	30,223	3	6	33,125	31,801
256	62,584	59,268	3	6	64,373	60,722

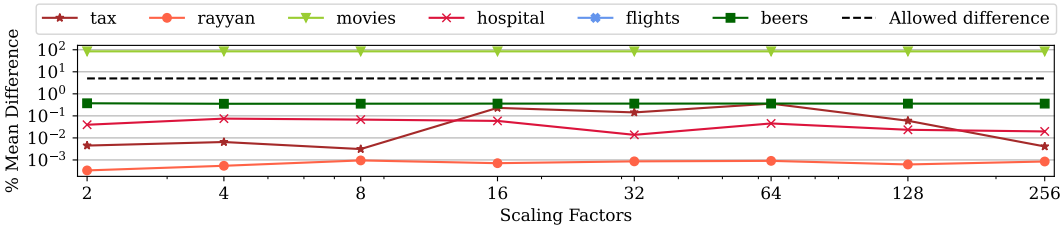
Table 4.12: Local errors distincts in movies

Scale	Typos		MV		Distinct	
	Est	Act	Est	Act	Est	Act
1	868	868	3	3	62,444	62,444
2	1,667	1,667	3	6	69,010	68,659
4	3,227	3,226	3	6	70,570	68,547
8	6,321	6,317	3	6	73,664	72,970
16	12,498	12,494	3	6	79,841	79,702
32	24,846	24,833	3	6	92,189	92,066
64	49,539	49,486	3	6	116,882	116,613
128	98,924	98,746	3	6	166,267	165,632
256	197,695	197,100	3	6	265,038	263,634

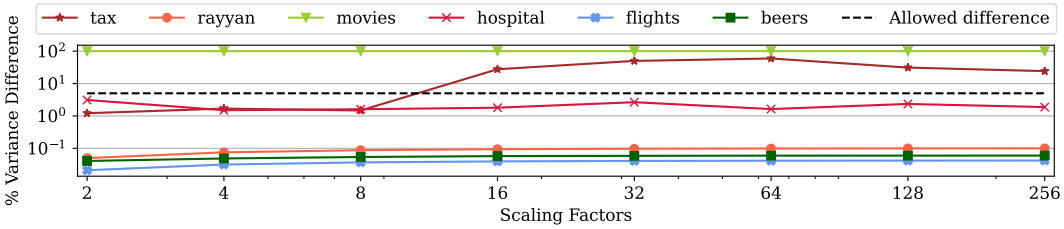
Table 4.13: Local errors distincts in tax

Scale	Typos		MV		Outliers	Distinct	
	Est	Act	Est	Act	Act	Est	Act
1	81	81	6	6	2	275,201	275,201
2	83	83	9	11	4	275,260	275,257
4	84	84	15	16	4	275,271	275,261
8	86	86	27	22	5	275,293	275,272
16	89	89	51	28	5	275,336	275,286
32	96	96	89	29	8	275,413	275,296
64	107	107	138	23	12	275,537	275,306
128	129	129	264	22	32	275,813	275,332

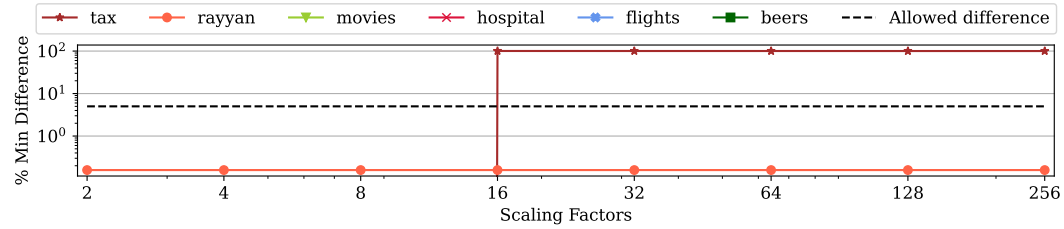
preserved (if possible) in both local and distributed data generators.



