SYDAG: A Synthetic Dataset Generator for Data Integration

Anne Marschner[1,*], Thorsten Papenbrock[1]

[1]Philipps University Marburg, Faculty of Mathematics and Computer Science, Hans-Meerwein-Straße 6, 35043 Marburg, Germany

Abstract

For the development and evaluation of data integration tools, developers require test datasets that represent realistic data integration scenarios. Due to a lack of publicly accessible test datasets with ground truth information, developers often use dataset generators to create their integration scenarios. Existing dataset generators are, however, often limited in functionality and offer only limited configuration options. We therefore develop the synthetic dataset generator SYDAG that generates realistic integration scenarios from real-world seed datasets and offers highly customizable configurations for output fine-tuning. It supports the injection of a wide range of error types and structural changes, which enable the generation of heterogeneous integration scenarios. For the generation, we propose a logical sequence of processing steps and efficiently implement them in the SYDAG system. For the evaluation of SYDAG, we generate integration scenarios of three complexity levels and examine how different schema matchers perform on them. Our results show that the performance of all matchers decreases as the complexity of the integration scenario increases, confirming that SYDAG is capable of generating complex integration scenarios.

Keywords

dataset generation, relational data, data integration

1. Synthetic Data for Integration

In recent years, the volume of stored data has grown significantly across nearly every sector of life [1, 2]. This data is highly diverse [3]. To describe the totality of large, diverse, and challenging amounts of data, scientists use the term *Big Data* [4]. The integration of datasets with such different properties can be very difficult [2].

An essential task for data scientists is to analyze and merge diverse data [5]. Therefore, data integration represents a significant academic research focus and has been a long-standing topic in the data management community [5, 6]. Data integration scenarios often target datasets whose structure or semantics are unknown to the data scientists beforehand [5]. The applied data integration algorithms must, therefore, be able to work with various characteristics of datasets. Consequently, high-quality test datasets that represent diverse scenarios are essential to benchmark any designed integration algorithm. For a reliable evaluation, data scientists also require ground truth information about these datasets [7]. The test datasets are expected to be of high quality, meaning the ground truth must be completely accurate. Furthermore, these datasets should be adaptable to different user requirements [8].

Because the generation of ground truth data is very labor-intensive, there is a ubiquitous lack of publicly accessible real-world datasets with ground truth that can be used to test novel integration methods [7, 9]. An algorithm or domain expert first needs to identify the correspondences in the datasets before creating the ground truth from them. Sometimes, the ground truth needs to be created manually without the help of algorithms [10]. Static test data is useful only to obtain a rough idea of the quality of an algorithm; for a better and more comprehensive evaluation, test data generators are

36th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 29 - October 01, 2025, Regensburg, Germany

*Corresponding author.

anne-marschner@web.de (A. Marschner); papenbrock@informatik.uni-marburg.de (T. Papenbrock)

https://github.com/anne-marschner (A. Marschner);

https://www.uni-marburg.de/en/fb12/research-groups/big_data_analytics/people/thorsten-papenbrock (T. Papenbrock)

0009-0001-7841-5527 (A. Marschner); 0000-0002-4019-8221 (T. Papenbrock)

preferable [2]. These generators can quickly generate different benchmarking scenarios and can be tailored to the user's needs (e.g., degree of heterogeneity, noise or pollution). Therefore, data generators are an effective tool for creating heterogeneous data integration scenarios that represent different types of data sources [7].

The development of data integration algorithms continues to increase [5]. However, while there is much research on integration, there is comparatively less research on test data generators that are needed to evaluate the integration algorithms [2]. Additionally, benchmarks are often limited to specific tasks [6]. This paper addresses these facts and aims to advance the generation of test datasets. To achieve this, we introduce the **Sy**nthetic **Da**taset **G**enerator SYDAG for the creation of data integration scenarios. SYDAG meets the described requirements for the generation of realistic and heterogeneous integration scenarios and introduces various features and configuration options that allow a particularly precise customization and distinguish our approach from existing data generators. The customizations include the horizontal and/or vertical splitting of the input dataset, the injection of a variety of schema and data errors, the adjustment of normalization degrees, and the obfuscation of schema and instance correspondences via merge and shuffle operations on rows and/or columns. Because relational databases are one of the most common formats for structuring data, SYDAG also targets this particular data model as in- and output [11]. We also provide a Graphical User Interface (GUI) to enable an easy way of configuration [9]. We test and evaluate SYDAG's ability to create integration scenarios of varying complexity by measuring the performance of three simple schema matchers in different integration scenarios generated by SYDAG.

The remainder of this paper is structured as follows: First, Section 2 provides a review of related work on dataset generators. Next, Section 3 describes the configuration options and design of SYDAG. Afterwards, Section 4 introduces SYDAG's implementation and accessibility. Then, Section 5 presents our evaluation with three standard matching techniques. Section 6 ends the paper with a conclusion.

