PHILIPPS UNIVERSITY MARBURG
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BIG DATA ANALYTICS GROUP

SYDAG: A Synthetic Dataset Generator for Data Integration Scenarios

Bachelor Thesis
Winter Term 2024/25

Submitted by:

Anne Marschner

Matriculation Number: 3559244

Supervisor: Prof. Dr. Thorsten Papenbrock

Marburg, 18.03.2025

Abstract

For the development and evaluation of data integration tools, developers need test datasets that represent realistic data integration scenarios. However, there is a lack of publicly accessible test datasets with ground truth, which is why developers use dataset generators to create data integration scenarios. The existing dataset generators are limited in functionality and offer the user only limited configuration options. This highlights the need to develop a publicly available dataset generator that focuses on customizable configurations. In addition, it should support a wide range of error types and structural changes in order to enable the generation of heterogeneous integration scenarios. We develop the synthetic dataset generator SYDAG, which meets these requirements. To achieve this, we analyze the possible components of a dataset generator and from that we create an efficient composition of the components of SYDAG. We integrate the analyzed components into the implementation in a logical sequence. For the evaluation, we divide the configuration options of SYDAG into three complexity levels, which generate integration scenarios of varying complexity. Furthermore, we examine three different schema matchers: the Levenshtein Matcher, the Jaccard-Instance Matcher, and the Distinct-Count Matcher. We evaluate the performance of these matchers using integration scenarios corresponding to the three complexity levels. Our results show that the performance of all matchers decreases as the complexity of the integration scenario increases. This confirms that SYDAG is capable of generating integration scenarios of varying complexity.

Zusammenfassung

Für die Entwicklung und Evaluierung von Datenintegrationstools benötigen Entwickler Testdatensätze, die realistische Datenintegrationsszenarien repräsentieren. Es herrscht jedoch ein Mangel an öffentlich zugänglichen Testdatensätzen mit Ground Truth, weshalb Entwickler Datensatzgeneratoren nutzen, um Datenintegrationsszenarien zu erzeugen. Die bereits existierenden Datensatzgeneratoren sind auf bestimmte Funktionen limitiert und bieten dem Nutzer nur beschränkte Konfigurationsmöglichkeiten. Daraus ergibt sich die Notwendigkeit der Entwicklung eines öffentlich zugänglichen Datensatzgenerators der den Fokus auf individuell anpassbare Konfigurationen legt. Zudem soll er eine möglichst breite Palette an Fehlerarten und Strukturveränderungen ermöglichen, um heterogene Integrationsszenarien generieren zu können. Wir entwickeln den synthetischen Datensatzgenerator SYDAG, der diese Anforderungen erfüllt. Dafür führen wir eine Analyse der möglichen Komponenten eines Datensatzgenerators durch und erstellen daraus eine effiziente Zusammensetzung der Bestandteile von SYDAG. Die analysierten Komponenten integrieren wir in der Implementierung in einer sinnvollen Reihenfolge. Für die Evaluierung gliedern wir die Konfigurationsmöglichkeiten von SYDAG in drei Komplexitätsstufen, die unterschiedlich komplexe Integrationsszenarien erzeugen. Zudem betrachten wir drei verschiedene Schema Matcher: den Levenshtein Matcher, den Jaccard-Instance Matcher und den Distinct-Count Matcher. Wir bewerten die Leistung dieser Matcher anhand von Integrationsszenarien, die den drei Komplexitätsstufen zugehörig sind. Unsere Ergebnisse zeigen, dass die Leistung aller Matcher mit steigender Komplexität des Integrationsszenarios abnimmt. Dies bestätigt, dass SYDAG in der Lage ist Integrationsszenarien unterschiedlicher Komplexität zu generieren.

Contents

Abstract I

Zusammenfassung II

List of Figures V

List of Abbreviations VI

List of Tables VII

1 Introduction 1
- 1.1 Research Questions . . . 2
- 1.2 Structure . . . 2

2 Literature Review 3
- 2.1 Basic Knowledge of Relational Databases . . . 3
  - 2.1.1 Constraints . . . 3
  - 2.1.2 Anomalies and Normalization . . . 4
- 2.2 Introduction to Data Integration . . . 8
  - 2.2.1 Faulty Data . . . 8
  - 2.2.2 String Matching and Similarity Measures . . . 9
  - 2.2.3 Performance Metrics . . . 12
  - 2.2.4 Schema Matching and Mapping . . . 13
- 2.3 Related Work . . . 14

3 Design 16
- 3.1 Approaches and Methods for Generation of Datasets . . . 16
- 3.2 Input, Output and Configuration Parameters . . . 18
- 3.3 Architecture of SYDAG . . . 20

4 Implementation 22
- 4.1 Selection of Programming Languages and Tools . . . 23
- 4.2 Backend . . . 24
- 4.3 Frontend . . . 30

5 Evaluation 31
- 5.1 Information on Datasets . . . 32
- 5.2 Creation of different Complexity Levels . . . 33
- 5.3 Schema Matchers and Metrics . . . 36
- 5.4 Analysis of Results . . . 38
  - 5.4.1 Levenshtein Matcher . . . 40

5.4.2 Jaccard-Instance Matcher 43
5.4.3 Distinct-Count Matcher 45
5.4.4 Overall Comparison 47

6 Conclusion **49**

References **50**

List of Figures

1 Visualization of the five nested normal forms of relations [12]. 6
2 Illustration of the decomposition of a relation into two new relations during the transformation to BCNF [15]. 7
3 Overall procedure of SYDAG: An input dataset and a user-defined configuration are received. After that, six steps are executed sequentially. The created files are output in the final step. 21
4 Example of the result of the join component for two columns of a relation. . 22
5 Overview of the components of SYDAG's backend. The classes are grouped by the different processing steps. 25
6 Activity diagram for the 'execute' method of the 'Generator' class of SYDAG. 30
7 Example of SYDAG's graphical user interface. 31
8 Comparison of the performance metrics of the Levenshtein Matcher at two different thresholds. 41
9 Comparison of the performance metrics of the Jaccard-Instance Matcher at two different thresholds. 44
10 Comparison of the performance metrics of the Distinct-Count Matcher at two different thresholds. 46
11 Overview of the performance of three matchers, averaged across all four test datasets. 49