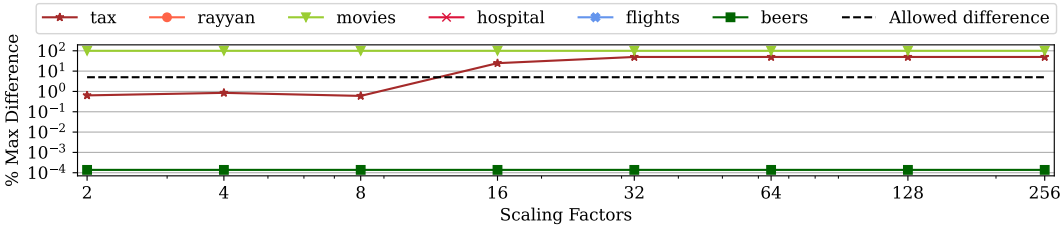
a) Mean difference in percent



b) Var difference in percent



c) Min difference in percent



d) Max difference in percent

Figure 4.4: Statistics difference in percent scaling clean and dirty datasets

Table 4.14: Local runtimes [s] with different scales

Scale	beers	flights	hospital	movies	rayyan	tax
2	2.0	2.7	2.0	5.0	1.9	25.0
4	2.2	3.1	2.1	5.4	2.1	42.8
8	1.8	3.4	2.3	6.6	2.3	100.6
16	3.0	5.1	2.6	9.4	3.3	325.6
32	3.8	9.1	3.4	16.5	5.6	1,228.1
64	5.6	26.2	4.9	43.1	13.3	5,192.6
128	8.8	98.1	7.5	151.5	44.9	21,380.5
256	17.3	390.7	13.4	581.0	176.0	– – –

4.3 Runtime Experiments

This section shows and analyzes the runtime experiments. It is separated into local (Section 4.3.1) and distributed (Section 4.3.2) execution.

4.3.1 Local Experiments

Table 4.14 shows the runtimes of each dataset at different scales. The scaling factors were increased until the distributed started outperforming the local execution. We observe that at small scales execution time is near constant, this can be seen for all datasets up to scaling factor 8 except *tax*, that is the largest dataset and contains all types of errors. Once trivial sizes are exceeded, doubling data size leads to more than doubled execution time. This indicates super linear scaling of the data generation and validation. It is assumed that the performance of the framework should be linear, but, according to the measured times, this is not the case for the error generation part of the framework. None of the operations used indicate super linear execution time. Only the swaps generation creates all-pairs of two filtered features, but swaps are present only in the *tax*. A hypothesis could be that random memory access leads to frequent cache misses and cache evictions.

4.3.2 Distributed Experiments

Table 4.15 shows the runtimes of different scaling factors in the distributed version of the data generator. The scaling factors were increased until the execution time exceeded 12 hours per dataset. It can be observed that some datasets scale better than others, but that is mainly because sizes are not as big as *tax* yet. The complexity of Spark job is dominated by joins and

Table 4.15: Distributed runtimes [s] with different scales

Scale	beers	flights	hospital	movies	rayyan	tax
8	85.5	96.8	86.4	118.3	84.7	582.5
64	92.7	102.9	93.5	262.4	89.9	2,978.0
256	177.7	164.0	147.0	767.0	116.6	11,369.3
1,024	536.4	364.2	378.7	1,799.1	276.4	48,499.5
2,048	923.7	– – –	661.3	3,785.6	454.3	– – –
8,192	2,769.4	– – –	2,061.6	14,512.8	1,591.7	– – –

repartitioning. The number of joins is based on error types introduced and amount of faulty columns in total. The joins are performed on sequential unique indices. The *tax* contains all 5 types of errors what leads to more complicated Spark jobs and more joins. Additionally, the runtime of tasks in central Spark jobs is highly skewed (seconds vs hours) leading to poor utilization of parallel resources. This leads to the performance that is worse by many orders of magnitude than expected. Also, the Spark Catalyst Optimizer tries to optimize the large sequences of instructions, this leads to long startup, compile and optimization times. For instance, in scaling the *beers* dataset to 2x, the Spark context start up takes 25 seconds, data is scaled up (lazy evaluation) in 11 seconds, errors are saved to HDFS in 5 seconds, and new error distribution/statistics is computed in 11 seconds, and the result dataset is saved to HDFS in 25 seconds, plus various overheads of 7 seconds. According to the results, the proposed distributed data generator scales linearly.