2. Related Work

To evaluate the performance of integration tools, benchmarks are essential [12]. A practical approach is to construct these benchmarks from real-world data. This enables developers to test schema matching tools in realistic conditions [13]. An example of this approach is the benchmark Thalia provided by Hammer et al. [14]. Thalia is a collection of over 25 publicly available test datasets, which include a large number of syntactic and semantic heterogeneities. Cabrera et al. presented a similar benchmark, called DIBS [12]. DIBS includes real data integration tasks from diverse domains and additionally provides metrics to assess performance in these scenarios. Crescenzi et al. introduced another benchmark, called Alaska [6], whose ground truth was created by experts. It consists of real datasets but differs from DIBS because of its flexibility. Alaska is based on real data from 71 e-commerce websites, which ensures that the benchmark reflects realistic challenges, and users can select subsets of the data to achieve the desired level of difficulty. Our approach also uses real-world data, but we generate the integration scenario automatically.

It is, in general, difficult to find real-world datasets with ground truth because the creation of ground truth is highly labor-intensive [7]. Moreover, the increasing number of data sources across various sectors and their heterogeneity require many different scenarios [2]. This has led to the creation of dataset generators instead of benchmarks with real-world data. These generators can quickly produce diverse synthetic datasets for benchmarking scenarios. Additionally, users can adapt the generated datasets to their own expectations [7]. Hence, the goal is to generate realistic datasets that reflect heterogeneity and volume.

Panse et al. introduced a test dataset generator called DaPo$^+$ [2, 7], which is an extension of the generator DaPo presented by Hildebrandt et al. [15]. The extended system takes an existing dataset as input and uses it to create multiple datasets with errors and duplicates. Because it supports both relational and non-relational data models, it can generate large and versatile datasets. The output includes the created datasets and ground truth information.

Ioannou et al. developed a generator for schema matching scenarios, which is called EMBench++ [16]. EMBench++ works with relational databases and expects a user-defined configuration. Depending on the configuration, it inserts duplicates and errors, such as misspellings or abbreviations, and simulates time-based changes, such as entity splits. This allows developers to test algorithms in realistic scenarios.

Lee et al. presented eTuner [17], which is a system that automatically optimizes schema matching algorithms by adjusting their parameters. It includes a workload generator that creates new schemata from a given schema by splitting the schema into disjoint sets and introducing errors in attribute labels and instance values. Because it creates schemata and their mappings, it can also be used to generate realistic matching scenarios.

Koutras et al. introduced Valentine [18], an open-source experiment suite for evaluating schema matching techniques. It includes a dataset generator that takes a tabular dataset and a user-defined configuration as input. Valentine's dataset generator, then, performs a horizontal and/or vertical split on the relations while retaining some overlapping columns and/or rows. It also inserts errors into the overlapping entries both at data and schema level. Additionally, it includes a user-friendly interface [9].

The generators that we mentioned so far use real data to create synthetic datasets. However, there are also generators that produce datasets without receiving a data source as input. For example, Alexe et al. introduced STBenchmark [19], which includes two generators for schemata and instances: First, SGen creates a mapping scenario; then, IGen generates the actual data instances corresponding to the created structures. While SGen can utilize either real or generated data, IGen does not require a real data source to generate data instances. The same applies to iBench, presented by Arocena et al. [20]. It is a metadata generator for the synthetic creation of schemata and their mappings from user-defined configurations, which define i.a. desired size and complexity properties, without any seed data. Unlike the other generators, iBench independently creates integrity constraints and other metadata.

Our data generator SYDAG is heavily inspired by related work and subsumes most of the unique features that we listed above: horizontal and vertical splits, error and noise generation, attribute splits and merges, etc. We consolidate and extend these features in a particularly adaptable data generator, which allows the user to make exact configurations and create individual integration scenarios. In contrast to existing works, SYDAG can also automatically change schema normalization degrees. Additionally, it profiles key constraints and offers options for schema and instance shuffling. Overall, SYDAG offers not only more features than every individual approach but also more customization options for to-be-generated datasets. We developed SYDAG for relational datasets, because the relational model is one of the most common data models [11].

3. Synthetic Dataset Generator

We aim to cover a wide selection of functionalities in SYDAG's design and, therefore, combine various features from existing generators with additional methods. For this, we consider how different generation approaches logically build upon each other. Certain processing steps need to be performed before others to generate datasets with desired properties. Figure 1 visualizes the sequence of SYDAG's generation pipeline that meets these requirements.

SYDAG takes a relational (real-world) dataset as input and lets the user specify the desired properties of the matching scenario. The first processing step is the key determination. With the data profiling algorithm HyUCC [21], this step automatically identifies all keys of the input relation; then, it selects the smallest, most likely key of every relation as the relation's primary key. Placing key inference at the beginning is necessary because some of the following steps require keys as input. In particular, the split and noise components depend on these keys to be known, because the keys must be maintained in all newly formed relations [18].