List of Abbreviations

1NF	First Normal Form
2NF	Second Normal Form
3NF	Third Normal Form
4NF	Fourth Normal Form
5NF	Fifth Normal Form
API	Application Programming Interface
BCNF	Boyce-Codd Normal Form
CORS	Cross-Origin Resource Sharing
DBMS	Database Management System
FD	Functional Dependency
GUI	Graphical User Interface
MVD	Multivalued Dependency
OCR	Optical Character Recognition
SYDAG	Synthetic Dataset Generator
TD	Transitive Dependency
UCC	Unique Column Combination

List of Tables

1 Summary of the characteristics of the datasets used for evaluation. 33
2 Performance results of the Levenshtein Matcher, Jaccard-Instance Matcher, and Distinct-Count Matcher on the four test datasets and their respective complexity levels. The values are measured under Threshold 1 (Lowest Relevant Score) and Threshold 2 (Mean minus Standard Deviation). 39
3 Presentation of the average F-measure values across the different datasets for each matcher and complexity level. A distinction is made between the two thresholds. 48

1 Introduction

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]. Big Data can be characterized by the *Four V's*, which highlight key challenges of data integration [2],[4]. The first, volume, refers to the fact that the size of datasets increases. Variety describes that datasets are becoming increasingly heterogeneous and differently structured. Velocity refers to the rapid pace at which data is being generated and modified. Lastly, veracity addresses that data is becoming less reliable because it can be incomplete and error-prone. It can be very difficult to integrate datasets with such different properties [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 include datasets whose structure or semantics the data scientists do not know beforehand [5]. The integration algorithms that they apply must, therefore, be able to work with various characteristics of datasets. Consequently, to benchmark a designed integration algorithm, it is essential to have high-quality test datasets that represent diverse scenarios. For a reliable evaluation, one also needs 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].

There is a lack of publicly accessible real-world datasets with ground truth that can be used to test integration methods because the generation of ground truth is very labor-intensive [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 only useful to obtain a rough idea of the quality of an algorithm. For a better and more comprehensive evaluation, test data generators are preferable [2]. These generators can quickly generate different benchmarking scenarios and can be tailored to the user's needs, such as the degree of heterogeneity. Therefore, they are effective for creating heterogeneous data schemas that represent different 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 thesis addresses these facts and aims to advance the generation of test datasets. To achieve this, we introduce a **Sy**nthetic **Da**taset **G**enerator called SYDAG for the creation of data integration scenarios. SYDAG is developed to meet the described expectations of an effective generator. In order to distin-

guish it from existing generators, we introduce certain features and configuration options that allow a particularly precise customization. This includes the horizontal and/or vertical splitting of the input dataset. SYDAG also enables the integration of schema and data errors, as well as normalization, joins, and shuffling of rows or columns. Because relational databases are one of the most common formats for structuring data, SYDAG works with these as input [11]. In addition, we provide a Graphical User Interface (GUI) because it allows the user to set the configuration in an easy way [9]. We test and evaluate SYDAG's ability to create integration scenarios of varying complexity. This includes an analysis of the performance of three schema matchers in different integration scenarios created by SYDAG.

1.1 Research Questions

The objective of this thesis is to develop the dataset generator SYDAG while we address the following research questions:

1. What approaches and methods are suitable components for the generation of synthetic datasets that represent data integration scenarios?

2. How can we effectively integrate the various generation components into a single generator for creating data integration scenarios?

3. How can users configure the synthetic dataset generator SYDAG to produce data integration scenarios with varying levels of complexity?

4. What effect do different levels of complexity in data integration scenarios created with SYDAG have on the performance of the Levenshtein Matcher, Jaccard-Instance Matcher, and Distinct-Count Matcher?

1.2 Structure

The thesis is structured into different sections. We begin with a literature review in Section 2, which covers basic concepts of relational databases and data integration, as well as related work. Next, in Section 3, we describe the design of SYDAG. This includes details about the features, configuration options, and architecture of the generator. After this, in Section 4, we focus on the implementation of SYDAG and explain both the backend and frontend, along with the tools that we use for our implementation. This is followed by the evaluation of the generator in Section 5, where we describe the several test datasets, schema matchers, and metrics that we use to measure and evaluate the results. The thesis ends with the conclusion in Section 6.

2 Literature Review

This chapter is intended to provide an understanding of the problem domain and reflect the current state of the art. Section 2.1 explains the basic knowledge of relational databases. The Section contains various characteristics of such databases that are needed in order to understand this thesis. It is followed by Section 2.2, which introduces the necessary information about data integration. Building on this, the related work on dataset generators is examined in Section 2.3.

2.1 Basic Knowledge of Relational Databases

A database stores collected information. It is important that the database not only serves as a repository of data but also includes information about the relationships between that data [12]. The term *metadata* describes the data that holds information about other data [13]. A database includes a collection of data, along with its metadata, and is managed by a *Database Management System* (DBMS) [14]. There are various data models to store information, including object-oriented data models, relational data models, and many others. The data model forms the logical level of a DBMS [15]. A DBMS acts as the interface between a user's request and the physical data storage [12]. Because this thesis focuses on relational databases, we explain their characteristic in greater detail [15].