Random error generation: A faster option would be to randomly generate errors based on crafted probabilities (according to the error distribution) while replicating the individual tuples, avoiding the need to large joins altogether. Although, in this case, choice of error type and preserving the error distribution is not guaranteed as in the proposed solutions.

Generation at large scales: The largest scaling factor run is 65,536x on *rayyan* dataset. The resulting datasize is 17.9 GB, the runtime is 17,711 seconds (equivalent to approximately 5 hours).

5 Conclusions

The framework proposed in this work initiates the large-scale generation of dirty data, in which original real-world observations are used to extract error patterns and create new observations. The new data generation framework scales existing datasets while preserving error distribution and statistics. The experiments show linear performance in distributed execution, and super linear in local execution. The biggest data size generated is 18 GB that corresponds to a scaling factor of 65,536x. Six different datasets with ground truth were used for experiments: Five real-world and one synthetic. Different scaling factors from 2 to 8,192 were tested on all datasets. Thus, it is demonstrated that it is possible to generate large datasets while preserving statistics of a small sample of dirty data.

The lessons learned include: Researching the context of data cleaning, studying existing state-of-the-art frameworks and algorithms to define the components of the data generator, implementation of novel techniques for error detection and classification from clean and dirty datasets, and execution and optimization of distributed workloads. This work lead to learning about Apache Spark intrinsic behaviour and Pandas DataFrame specialization for handling clean and dirty data.

It is challenging to generate datasets while maintaining statistical properties of the original dirty dataset. From the implementation and the experimental results we can draw a conclusion that it is feasible to generate large-scale datasets with the distributed data generator, but it can be time-consuming because of skewed runtime of tasks in central Spark jobs. Our data generator can be applied for benchmarking cleaning for Machine Learning tools. Interesting directions for future work include (1) understanding of existing skew of Spark tasks and optimization of the distributed execution, (2) optimization of the local execution, (3) more advanced and fine-grained error detection and classification algorithms, (4) support of more error types, (5) integration into data cleaning benchmarks, and (6) different techniques for scaling up such as random choice of tuples from the clean dataset.

Bibliography

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. "Detecting Data Errors: Where Are We and What Needs to Be Done?" In: PVLDB 9.12 (2016), pp. 993–1004. DOI: 10.14778/2994509.2994518 (cit. on p. 7).

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding." In: vol. 30. Curran Associates, Inc., 2017 (cit. on p. 1).

[3] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. "Messing up with BART: Error Generation for Evaluating Data-Cleaning Algorithms." In: PVLDB 9 (Oct. 2015), pp. 36–47. DOI: 10.14778/2850578.2850579 (cit. on pp. 1, 6, 8, 22).

[4] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Pieter Gijsbers, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. *OpenML Benchmarking Suites*. 2017. DOI: 10.48550/ARXIV.1708.03731 (cit. on pp. 6, 7).

[5] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. "Conditional Functional Dependencies for Data Cleaning." In: 2007, pp. 746–755. DOI: 10.1109/ICDE.2007.367920 (cit. on p. 6).

[6] Raul Castro Fernandez, Dong Deng, Essam Mansour, Abdulhakim A. Qahtan, Wenbo Tao, Ziawasch Abedjan, Ahmed Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. "A Demo of the Data Civilizer System." In: SIGMOD. 2017, pp. 1639–1642. DOI: 10.1145/3035918.3058740 (cit. on p. 7).