The split component is the second processing step. It incorporates functionalities from Valentine's dataset generator, which supports both horizontal and vertical splits (and combinations of both). It also allows users to specify the degree of overlap for rows and columns [18]. In addition, SYDAG provides configuration options to control the distribution of non-overlapping columns or rows between the new

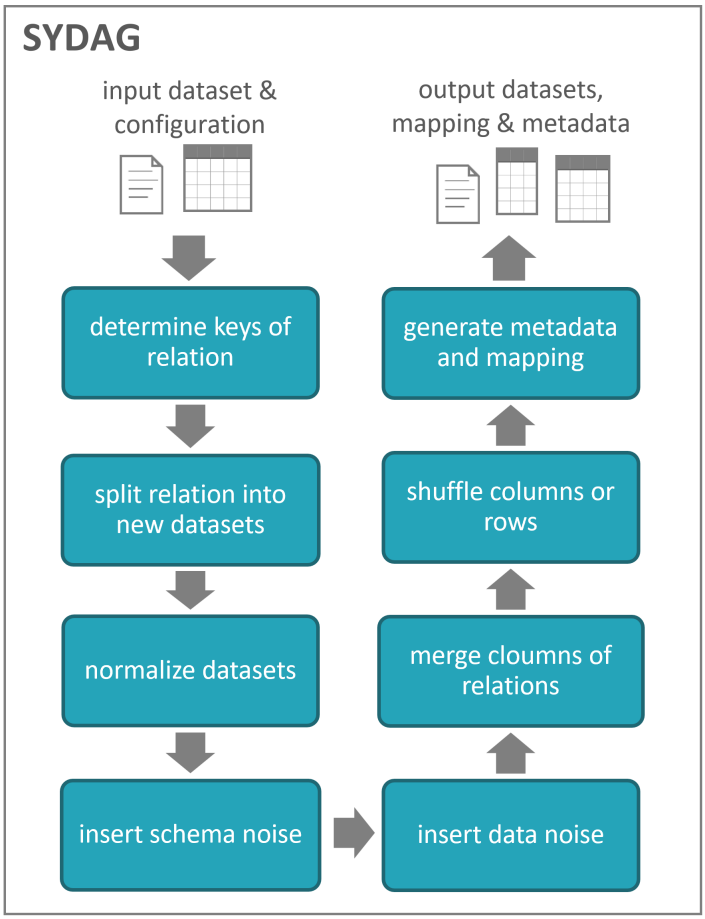

Figure 1: Overall procedure of SYDAG: An input dataset and a user-defined configuration are received. After that, eight steps are executed sequentially. Finally, the created files are output.

relations, which enables one relation to have more columns than the other. For horizontal splits, the user can choose whether the overlapping rows should be selected within a block or randomly scattered. The split needs to be executed in the beginning of the process, because it determines the number of new datasets that SYDAG creates and, therefore, provides the basis for the dataset generation. SYDAG splits the input relation into either two or four new relations, each representing a distinct dataset of the final integration scenario. By placing the split at the start, it is possible to select the error levels and structural changes individually for each newly created dataset, which enables the creation of heterogeneous integration scenarios.

Normalization to Boyce-Codd Normal Form (BCNF) follows as the next component and optional computation step. It causes the datasets created during the split to no longer contain only one but potentially multiple linked relations. The user can choose whether SYDAG should apply normalization to BCNF individually for each of the two (or four) current datasets. Additionally, a percentage can be specified that determines how many of the possible decomposition steps should be executed. To execute the normalization, SYDAG applies the schema normalization algorithm Normalize [22]. Normalize automatically profiles functional dependencies and uses them to transform relational datasets into BCNF. We execute the normalization after the split and before any noise injection to create meaningful decompositions and foreign key constraints. When SYDAG later adds noise to the relations, the user can choose to preserve the key and foreign-key constraints or allow them to be broken.

The following component inserts noise, i.e., errors into the schemata. Because normalization has already been applied, any schema within a dataset can receive errors. The split(s) that SYDAG performed in the previous component provide information about the overlapping columns or rows. This enables the

selection of attributes in the schema that share overlapping entries with other relations. These attributes are candidates for noise injection. For each newly created dataset, there are configuration options for adding noise to the schema. The user can choose whether to include noise and, if so, whether key attributes should be affected. If the user enables noise, she must specify the percentage of overlapping attributes with other datasets that the generator will modify. The user can select from nine error methods to introduce the noise or choose to delete the schema completely, which results in a relation without headers. The available error methods include some approaches from existing generators, such as perturbing the column names via prefixing, abbreviation or vowel dropping (Valentine [18]), and replacing column names with synonyms or changing names to random characters (ETuner [17]). If a relation includes these errors, it causes discrepancies with the names of the overlapping columns from other relations and, hence, complicates integration scenarios. SYDAG additionally adds another four noise options (see later), but also allows the user to apply no changes.

To generate complex scenarios, generators should not only add errors into the schema but also into the instance data [16]. For this purpose, data noise injection is the next component in SYDAG's pipeline. Similar to the schema noise component, SYDAG can use the information from the split to identify overlapping rows or columns and insert errors specifically into their entries, as inserting errors in non-overlapping records has less impact on integration scenarios. The configurations for data errors are similar to those for the schema. For each created dataset, the user can select whether the generator should introduce noise. This includes the specification whether SYDAG is allowed to break the key constraints by adding errors to key columns. Additionally, the user can select the percentage of noisy rows or columns based on the split type. For horizontal splits, the generator inserts a user specified percentage of noise into the overlapping rows, while for vertical splits, it affects the overlapping columns. Furthermore, the user can specify a percentage that indicates how many of the entries within a for noise selected column or row should receive errors. There are 14 available noise methods that the user can choose from (see Section 4). Again, some error methods are inspired by existing generators: random typing errors based on keyboard probabilities and random numerical value changes based on value distributions (Valentine [18]), as well as random data format changes (ETuner [17]), word permutations, missing value injections, and word abbreviations in string values (EMBench++ [16]).