In the relational database model, a database is represented as a collection of tables, which data scientists call *relations*. Every relation has a distinct name. The relations table header specifies the number, names, and types of the columns. Experts refer to it as the *relation schema* [16]. The headings of the columns in the table represent *attributes* [12]. Graphical representations display the attribute names, while they often omit the additional specifications such as the data type. Data scientists refer to each row in the table as *tuple* or *record* [17]. In the following, we will explore two important aspects of relational databases: Section 2.1.1 covers different constraints, while Section 2.1.2 focuses on anomalies and normalization.

2.1.1 Constraints

Because there are no direct references to data objects in a relational database, users must manage their connections differently. This is achieved through integrity constraints. One significant type of constraint is the *key constraint*, which specifies that the attribute values in one or more columns uniquely identify the stored tuples [16]. Attribute sets that satisfy this condition are called *super keys*. When we add the requirement that a key must contain the minimal possible set of attributes, we derive a *candidate key* [15]. A relation can have multiple candidate keys, but only one is selected as the *primary key* [16]. This selection is essential for establishing references between tuples in different relations within

the relational model [15]. The key values of a table can serve as unique references in another table or even in the same one. Experts refer to such references as *foreign keys* [16]. A constraint known as *referential integrity* is applied to manage these key references. It states that any foreign key value that is not null must correspond to a valid primary key value. This is one of the most important constraints because it guarantees the integrity of cross-references between tables [12].

Another important constraint that we need to mention is the concept of a *Functional Dependency* (FD), which generalizes the idea of key dependency [18]. FDs are part of the most commonly used dependencies between attributes. Given two sets of attributes X and Y in a relation r, an FD exists if the attribute values of X uniquely determine the attribute values of Y in each tuple. In other words, if the X-values of two tuples are identical, all their Y-values must also be identical. This relationship is formally expressed in Equation 1, where $t_i(X)$ denotes the values of the attribute set X in the tuple t_i of relation r [16].

$$\forall t_1, t_2 \in r : t_1(X) = t_2(X) \implies t_1(Y) = t_2(Y) \tag{1}$$

We denote the FD as $X \rightarrow Y$ [16]. This means that the X-values functionally determine the Y-values. In other words, the Y-values are functionally dependent on the X-values [15]. An FD is trivial if the right-hand side of the FD is a subset of the left-hand side, such as $X \rightarrow X$. FDs that do not satisfy this condition are non-trivial [16]. A key is a special case of an FD because it represents an attribute set that functionally determines all other attributes in a relation [14]. FDs can be used to derive a *Transitive Dependency* (TD). A TD implies that if X, Y and Z are attribute sets of a relation and $X \rightarrow Y$ and $Y \rightarrow Z$ are FDs, it follows that X also functionally determines Z [16].

The last important constraint is a *Multivalued Dependency* (MVD), which exists when for each value of attribute X, there is a finite set of values for attribute Y and a finite set of values for attribute Z both associated with X. The attributes Y and Z are independent of each other [12]. Every FD is also an MVD but not vice versa [15].

2.1.2 Anomalies and Normalization

One of the problems that can occur with relational databases is redundancy. Redundancies are problematic because they take up unnecessary storage space. Their bigger problem, however, is that they make it difficult to modify the underlying relations. If a piece of information is redundant, the user must adjust it in all places where it occurs. Local integrity constraints further complicate this process [16]. Problematic change operations include *update anomalies*, *insertion anomalies*, and *deletion anomalies* [18]. If a user wants to change a redundant entry in the database, then they need to change all of the corresponding entries at the same time. This can result in an update anomaly if the user

does not change all occurrences simultaneously [15]. An insert anomaly occurs when a user tries to insert a tuple that does not have a value for each attribute of the key. Missing values outside the key can be filled with NULL-values if these are allowed [18]. The deletion anomaly occurs when a user deletes a specific piece of information and this causes the unintentional loss of other pieces of data. This means that information that should have been kept is lost because it was associated with the information that was deleted [15].

A particularly large number of redundancies occur when relational databases consist of a small number of very large tables. This can cause the described anomalies to occur more frequently. *Normalization* can be used to avoid these problems and make the database efficient and reliable [17]. The goal of normalization is the reduction of redundant information by decomposing relational schemas. We consider a relational schema R that we decompose into the schemas $R_1, ..., R_n$. The schemas $R_1, ..., R_n$ each contain a subset of the attributes of R, which means that $R_i \subseteq R$ is valid for $1 \leq i \leq n$ [15].

There are two important aspects to consider in the decomposition process: *dependency preservation* and *lossless join decomposition*. Dependency preservation states that the functional dependencies that apply to R must be transferable to the schemas $R_1, ..., R_n$ [15]. More specifically, the set of dependencies of R must be equivalent to the set of key constraints in the decomposed database schema [16]. Lossless join decomposition ensures that a user can reconstruct the original relation by applying a *natural join* to the decomposed relations. The natural join connects relational tables through columns with the same name and identical attribute values [16]. The information represented by the original relation schema R must be fully recoverable from the relation schemas $R_1, \ldots, R_n$ using natural joins [15].

Normalization can reduce all dependencies to key dependencies. To achieve this, an algorithm or expert must decompose all relations that do not meet this criterion. For this process, experts define specific *normal forms*, which are theoretical rules that a relation must satisfy [14]. The rules become progressively stricter as the level of normalization increases [16]. The most well-known are the first to fourth normal forms. Although other normal forms exist, they are less commonly used and are not relevant to this thesis [16]. Figure 1 shows the five nested normal forms. According to the nesting, a relation in a higher normal form automatically conforms to the rules of the lower forms [12].