[7] Jan Chomicki and Jerzy Marcinkowski. *Minimal-Change Integrity Maintenance Using Tuple Deletions*. 2002. DOI: 10.48550/ARXIV.CS/0212004 (cit. on p. 4).

[8] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. "Holistic data cleaning: Putting violations into context." In: ICDE. 2013, pp. 458–469. DOI: 10.1109/ICDE.2013.6544847 (cit. on p. 22).

[9] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. "Vizdom: Interactive Analytics through Pen and Touch." In: PVLDB 8.12 (2015), pp. 2024–2027. DOI: 10.14778/2824032.2824127 (cit. on p. 2).

[10] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. "NADEEF: A Commodity Data Cleaning System." In: 2013, pp. 541–552. DOI: 10.1145/2463676.2465327 (cit. on p. 22).

[11] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, Pradap Konda, Yash Govind, and Derek Paulsen. *The Magellan Data Repository*. https://sites.google.com/site/anhaidgroup/useful-stuff/the-magellan-data-repository. 2016 (cit. on p. 22).

[12] Cleaning Big Data. *Most Time-Consuming, Least Enjoyable Data Science Task*. 2016. URL: https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/?sh=690a63a46f63 (cit. on p. 1).

[13] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. "Duplicate Record Detection: A Survey." In: *IEEE* 19.1 (2007), pp. 1–16. DOI: 10.1109/TKDE.2007.250581 (cit. on p. 1).

[14] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. "Conditional Functional Dependencies for Capturing Data Inconsistencies." In: ACM Trans. Database Syst. 33.2 (2008). ISSN: 0362-5915. DOI: 10.1145/1366102.1366103. URL: https://doi.org/10.1145/1366102.1366103 (cit. on p. 6).

[15] Matthias Feurer, Aaron Klein, Jost Eggensperger Katharina Springenberg, Manuel Blum, and Frank Hutter. "Efficient and Robust Automated Machine Learning." In: 2015, pp. 2962–2970 (cit. on p. 7).

[16] Benoit Frenay and Michel Verleysen. "Classification in the Presence of Label Noise: A Survey." In: IEEE 25.5 (2014), pp. 845–869. DOI: 10.1109/TNNLS.2013.2292894 (cit. on p. 1).

[17] Jochen Görtler, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, Donghao Ren, Rahul Nair, Marc Kirchner, and Kayur Patel. "Neo: Generalizing Confusion Matrix Visualization to Hierarchical and Multi-Output Labels." In: 2022 (cit. on p. 2).

[18] Peter J. Haas and Lynne Stokes. "Estimating the Number of Classes in a Finite Population." In: vol. 93. Journal of the American Statistical Association 444. Taylor & Francis, 1998, pp. 1475–1487. DOI: 10.1080/01621459.1998.10473807 (cit. on pp. 13, 15).

[19] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. "Goods: Organizing Google's Datasets." In: SIGMOD. ACM, 2016, pp. 795–806. DOI: 10.1145/2882903.2903730. URL: https://doi.org/10.1145/2882903.2903730 (cit. on p. 1).

[20] Joachim Hammer, Michael Stonebraker, and Oguzhan Topsakal. "THALIA: Test Harness for the Assessment of Legacy Information Integration Approaches." In: ICDE. IEEE Computer Society, 2005, pp. 485–486. DOI: 10.1109/ICDE.2005.140 (cit. on p. 1).

[21] Peter J. Hass, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. "Sampling-Based Estimation of the Number of Distinct Values of an Attribute." In: PVLDB. 1995, pp. 311–322. ISBN: 1558603794 (cit. on pp. 13, 15).

[22] Yeye He, Kris Ganjam, Kukjin Lee, Yue Wang, Vivek Narasayya, Surajit Chaudhuri, Xu Chu, and Yudian Zheng. "Transform-Data-by-Example (TDE): Extensible Data Transformation in Excel." In: SIGMOD. Houston, TX, USA, 2018, pp. 1785–1788. DOI: 10.1145/3183713.3193539 (cit. on p. 1).