The merge component follows next in the pipeline and can merge two columns of a relation into a single column. This procedure is inspired by eTuner [17], but SYDAG can execute multiple of these merges. We visualize an example of the process in Figure 2. Each merge creates a new attribute that contains concatenated values. SYDAG allows the user to individually select for each dataset whether to apply the merge component. This includes the specification of the percentage of overlapping columns to merge. Because we want to enable different errors within the linked values, we place the merge component after noise insertion.

name	surname	age
Jane	Smith	22
Tim	Miller	45
Mike	Brown	14

name, surname	age
Jane, Smith	22
Tim, Miller	45
Mike, Brown	14

Figure 2: Example of the merge component for two columns.

The shuffle component is the last processing step before the output creation. For each created dataset, the user can choose between no changes, shuffling rows, or shuffling columns. This means that relations representing the same real-world entity no longer maintain the same order of entries, which can be a significant challenge for matchers that consider the order of attributes or tuples [23]. Placing this step at the end is necessary to allow users to select the block overlap as a split option. The shuffling later disperses the overlapping rows/columns over the entire datasets. These rows originally belong to one block and may share some similarities, but when SYDAG shuffles them, it makes it harder to rediscover these blocks. The difference to using random overlap is that in that case SYDAG takes the overlapping rows directly from different locations, meaning they might share less similarities.

After processing the datasets, SYDAG creates the output. The output covers the created relations grouped by dataset, the metadata information on keys and foreign-keys, and the mapping information, i.e., the gold standard information for a perfect matching of the resulting schemata. The mapping information specifies which attributes in the new relations correspond to the original attributes. The output format is inspired by the generators iBench [17] and eTuner [20], which also generate and output precise mappings for the generated datasets. Developers can use these mappings to evaluate their integration strategies. The output comprises two or four datasets, each with one or multiple relations depending on the individual normalization steps.

4. SYDAG Implementation

To realize SYDAG as a practical application, we implemented the components described in Section 3 in a typical backend-frontend architecture. We provide the core details of this implementation here and further technical details online [24].

4.1. Backend

Our backend consists of multiple classes that SYDAG utilizes during the generation process. We categorize the classes based on their functionality and explain the core components of SYDAG's generation algorithm in this section.

The *Splitting Components* are an important part of SYDAG, because they divide the input relation into new relations. The 'Split' class provides methods for the different split options. For the vertical split, it provides a method that generates a uniformly random column overlap: First, the method adds the key columns to both newly created relations. Then, it randomly selects non-key columns proportional to the user-specified percentage of overlap and inserts them into both new relations. After that, it randomly distributes the remaining columns to the new relations according to the configuration. For the horizontal split, there are two different methods. One of the two horizontal methods creates a block overlap; it, first, selects a random start index for the overlapping rows; then, it determines the end index based on the user-selected percentage of overlap and inserts the rows between start and end as the first rows in both new relations; after that, it distributes the remaining rows to the new relations according to the configuration. The other horizontal split method creates an overlap of random rows; unlike block overlap, it picks overlapping rows randomly from the entire dataset rather than selecting consecutive rows. If the user chooses both split types, SYDAG first applies the horizontal split to create two new relations; then, it splits both of the created relations vertically, resulting in four relations.

The *Structure Change Components* include four classes that SYDAG uses to modify the structure of the relations. Their 'Merge' class is responsible for merging columns within a relation. It relies on the user-defined percentage to determine how many of the overlapping columns should be merged. The merge process is repeated until the desired number of merged columns is reached. In each step, a method identifies the best column pair for merging by calculating a score for all possible pairs. Pairs receive a higher score if they share the same enumeration type, if their column indices are close to each other, and if they have not yet been merged with another column. The method selects the column pair with the highest score and, then, merges the attribute names and data entries of the columns into a new column; the new column gets inserted into the relation and the two original columns are removed. The other classes in the Structure Change Components are responsible for normalization. The 'Normalization' class provides methods that decompose a relation into BCNF. Its core method first applies the Normalize algorithm [22] to find BCNF-compliant relations. It stores the column and key indices of these relations in 'IndexSummary' objects. Then, the method adjusts the indices to the input relation so that they reference the correct column positions. Afterwards, the method executes as many of Normalize's decomposition steps as the user specified in the configuration. The final step is the creation of the selected relations. The method extracts the columns corresponding to the indices in the 'IndexSummary' object, removes the overlapping rows, and then outputs the generated relations.