A table is in *First Normal Form* (1NF) if all of its attribute values represent atomic data values. They must also not contain any hidden structures within a string, such as lists, enumerations, or vectors separated by commas [16]. It is mandatory that each column contains only one value for each row in the table [17]. Relations in 1NF that do not additionally satisfy the rules of stricter normal forms can contain all of the anomaly types mentioned. Therefore, a stricter normal form is necessary if you want to avoid the anomalies [12].

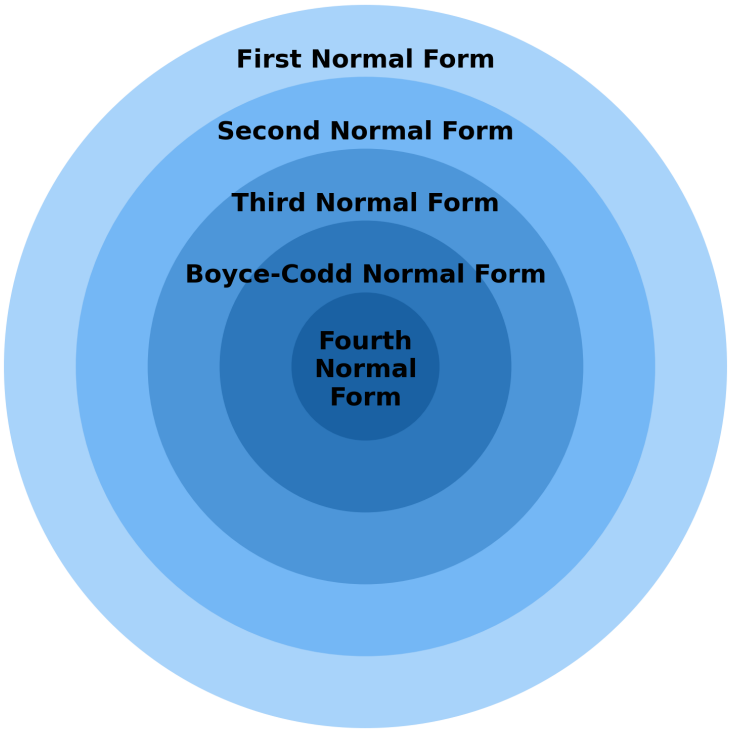

Figure 1: Visualization of the five nested normal forms of relations [12].

A relation is in *Second Normal Form* (2NF) if it satisfies the requirements of 1NF and additionally fulfills the following rule [17]. All attributes that are not part of the key must be functionally dependent on the entire primary key. Dependencies on subsets of the key are not sufficient. Achieving 2NF resolves issues within relations. However, anomalies can still occur, which is why the application of a more advanced normal form is necessary[12].

A relation satisfies the *Third Normal Form* (3NF) if it is in 2NF and contains no TDs. For many relations, 3NF represents a strong design choice because it typically eliminates most anomalies. However, certain unique characteristics may still result in anomalies. To address these cases, even stricter normal forms are available [12].

The *Boyce-Codd Normal Form* (BCNF) is a stricter version of 3NF. It requires that for every non-trivial FD $X \rightarrow A$, X must be a superkey [17]. This ensures that every attribute of the relation is fully dependent on every key [18]. Despite the advantage that BCNF further eliminates anomalies, it must be noted that BCNF also leads to disadvantages. BCNF and dependency preservation are generally incompatible and it is not always possible to achieve them simultaneously for a relation. A BCNF-decomposed relation cannot guarantee dependency preservation because not all functional dependencies remain directly enforceable. Instead, it may require joins to verify certain dependencies [16]. Similar to BCNF, the *Fourth Normal Form* (4NF) is intended for relations with rarely occurring characteristics. It is fulfilled if the relation is in BCNF and if the relation contains no MVDs [12].

In the following, we focus on BCNF because it serves as the normalization form for SYDAG. There is a decomposition algorithm for transforming a relation into BCNF. When fully executed, this algorithm produces BCNF-compliant relations, which preserve lossless join decomposition [18]. The goal is to decompose the relation R into the set $Z = \{R_1, ..., R_n\}$. We start the algorithm with $Z = \{R\}$. As long as there is a relation $R_i \in Z$ that is not in BCNF, we apply the following steps:

1. Identify a non-trivial FD $(X \rightarrow Y)$ valid for R_i, such that:
   (a) X and Y have no attributes in common, i.e., $X \cap Y = \emptyset$.
   (b) X is not a superkey of R_i, i.e., $X \not\rightarrow R_i$.
2. Decompose R_i into the relations:

$$R_{i_1} = X \cup Y \quad \text{and} \quad R_{i_2} = R_i - Y.$$

3. Remove R_i from Z and insert R_{i_1} and R_{i_2}, i.e.:

$$Z := (Z - \{R_i\}) \cup \{R_{i_1}\} \cup \{R_{i_2}\}. \quad [15]$$

Figure 2 shows the decomposition of the relation schema. It visualizes how the algorithm decomposes the relation R_i into the relations R_{i_1} and R_{i_2} using the FD $X \rightarrow Y$ [15].

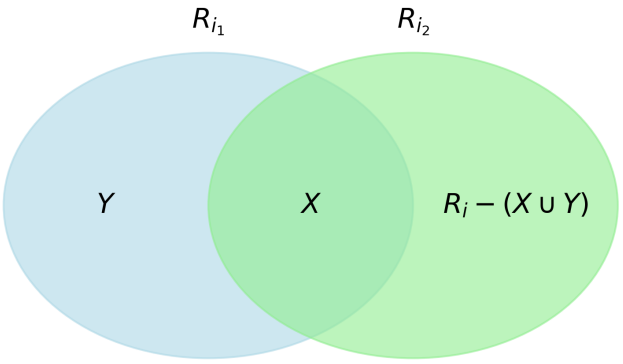

Figure 2: Illustration of the decomposition of a relation into two new relations during the transformation to BCNF [15].