[23] Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. "HoloDetect: Few-Shot Learning for Error Detection." In: SIGMOD. 2019, pp. 829–846. DOI: 10.1145/3299869.3319888 (cit. on pp. 1, 5, 6, 8, 22).

[24] Joseph M. Hellerstein. *Quantitative Data Cleaning for Large Databases*. 2008 (cit. on p. 1).

[25] Jean-Nicholas Hould. *Craft Beers Dataset*. URL: https://www.kaggle.com/datasets/nickhould/craft-cans (cit. on p. 22).

[26] Jean-Nicholas Hould. *Scraping for Craft Beers*. URL: http://www.jeannicholashould.com/python-web-scraping-tutorial-for-craft-beers.html (cit. on p. 22).

[27] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. "Neural Architecture Search with Bayesian Optimisation and Optimal Transport." In: *CoRR* (2018) (cit. on p. 1).

[28] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. "Model Assertions for Monitoring and Improving ML Models." In: MLSys. 2020 (cit. on p. 1).

[29] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. "Magellan: Toward Building Entity Matching Management Systems." In: PVLDB (2016), pp. 1197–1208. DOI: 10.14778/2994509.2994535 (cit. on p. 22).

[30] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA." In: *Journal of Machine Learning Research* 18.25 (2017), pp. 1–5 (cit. on pp. 1, 7).

[31] Sanjay Krishnan and Eugene Wu. "AlphaClean: Automatic Generation of Data Cleaning Pipelines." In: CoRR abs/1904.11827 (2019) (cit. on p. 1).

[32] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. "ActiveClean: Interactive Data Cleaning for Statistical Modeling." In: PVLDB 9.12 (Aug. 2016), pp. 948–959. DOI: 10.14778/2994509.2994514 (cit. on pp. 1, 5).

[33] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. "BoostClean: Automated Error Detection and Repair for Machine Learning." In: *CoRR* abs/1711.01299 (2017) (cit. on pp. 1, 5, 8).

[34] Ga Young Lee, Lubna Alzamil, Bakhtiyar Doskenov, and Arash Termehchy. *A Survey on Data Cleaning Methods for Improved Machine Learning Model Performance*. 2021 (cit. on p. 4).

[35] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. "Pretzel: Opening the Black Box of Machine Learning Prediction Serving Systems." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI. 2018, pp. 611–626. ISBN: 9781931971478 (cit. on p. 2).

[36] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization." In: *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52 (cit. on p. 1).

[37] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. "CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks." In: IEEE, 2021, pp. 13–24. DOI: 10.1109/ICDE51399.2021.00009 (cit. on pp. 1, 7).

[38] Xian Li, Xin Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. "Truth Finding on the Deep Web: Is the Problem Solved?" In: PVLDB 6 (Mar. 2015) (cit. on pp. 1, 22).

[39] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. *Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent*. 2017. DOI: 10.48550/ARXIV.1705.09056 (cit. on p. 1).

[40] Andrei Lopatenko and Loreto Bravo. "Efficient Approximation Algorithms for Repairing Inconsistent Databases." In: IEEE. 2007, pp. 216–225. DOI: 10.1109/ICDE.2007.367867 (cit. on p. 6).

[41] Mohammad Mahdavi and Ziawasch Abedjan. "Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning." In: PVLDB 13.12 (2020), pp. 1948–1961. DOI: 10.14778/3407790.3407801 (cit. on pp. 1, 5, 6, 8).

[42] Mohammad Mahdavi and Ziawasch Abedjan. "Baran: Effective error correction via a unified context representation and transfer learning." In: *PVLDB* 13.11 (2020), pp. 1948–1961 (cit. on pp. 21, 22).

[43] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. "Raha: A Configuration-Free Error Detection System." In: SIGMOD. 2019, pp. 865–882. DOI: 10.1145/3299869.3324956 (cit. on pp. 1, 5, 6, 8, 22).