The *Noise Insertion Components* are the most essential part of SYDAG and cover the following error methods:

1. Removal of all vowels from a given string
2. Abbreviation of all words' first letters in a string
3. Shortening of all words in a string to random lengths
4. Shuffling of all letters within a string
5. Shuffling of all words within a string
6. Generation of random strings with length in [1,10]
7. Adding of random character prefixes of length [1,4]
8. Replacing words in a string with synonyms
9. Replacing words in a string with their translations
10. Generation of `null` values
11. Insertion of phonetic errors in strings
12. Insertion of OCR errors in strings
13. Insertion of random typing errors in strings
14. Reformatting strings by swapping "-", "_", ".", and " "
15. Abbreviation of words in strings to random lengths
16. Generation of a random numeric value from a column's normal distribution of values
17. Generation of a random numeric outlier for a column
18. Mapping of string values to numeric values

SYDAG can apply Methods 1 to 9 to the schema of the relations and Methods 5 to 18 to the instance data. Methods 8 and 9 rely on external API calls: Synonyms for data entries are generated using the Datamuse API [25], while translations are provided by the MyMemory API [26]. If the user selects the methods that generate synonyms or translations, the generation process of SYDAG can take longer than without this selection, because calling the APIs is more time-consuming than using a naive error method.

The 'SchemaNoise' class is part of the Noise Insertion Components and extends the 'Noise' class. We use it to add errors to the attribute names. It includes a method, which receives a relation and the user's configuration. If chosen, the method deletes the schema; otherwise, it determines the overlapping attributes and includes or excludes the key columns from the noise based on the configuration. Then, the method selects attributes uniformly at random from the overlapping attributes to receive noise. For each of the selected attributes, it applies one of the user-selected error methods. It reuses a method only after all other selected methods have been applied at least once. An exception to this are Methods 1 and 5 because they are not always applicable. They require the attribute name to contain vowels or several words. If selected, SYDAG first attempts to use Methods 1 and 5. If this is not possible for the current attribute, it uses another user-selected error method. If the user selected only Methods 1 and 5, but neither is applicable, Methods 6 and 7 serve as fallback.

The 'Data Noise' class also extends the 'Noise' class. We use it to insert errors into instance data. It provides two different methods, which are used depending on the chosen split type. SYDAG perturbs the columns in case of a vertical split, the rows in case of a horizontal split, and both when a double split is performed. The row perturbation consists of many sequential steps. First, the method calculates the number of rows that will receive errors based on the user-specified percentage. If the value exceeds 0, the method selects that many row indices uniformly at random. Next, the method filters the column indices to determine where it can insert errors. If the user chooses to preserve key constraints, the method removes the key indices from the available error positions. In case of a double split, the method also removes the overlapping column indices because these columns already received noise through the column perturbation. Then, for each row selected for noise, the method determines the number of affected entries based on the user-defined percentage. After that, it randomly selects the calculated number of entries from the available error positions in the row. For each selected entry, it checks which of the user-specified error methods are applicable, then selects one uniformly at random and executes it. The new entry replaces the original entry in the relation. The column perturbation method uses

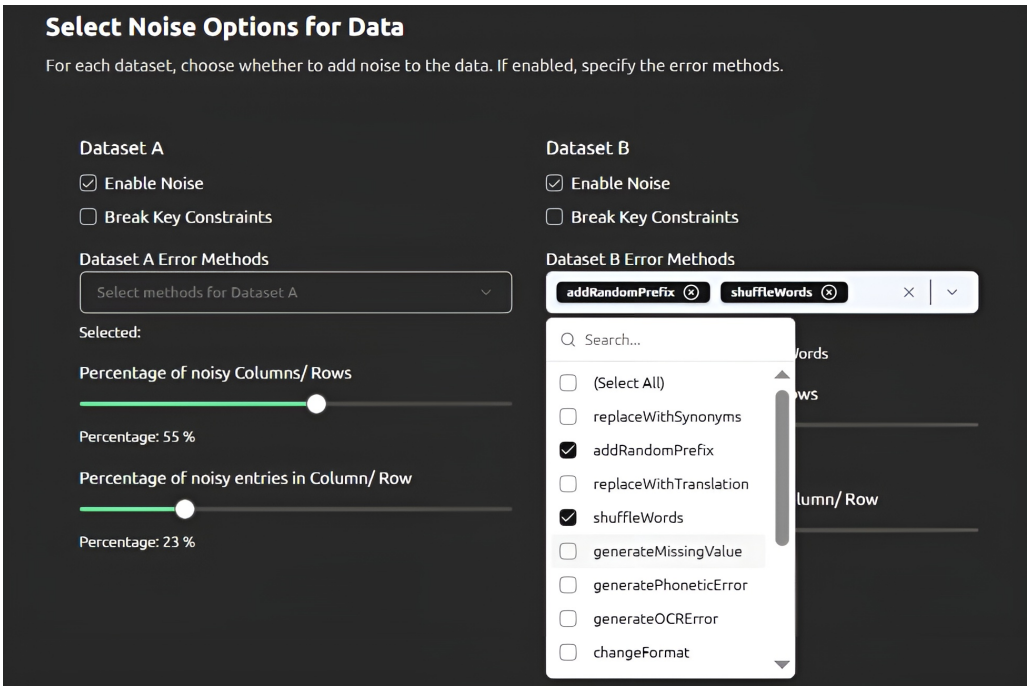

Figure 3: Example of SYDAG's graphical user interface.

a similar approach. However, a difference lies in the selection of the entries that receive noise. This method selects the columns that will receive noise while maintaining the same ratio of numeric to alphanumeric columns as in the relation. Then it calls two individual methods: One of them inserts errors into the selected numeric columns and the other into the alphanumeric columns. Both of these methods first randomly select the entries in the column and, then, apply a random error method. Lastly, the perturbed relation is output.