When working with databases, there is also the possibility of using *denormalization* instead of normalization. This involves reintroducing redundant data to enhance performance [14]. A denormalized schema is the reverse of a normalized schema. It does not conform to the rules of normal forms, which results in duplicated data entries. Besides the normalized and denormalized schema, other schema types exist, such as the star schema and the snowflake schema [19]. These will not be explained here because SYDAG is designed to work with normalized schemas.

2.2 Introduction to Data Integration

In Section 2.1.2, we discuss the problems caused by redundant data. To avoid data redundancy, scientists can perform data integration [14]. It describes the process of merging data to create databases that are as consistent and as complete as possible. It is particularly useful in scenarios where data is distributed but an analyst requires a unified database [20]. The main challenge in data integration is heterogeneity, which describes the different structures, models, and implementations that can occur in data. Scientists can distinguish different types of heterogeneity, but in reality the types often overlap. The four types are *technical*, *syntactic*, *structural*, and *semantic heterogeneity*. Technical heterogeneity refers to the fact that there are differences between the hardware, software, interfaces, and query languages. It focuses on the differences in data access methods. Syntactic heterogeneity refers to technical differences in the representation of identical facts, such as different character encodings or number formats [13],[20]. Structural heterogeneity occurs when schemas differ despite representing the same real-world entities [13]. This can happen due to different requirements or levels of expertise among developers. An example is the use of different data models. Semantic heterogeneity describes the different meanings and contexts of the data, such as variations in attribute names or scales [20]. In addition to heterogeneity, *source autonomy* complicates data integration. Sources do not always belong to a single administrative unit or may be operated by different sub-organizations. Therefore, it is not always possible to have unrestricted access to a source's data. Integration becomes challenging even with a small number of sources. As the number of sources increases, so does the complexity of integration [21].

In the following, we take a closer look at data integration by describing its fundamental principles. Because the field includes various concepts and techniques, we focus only on the aspects that are essential for this thesis. Section 2.2.1 gives an overview of faulty data. This is followed by an explanation of string matching and similarity measures in Section 2.2.2. Next, Section 2.2.3 summarizes possible metrics for evaluating performance. Finally, we give an explanation of schema matching in Section 2.2.4.

2.2.1 Faulty Data

When scientists examine different data sources and their relationships, they often find errors and conflicts within the data. Experts divide these data errors into two classes. The first class includes errors that exist within individual data sources. These are further divided into data level and schema level. The data level includes various errors. Part of them are spelling errors [13]. These can result from typos during data entry or from *Optical Character Recognition* (OCR) problems. OCR problems describe cases where computer systems misinterpret a letter for a number or vice versa (e.g., 'L0VE' instead of 'LOVE') [8]. Missing and incorrect values are another common issue at the data level. While missing

values are clearly visible, analysts may not notice incorrect values if they cannot verify them against real-world observations [13]. Incorrect alphanumeric values can arise, for instance, from custom abbreviations or variations in formatting conventions [21]. In numerical data, incorrect values may occur as outliers. Data level errors also include embedded values, meaning a developer stores multiple data points in a single field although they should be stored separately (e.g., title = 'Lily Smith Marburg Germany'). Additionally, duplicates and data conflicts are common. These conflicts appear when contradictory entries exist between duplicates (e.g., ID=1, name='Lily' and ID=1, name='Jane') [13].

At the schema level, analysts can observe other types of errors. These include invalid values, which are entries that fall outside the defined domain of an attribute. Datasets can also contain violations of attribute dependencies, meaning that two dependent attributes include conflicting values. Additionally, datasets can include breaches of uniqueness, which indicates that attributes marked as unique contain duplicate values [13]. Lastly, datasets may include violations of constraints. This implies that a data value designated as a foreign key does not appear in the corresponding referenced attribute [20].

The second class contains errors that only appear during the integration process [13]. These errors relate to connections between multiple fields [8]. As with the first class, experts can distinguish between errors at the schema and the data level. Errors from the first class may also appear at the data level of the second class. However, there are additional issues we need to consider. First, inconsistent values can arise. This happens, when data scientists merge data from multiple sources, which include attributes that do not align (e.g., age = 12 but birthday = '1.1.1790'). Different representations can also create problems. They occur when different data sources represent the same fact differently (e.g. room = 'bathroom', room = 3). The use of different units is another common error. An example are the measurements of height (height = '1m' or height = '100cm') [13]. This makes the standardization of names and units an important step in the integration process [20]. Additionally, different levels of detail can cause errors (e.g., length = '1h' vs. length = '1h10min'). The schema level of the second class includes structural and semantic heterogeneity, which we already explained above [13].

Data errors can arise for various reasons. They can be caused by incorrect input, obsolescence, and faulty transformations [13]. It is difficult to completely avoid errors. This is why data scientists need integration algorithms and techniques that can overcome the challenges of incorrect data and recognize corresponding attributes despite errors [20].

2.2.2 String Matching and Similarity Measures

To identify the corresponding schema elements from different data sources, analysts need to determine similarities between the sources' values. These similarities help them identify whether the values correspond to the same attribute [20]. To assess similarities between values, analysts can employ *string matching*. It describes the process of identifying

strings that relate to the same real-world entity [21]. Suppose there are two sets of strings X and Y. The goal is to identify all matching pairs (x, y) where $x \in X$ and $y \in Y$. A match exists if x and y relate to the same real-world entity. To address this, analysts can define a *similarity measure*, which evaluates two strings x and y and returns a score between 0 and 1. This score indicates whether the strings are a match. For this purpose, analysts can establish a threshold value t. If the measured similarity satisfies or exceeds this threshold, it means that the strings are a match. More specific, if similarity s satisfies $s(x, y) \geq t$, then x and y are a match [21]. To evaluate the similarity between sets of strings, analysts can define a function to aggregate individual similarities into an overall measure. This function should return a high value if the sets describe the same real-world entity [13].