[44] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. "Raha: A configuration-free error detection system." In: SIGMOD. 2019, pp. 865–882 (cit. on pp. 21, 22).

[45] Mark Lukas Möller, Meike Klettke, and Uta Störl. "EvoBench – A Framework for Benchmarking Schema Evolution in NoSQL." In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 1974–1984. DOI: 10.1109/BigData50022.2020.9378278 (cit. on p. 5).

[46] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. "Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations." In: SIGMOD. 2019, pp. 1589–1606. DOI: 10.1145/3299869.3319874 (cit. on p. 2).

[47] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. *TensorFlow-Serving: Flexible, High-Performance ML Serving*. 2017. DOI: 10.48550/ARXIV.1712.06139 (cit. on p. 2).

[48] Mourad Ouzzani, Hossam Hammady, Zbys Fedorowicz, and Ahmed Elmagarmid. "Rayyan-a web and mobile app for systematic reviews." In: Systematic Reviews. 2016. DOI: 10.1186/s13643-016-0384-4 (cit. on p. 22).

[49] Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. "Pattern Functional Dependencies for Data Cleaning." In: PVLDB 13 (2020), pp. 684–697. DOI: 10.14778/3377369.3377377 (cit. on p. 6).

[50] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. "HoloClean: Holistic Data Repairs with Probabilistic Inference." In: PVLDB (2017), pp. 1190–1201. DOI: 10.14778/3137628.3137631 (cit. on pp. 1, 4, 6, 8, 22).

[51] Valerie Restat, Gerrit Boerner, André Conrad, and Uta Störl. "GouDa - Generation of Universal Data Sets: Improving Analysis and Evaluation of Data Preparation Pipelines." In: SIGMOD. 2022. ISBN: 9781450393751. DOI: 10.1145/3533028.3533311 (cit. on pp. 5, 6, 8, 22).

[52] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier." In: KDDM. ACM, 2016, pp. 1135–1144. DOI: 10.1145/2939672.2939778 (cit. on p. 2).

[53] Svetlana Sagadeeva and Matthias Boehm. "SliceLine: Fast, Linear-Algebra-Based Slice Finding for ML Model Debugging." In: SIGMOD. 2021, pp. 2290–2299. DOI: 10.1145/3448016.3457323 (cit. on p. 1).

[54] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. "JENGA: A framework to study the impact of data errors on the predictions of machine learning models." In: EDBT. 2021 (cit. on pp. 5, 6, 8).

[55] Vraj Shah and Arun Kumar. "The ML Data Prep Zoo: Towards Semi-Automatic Data Preparation for ML." In: 2019. DOI: 10.1145/3329486.3329499 (cit. on p. 6).

[56] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. "Towards Benchmarking Feature Type Inference for AutoML Platforms." In: SIGMOD. 2021, pp. 1584–1596. DOI: 10.1145/3448016.3457274 (cit. on p. 1).

[57] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. "Towards Benchmarking Feature Type Inference for AutoML Platforms." In: SIGMOD. 2021, pp. 1584–1596. DOI: 10.1145/3448016.3457274 (cit. on p. 6).

[58] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*. 2013. DOI: 10.48550/ARXIV.1312.6034 (cit. on p. 2).

[59] Michael Stonebraker and Ihab Ilyas. "Data Integration: The Current Status and the Way Forward." In: IEEE 41 (2018), pp. 3–9 (cit. on p. 1).

[60] Jiannan Wang and Nan Tang. "Towards Dependable Data Repairing with Fixing Rules." In: SIGMOD. 2014, pp. 457–468. DOI: 10.1145/2588555.2610494 (cit. on p. 6).

[61] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. "Rafiki: Machine Learning as an Analytics Service System." In: PVLDB 12.2 (2018), pp. 128–140. DOI: 10.14778/3282495.3282499 (cit. on p. 2).