The last important group are the *File Processing Components*, which include the 'CSVTool' class. This class includes methods that write the generated relations to CSV-files. Depending on the user's configuration, SYDAG may shuffle columns or rows before writing the CSV-file. Therefore, three different methods exist: one for writing the file without shuffling, one for shuffling columns before writing, and one for shuffling rows before writing. In addition, there is a method that generates a TXT-file containing the key relationships for a generated dataset. Finally, another method writes the entire mapping between the generated datasets to a TXT-file.

4.2. Frontend

To make SYDAG user-friendly, we provide a GUI through which users can specify input parameters and generation configurations. The GUI also offers an option to upload a JSON configuration file containing all of the input parameters; SYDAG provides a template with an example of a JSON configuration file to create these configurations. Figure 3 shows a screenshot of the GUI for configuring data noise settings. For a simple and efficient application startup, we use Docker to containerize SYDAG with its backend and frontend components [27]. Our generator is publicly available on GitHub [24].

5. Evaluation

The goal of our evaluation is to assess SYDAG's ability to generate integration scenarios of different complexity that actually challenge schema matchers. To achieve this, we alter SYDAG's configuration parameters to create different integration scenarios. After that, we evaluate the performance of three schema matchers in these scenarios.

Table 1
Characteristics of the four datasets used for evaluation.

Dataset	Rows	Numeric Columns	Alphanumeric Columns	Key Columns
Bridges	108	4	9	1
Diabetes	768	9	0	3
Mental	5000	7	13	1
Gym	973	13	2	2

Table 2
Average F-measure values across the different datasets for each matcher and complexity level.

F-Measure			
Complexity Level	Level 1	Level 2	Level 3
Levenshtein Matcher	1.0000	0.5905	0.1383
Jaccard Matcher	0.9393	0.3338	0.2482
Distinct-Count Matcher	0.6942	0.2205	0.1545

5.1. Experimental Setup

For the evaluation, we choose four datasets with diverse properties to test SYDAG across different requirements. Table 1 summarizes the characteristics of the four datasets. The dataset *'Bridges'* [28] is available in the UCI Machine Learning Repository; *'Diabetes'* [29], *'Mental'* [30], and *'Gym'* [31] can all be found on Kaggle. We apply SYDAG to generate integration scenarios of three complexity levels for each test dataset, which creates 12 scenarios in total. To create the higher complexity levels, we increase the percentages of structural changes, noise and shuffling, and decrease the overlap percentages of the split; the different SYDAG configuration files and definitions can be found on GitHub [24].

To measure how different matchers perform on our created integration scenarios, we use a schema-based Levenshtein Matcher, an instance-based Jaccard Matcher (with 2-gram tokenization), and a Distinct-Count Matcher that calculates the similarity of two columns by counting their distinct entries and dividing the smaller count by the larger count [32, 33, 34]. The schema matching tool Schematch [34] provides their implementations. To evaluate the matching, we calculate the F-measure using the smallest similarity value that describes an actual match as our threshold [34].

5.2. Analysis of Results

We compare the overall performance of the three matchers by calculating the average F-measure across all integration scenarios for each complexity level (see Table 2). More detailed analyses of matcher's performances on the individual datasets are available on GitHub [24].

Figure 4 visualizes the results of the average performances of the three matchers. For all matchers, we notice a significant drop in performance across the complexity levels. For both the Distinct-Count Matcher and the Jaccard Matcher, we observe that the performance drops are higher from the first to the second complexity level than from the second to the third. The performance of the Distinct-Count Matcher decreases by 0.47 between Complexity Levels 1 and 2 and by 0.07 between Complexity Levels 2 and 3, which shows that even small percentages of inserted errors present a challenge, because they distort the number of distinct elements in the columns. For the Jaccard Matcher, the performance drops by 0.6 from Complexity Level 1 to 2 and by 0.09 from Complexity Level 2 to 3. This means that when the goal is to identify all matches, the error insertion in the second complexity level already has a strong impact on the performance. It also shows that error insertion presents a significant challenge, even when only minor structural changes are made.