There is a wide variety of similarity measures [21]. Each of them is highly dependent on the specific application. It is difficult to formulate general rules for numeric values, which is why separate measures exist for attributes like age, monetary amounts, and others. For strings, however, more general measures are available [13]. In the evaluation of SYDAG, we use two similarity measures for strings: a sequence-based and a set-based similarity measure. Sequence-based similarity measures treat strings as a sequence of characters and calculate the cost of converting one string into another [21]. A common form of this measure is the *Levenshtein distance* [13]. It determines the number of operations needed to convert the first string into the second [22]. Possible editing operations are deleting a character, inserting a character, or replacing one character with another [13]. Developers can implement the Levenshtein distance $d(x, y)$ by using dynamic programming. Let x be of length n and y of length m, with $x = x_1 x_2 \dots x_n$ and $y = y_1 y_2 \dots y_m$. Here, x_i represents the character of string x at position $i \in [1, n]$ and y_j represents the character of string y at position $j \in [1, m]$. Let $d(i, j)$ be the Levenshtein distance computed between $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$. It is captured in Equation 2 [21]. Using the dynamic programming approach, an algorithm can derive the value $d(i, j)$ from the existing values $d(i-1, j)$, $d(i, j-1)$ and $d(i-1, j-1)$. Inserting, deleting, or substituting a character has a cost of 1. Therefore, developers calculate the cost of the deletion of x_i as $d(i-1, j)+1$. For the insertion of y_j, it is $d(i, j-1)+1$ [21]. In the case of a substitution, the result is $d(i-1, j-1)+1$. If $x_i = y_i$, then the added costs are 0 because the characters match. This results in a cost of $d(i-1, j-1)$ in case of a match [13]. The algorithm defines the boundary values as $d(i, 0) = i$ and $d(0, j) = j$ [22]. Using a bottom-up approach, developers can dynamically compute the values starting from the boundary values [13].

$$d(i, j) = \min \begin{cases} d(i-1, j-1) \\ d(i-1, j-1)+1 \\ d(i-1, j)+1 \\ d(i, j-1)+1 \end{cases} \tag{2}$$

The Levenshtein distance is useful for the detection of editing errors, such as swapped letters. To utilize it as a similarity measure, analysts must convert it into a similarity function. This is shown in Equation 3 [21]. The equation normalizes the Levenshtein distance to the lengths of the longer string and then subtracts it from 1. Consequently, it returns $s(x, y) = 1$ for identical strings and $s(x, y) = 0$ for completely different strings [13].

$$s(x, y) = 1 - \frac{d(x, y)}{max(|x|, |y|)} \tag{3}$$

There are scenarios where the Levenshtein distance is less suitable because it is sensitive to sequence inversions. This can be problematic for multi-word strings. Strings can be very similar even if someone rearranges their words, which the Levenshtein distance does not take into account [13]. In this case, set-based similarity measures are a better choice [21]. Unlike sequence-based similarities, which treat strings as sequences of characters, set-based similarity measures view strings as sets of tokens [21]. First, an algorithm breaks down the strings into tokens. Then, it determines which and how many tokens the strings have in common. There are various ways to decompose strings into tokens [13]. In *character-level tokenization*, the algorithm identifies each character as a separate token. In contrast, *word-level tokenization* uses entire words as tokens [23]. The algorithm can employ spaces, punctuation marks, hyphens, or other special characters as separators for word-level tokenization. However, this approach can be error-prone because such characters frequently appear in proper names [13]. *Subword-level tokenization* provides a balance between character-level and word-level approaches by dividing words into small units [23]. This process is called *n-gram* generation because it generates substrings of length n. To ensure the equal importance of the letters, such algorithms add special characters at the beginning and end of a word [13]. A set of 3-grams for the string 'home' may look like this: {##h, #ho, hom, ome, me#, e##} [21].

Another widely used similarity measure is the token-based *Jaccard measure* [13]. Let B_x and B_y represent the sets of tokens from the strings x and y. Equation 4 defines the Jaccard measure on these token sets [21]. It is the size of the intersection of the two sets divided by the size of their union. The measure equals $J(x, y) = 1$ in case of $|B_x \cup B_y| = 0$ [24]. Because an integration tool generates the token sets first and independently of the Jaccard measure, analysts can choose which tokenization method to apply. They can use the Jaccard measure with any tokenization approach [13]. To evaluate whether two strings match, analysts can apply the two similarity measures we described. Additional metrics are also relevant for this procedure and we address them in the next section [21].

$$J(x, y) = \frac{|B_x \cap B_y|}{|B_x \cup B_y|} \tag{4}$$

2.2.3 Performance Metrics

Before analysts can determine whether two strings are a match, they must define the threshold. The decision on a threshold depends on the individual integration problem [21]. In the case of SYDAG, we use several statistical concepts to define our threshold. The first one is the *arithmetic mean*, which represents the average of the values $x_1, x_2, \ldots, x_n$ and is shown in Equation 5 [25]. Additionally, we need to consider the *empirical variance* and *standard deviation* of a distribution. These are the most well-known measures of the dispersion of a distribution. Variance is shown in Equation 6 and standard deviation in Equation 7 [25], [26].

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{5}$$

$$s^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{6}$$

$$s = +\sqrt{s^2} \tag{7}$$

Once analysts have set the threshold for a similarity measure, they can start to classify the matches [21]. They can identify true matches but false matches may also occur. There are four possible scenarios [13]:

1. *True positives*: Two strings representing the same real-world entity are correctly identified as a match.