In contrast, the Levenshtein Matcher shows a different behavior. We observe a decrease of 0.41 between the first and second complexity levels and a further drop of 0.45 between the second and third,

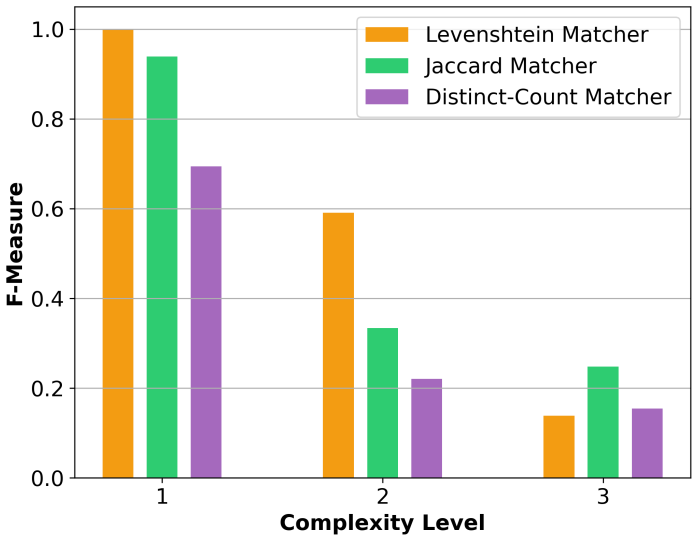

Figure 4: Overview of the performance of three matchers, averaged across all four test datasets.

reflecting a consistent rise in difficulty across levels. This indicates that even the small percentages of errors that we created in Complexity Level 2 make matching more difficult. However, since we mostly use error methods that do not make attribute names unrecognizable, the matcher can still ensure relatively high performance. The performance drop between Complexity Levels 2 and 3 is caused by the high number of inserted errors, the use of complex error methods and structural changes, which include in particular complex merged attribute names.

For all matchers, we observe a decline in performance across the complexity levels. We can identify some common configurations that increase the overall complexity. These include the number of splits. The more datasets SYDAG splits an integration scenario into, the greater the challenges for the integration tool. Another aspect are the structural changes. As SYDAG applies more diverse structural changes or shuffle options, the integration of the datasets becomes increasingly heterogeneous and challenging. Furthermore, data and schema noise play an important role. The more noisy relations we create and the higher the noise percentages are chosen, the more challenging it becomes for matchers to identify overlapping attributes and recognize that they represent the same real-world entities. Overall, we can state that the complexity of the integration scenarios clearly increases with increasing noise percentages. With SYDAG, users can, hence, create individual challenges by fine-tuning the difficulty of their scenarios to the weaknesses of the integration tools they are testing.

6. Conclusion

We introduced the synthetic dataset generator SYDAG, which developers can use to create customized data integration scenarios. Our performance evaluation of three schema matchers showed that SYDAG is able to generate complex integration scenarios. SYDAG offers the following advantages over existing dataset generators: It combines error insertion and structural changes, enabling it to generate integration scenarios with high heterogeneity. Additionally, SYDAG allows the user to customize the generation by specifying percentages of modifications and selecting combinations of error methods. SYDAG is applicable to datasets with and without headers and contains a built-in key identification tool, meaning the user does not have to specify the keys.

Despite its advantages, SYDAG requires seed input data, because it does not generate data itself and only restructures and perturbs the input. Additionally, SYDAG is currently limited to CSV input files, but extensions to relational databases and semi-structured formats are planned.

Declaration on Generative AI

During the preparation of this work, the authors used DeepL for grammar and spell checking. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] C. Manzano, A. Miskolczi, H. Stiele., V. Vybornov, T. Fieseler, S. Pfalzner, Learning from the present for the future: the Jülich LOFAR long-term archive, in: Astronomy and Computing, volume 48, 2024, pp. 1–11. doi:10.1016/j.ascom.2024.100835.

[2] F. Panse, W. Wingerath, B. Wollmer, Towards scalable generation of realistic test data for duplicate detection, 2023. arXiv:2312.17324.

[3] N. Marz, J. Warren, Big Data : Principles and best practices of scalable real-time data systems, Manning Publications Co., 2015.

[4] D. T. Speckhard, T. Bechtel, L. M. Ghiringhelli, M. Kuban, S. Rigamonti, C. Draxl, How big is big data?, 2024. arXiv:2405.11404.

[5] A. Bogatu, N. W. Paton, M. Douthwaite, A. Freitas, Voyager: Data discovery and integration for data science, in: Proceedings of the International Conference on Extending Database Technology (EDBT), 2022, pp. 537–548. doi:10.48786/edbt.2022.47.

[6] V. Crescenzi, A. D. Angelis, D. Firmani, M. Mazzei, P. Merialdo, F. Piai, D. Srivastava, Alaska: A flexible benchmark for data integration tasks, 2021. arXiv:2101.11259.

[7] F. Panse, M. Klettke, J. Schildgen, W. Wingerath, Similarity-driven schema transformation for test data generation, in: Proceedings of the International Conference on Extending Database Technology (EDBT), 2022, pp. 408–413.

[8] F. Panse, A. Düjon, W. Wingerath, B. Wollmer, Generating realistic test datasets for duplicate detection at scale using historical voter data, in: Proceedings of the International Conference on Extending Database Technology (EDBT), 2021, pp. 570–581. doi:10.5441/002/edbt.2021.67.

[9] C. Koutras, K. Psarakis, G. Siachamis, A. Ionescu, M. Fragkoulis, A. Bonifati, A. Katsifodimos, Valentine in action: Matching tabular data at scale, in: Proceedings of the VLDB Endowment, volume 14, 2021, pp. 2871–2874. doi:10.14778/3476311.3476366.

[10] D. Ritze, O. Lehmberg, C. Bizer, Matching HTML tables to DBpedia, in: Proceedings of the International Conference on Web Intelligence, Mining and Semantics (WIMS), 2015, pp. 1–6. doi:10.1145/2797115.2797118.

[11] J. Minder, L. Brandenberger, L. Salamanca, F. Schweitzer, Data2Neo - a tool for complex Neo4j data integration, 2024. arXiv:2406.04995.

[12] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, R. D. Chamberlain, DIBS: A data integration benchmark suite, in: Proceeding of the ACM/SPEC International Conference on Performance Engineering Companion (ICPE), 2018, pp. 25–28. doi:10.1145/3185768.3186307.

[13] F. Duchateau, Z. Bellahsene, E. Hunt, XBenchMatch: a benchmark for XML schema matching tools, in: Proceedings of the VLDB Endowment, 2007, pp. 1318–1321.

[14] J. Hammer, M. Stonebraker, O. Topsakal, THALIA: Test harness for the assessment of legacy information integration approaches, in: Proceedings of the International Conference on Data Engineering (ICDE), 2005, pp. 485–486. doi:10.1109/ICDE.2005.140.

[15] K. Hildebrandt, F. Panse, N. Wilcke, N. Ritter, Large-scale data pollution with Apache Spark, in: IEEE Transactions on Big Data, volume 6, 2020, pp. 396–411. doi:10.1109/TBDATA.2016.2637378.

[16] E. Ioannou, Y. Velegrakis, EMBench++: Data for a thorough benchmarking of matching-related methods, in: Semantic Web, volume 10, 2019, pp. 435–450. doi:10.3233/SW-180331.

[17] Y. Lee, M. Sayyadian, A. Doan, A. S. Rosenthal, ETuner: Tuning schema matching software using

synthetic scenarios, in: VLDB Journal, volume 16, 2007, pp. 97–122. doi:https://doi.org/10.1007/s00778-006-0024-z.

[18] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, A. Katsifodimos, Valentine: Evaluating matching techniques for dataset discovery, in: Proceedings of the International Conference on Data Engineering (ICDE), 2021, pp. 468–479. doi:10.1109/ICDE51399.2021.00047.

[19] B. Alexe, W. C. Tan, Y. Velegrakis, STBenchmark: towards a benchmark for mapping systems, in: Proceedings of the VLDB Endowment, volume 1, 2008, pp. 230–244. doi:10.14778/1453856.1453886.

[20] P. C. Arocena, B. Glavic, R. Ciucanu, R. J. Miller, The IBench integration metadata generator, in: Proceedings of the VLDB Endowment, volume 9, 2015, pp. 108–119. doi:10.14778/2850583.2850586.

[21] T. Papenbrock, F. Naumann, A hybrid approach for efficient unique column combination discovery, in: Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), 2017, pp. 195–204.

[22] T. Papenbrock, F. Naumann, Data-driven schema normalization, in: Proceedings of the International Conference on Extending Database Technology (EDBT), 2017, pp. 342–353.

[23] U. Leser, F. Naumann, Informationsintegration : Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen, 1 ed., dpunkt.verlag, 2007.

[24] A. Marschner, SYDAG, 2025. URL: https://github.com/anne-marschner/SYDAG, last accessed 6-July-2025.

[25] Datamuse api, 2016. URL: https://www.datamuse.com/api/, last accessed 6-July-2025.

[26] Mymemory, 2015. URL: https://mymemory.translated.net/doc/spec.php, last accessed 6-July-2025.

[27] K. Matthias, S. P. Kane, Docker Praxiseinstieg, 2 ed., mitp Verlag, 2020.

[28] Y. Reich, S. Fenves, Pittsburgh Bridges, UCI Machine Learning Repository, 1990. URL: http://archive.ics.uci.edu/dataset/18/pittsburgh+bridges, last accessed 6-July-2025.

[29] H. Rahman, Diabetes Dataset, 2024. URL: https://www.kaggle.com/datasets/hasibur013/diabetes-dataset, last accessed 6-July-2025.

[30] I. Ramzan, Remote Work & Mental Health, 2024. URL: https://www.kaggle.com/datasets/iramshahzadi9/remote-work-and-mental-health, last accessed 6-July-2025.

[31] V. Khorasani, Gym Members Exercise Dataset, 2024. URL: https://www.kaggle.com/datasets/valakhorasani/gym-members-exercise-dataset, last accessed 6-July-2025.

[32] A. Doan, A. Halevy, Z. Ives, Principles of Data Integration, Morgan Kaufmann, 2012. doi:https://doi.org/10.1016/C2011-0-06130-6.

[33] N. Golov, A. Filatov, S. Bruskin, Efficient exact algorithm for count distinct problem, in: Computer Algebra in Scientific Computing, 2019, pp. 67–77. doi:10.1007/978-3-030-26831-2_5.

[34] A. Vielhauer, Schematch, 2024. URL: https://github.com/avielhauer/schematch, last accessed 6-July-2025.