2. *False positives*: Two strings are classified as a match, although they do not represent the same real-world entity.

3. *True negatives*: Two strings that do not represent the same real-world entity are correctly not considered a match.

4. *False negatives*: Two strings representing the same real-world entity are incorrectly not considered a match.

Analysts can use this classification to evaluate the effectiveness of similarity measures. For this purpose, they can apply different performance metrics [13]. *Precision* is the ratio of true positives to the sum of true positives and false positives, as shown in Equation 8 [13]. Analysts can achieve high precision by using a high threshold, which only detects a match when two strings are very similar. However, this can cause that they miss some actual matches, whose similarity scores are below the threshold [13]. To address this, they can use *recall* as an additional measure. It is the number of true positives divided by the sum of true positives and false negatives, as shown in Equation 9 [27]. A high recall indicates that most of the actual matches are found. Analysts can often achieve a high recall

by using a more tolerant similarity measure. This leads to a trade-off between precision and recall, which they must balance depending on the application. They can use the *F-measure*, which is the harmonic mean of precision and recall, shown in Equation 10 [13].

$$\text{precision} = \frac{|\text{true-positives}|}{|\text{true-positives}| + |\text{false-positives}|} \tag{8}$$

$$\text{recall} = \frac{|\text{true-positives}|}{|\text{true-positives}| + |\text{false-negatives}|} \tag{9}$$

$$\text{F-measure} = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \tag{10}$$

2.2.4 Schema Matching and Mapping

The string matching techniques we describe in Section 2.2.2 form the basis for *schema matching* and *mapping* [21]. These are two sequential steps within the data integration process [13]. Schema matching aims to identify correspondences between two schemas [20]. A correspondence describes a match between attributes of the schemas. In a 1:1-correspondence, one attribute in the source schema is directly linked to one attribute in the target schema. There are also complex multi-valued correspondences involving multiple attributes from the source, target or both schemas [13]. Schema matching faces several challenges that complicate the identification of correspondences [21]. These include heterogeneity, size, and complexity of schemas as well as foreign language schemas [13].

Matchers are algorithms or combinations of algorithms that discover correspondences [20]. A matcher can take two schemas as input. Depending on its configuration, it may also receive additional information, such as data instances. The output of the matcher is a similarity matrix that includes a number between 0 and 1 for each pair of attributes. Analysts identify matches based on the entries in the similarity matrix. The higher the value, the greater the confidence of the matcher that the attribute pairs are a match. A straightforward approach is to define a threshold [21]. Analysts consider an attribute pair a match if its similarity score meets or exceeds this threshold. If the goal is to achieve accurate results, it is crucial to choose an appropriate threshold [13].

Experts group matchers into two categories: *schema-based* and *instance-based matchers*. The difference is that schema-based matchers assess the similarity of attributes across different relations based on the schema structure. In contrast, instance-based matchers rely on the actual values of the relations to determine similarity [20]. Schema-based matchers compare attribute names, which requires the use of an effective similarity measure. Among others, the Levenshtein distance can be used. For instance-based matchers, one possible approach is to apply tokenization and the Jaccard measure [21].

Matchers do not have to rely on the linguistic or semantic content of the data. They can also use statistical methods [20]. An important matcher for evaluating SYDAG is the

Distinct-Count Matcher. The number of distinct elements in a dataset offers information about the similarity of attributes. The closer the counts of distinct elements in the respective attribute columns, the higher the similarity value [13]. Analysts can effectively determine the number of distinct elements of a column using various methods [28]. Additionally, there are combined matchers. They can either be hybrid, meaning they combine several similarity measures into a single one, or they are composite. This means that they combine the results of the separate matching methods in the final step [13].

Analysts usually identify matches by using a schema matching system [21]. However, these matches serve only as suggestions and an expert with the appropriate domain knowledge must verify them. Once this process is complete, analysts can perform schema mapping [20]. The goal of schema mapping is to transform the matches into a mapping. The challenge is to figure out how to translate tuples from one schema to the other. There can be more than one way to join the data. Therefore, the process must define the tabular organization of the source and target data by specific joins and unions [21].

Schema matching and mapping are essential steps in the data integration process [20]. We want to position them within the context of the entire data integration workflow. Therefore, we outline the typical sequence of steps. First, analysts extract the data to be integrated. Then, they must transform the datasets into a unified model [17]. After that, they can profile the data to gather characteristics about it [20]. The next step involves the application of schema mapping and matching. After that, analysts apply schema integration to generate a unified schema [21]. Following this, they employ entity resolution to identify and merge identical entities from different data sources [29]. Finally, they optimize the data [21]. We discussed the steps of the data integration process that are relevant to this thesis. Further details about the remaining steps are described by Doan et al. [21].

2.3 Related Work

To evaluate the performance of integration tools, benchmarks are essential [30]. A practical approach is to construct these benchmarks from real-world data. This enables developers to test schema matching tools in realistic conditions [31]. An example of this approach is the benchmark Thalia provided by Hammer et al. [32]. Thalia is a collection of over 25 publicly available test datasets, which include a large number of syntactic and semantic heterogeneities [32]. Cabrera et al. present a similar benchmark, called DIBS. DIBS includes real data integration tasks from diverse domains and additionally provides metrics to assess performance in these scenarios [30]. Crescenzi et al. introduce another benchmark, called Alaska [6]. 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. Users can select subsets of the data to achieve the desired level of difficulty. Experts manually created the ground truth [6].

It is 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]. The goal is to generate realistic datasets that reflect heterogeneity and volume [2].

Panse et al. introduce a test dataset generator called DaPo$^+$ [2]. It is an extension of the generator DaPo presented by Hildebrandt et al. [33]. Panse et al. considered DaPo with a new approach and further developed it to create DaPo$^+$ [2], [7]. It receives a dataset 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 [2].

Another generator for schema matching scenarios is EMBench++, presented by Ioannou et al. [34]. 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 dynamic and realistic scenarios [34].

Lee et al. present eTuner [35]. It is a system that automatically optimizes schema matching systems by adjusting parameters for tuning components. It includes a workload generator that creates new schemas from a given one by splitting the schema into disjoint sets and introducing errors in column names and data. It outputs the new schemas and their mappings. Due to the modifications, it can generate realistic matching challenges. However, this functionality is not used individually. The generator directly passes the results to the staged tuner, which then optimizes the underlying matching system [35].

Koutras et al. introduce Valentine, an open-source experiment suite for evaluating schema matching techniques in dataset discovery [36]. It includes a dataset generator that accepts a tabular dataset and a user-defined configuration as input. Valentine performs a horizontal and/or vertical split on relations while retaining some overlapping columns or rows. It also inserts errors into the overlapping entries at both the data and schema levels [36]. In addition, 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. Alexe et al. introduce STBenchmark, which includes two generators for mapping and instance scenarios [37]. 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 [37]. The same applies to iBench, presented by Arocena et al. [38]. It is a metadata generator for the synthetic creation of schemas and mappings. It requires

only a user-defined configuration rather than a real data source. There are flexible control options over the size and complexity of the generated data. Unlike the other generators, iBench independently creates constraints and other metadata [38].

The presented examples show that various data generators with distinct focuses exist. In the development of SYDAG, we take their different characteristics into account. We filter, combine, and extend them to form a particularly adaptable data generator, which allows the user to make exact configurations and create individual integration scenarios.

3 Design

There are several aspects that we must consider for the creation of a dataset generator. The aim of this chapter is to provide an overview of the development steps that we perform to design SYDAG. To achieve this, Section 3.1 first discusses the approaches and methods that we can use to create a dataset generator. Section 3.2 then explains the user's configuration options and highlights the benefits that they offer. Finally, in Section 3.3 we present the architecture of SYDAG and explain the sequence and roles of its components.

3.1 Approaches and Methods for Generation of Datasets

When we design our generator, we are faced with a wide selection of approaches and methods. Our challenge is to select the most suitable options that enable a useful dataset generation [34]. This challenge brings us to the first research question. By answering this question, we identify the necessary components required for SYDAG's generation process.

First Research Question:

What approaches and methods are suitable components for the generation of synthetic datasets that represent data integration scenarios?

When we analyze which components are useful for generating datasets, it is essential to examine the functionalities of the generators discussed in Section 2.3. First, the developers must define the required input for the generator. They need to decide whether the generator receives a dataset as input or generates the schema and instances itself, as is the case with iBench and STBenchmark [37],[38]. If the developer requires the user to provide an input dataset, that dataset can directly serve as ground truth. This allows users to test an integration tool by checking if it can reconstruct the structures of the input dataset [36]. A generator may also require a user-defined configuration as input. This is necessary if it should be possible to tailor the generation to specific expectations. For example, the configuration of DaPo$^+$ allows the user to specify the degree of data errors [2]. The more configuration options the developer provides, the more precisely the user can customize the integration scenario [38].

Another important consideration is the selection of supported data models. The developer must specify the models because the modifications that the generator can apply to the datasets depend on them. In the case of SYDAG, we choose to support relational datasets because they are one of the most common formats for structuring data [11]. If a user requires non-relational datasets, DaPo$^+$ provides a suitable alternative [2].

After a dataset is provided, a generator can perform one or more splits as the first processing step. The process divides the existing relations and generates new relations from them [38]. While eTuner exclusively performs a horizontal split without overlap, Valentine offers more options [35],[36]. It supports a horizontal split, a vertical split, or a combination of both. Additionally, Valentine's users can specify the degree of overlap for rows and columns [36]. The fewer data tuples overlap, the more complex the matching process becomes [35]. Another feature is the selection whether Valentine should choose the overlapping rows randomly or as a contiguous block from the start of the relation [36].

Errors in the schema play another important role if users want to add complexity to the integration scenario [2]. We observe several approaches in the generators. Valentine offers options for perturbing the column names. These include prefixing the column names with the table name, abbreviating the column names, and dropping the vowels [36]. ETuner adds more options. These include replacing words with synonyms and changing the column name to random characters [35]. If a relation includes these errors, it causes discrepancies with the names of the overlapping columns from other relations. This increases the complexity of the integration [36]. The configuration may also allow the user to apply no changes, which results in a less challenging integration scenario [35],[36].

In order to generate complex scenarios, generators should not only add errors into the schema but also into the data. Therefore, another important component of dataset generation is the perturbation of the data values [34]. Valentine provides two methods for this. First, it can introduce random typing errors based on keyboard probabilities. Second, it can randomly change numeric values depending on their distribution [36]. ETuner offers additional methods, such as changing the data format and adding decimal places [35]. Other methods are available in EMBench++. These include word permutations, missing values, and abbreviations. In the configuration of EMBench++, the user can specify a percentage of values receiving errors [34]. Valentine offers an additional control option. The user can specify how many characters within a single value it should modify. We refer to the errors introduced into the data as *noise* [36].

It is important to note that split and noise components depend on knowing the keys of the relations because the keys must be part of all newly formed relations [36]. To address this, Valentine allows users to specify which columns represent the primary key [36]. Integration tools are extremely challenged when they are required to restore the original structure of datasets that contain relations with noisy key values [37]. It is therefore important to determine the keys before a generator splits and inserts errors to a dataset [36].

Another component we identify is structural changes of the relation via joins. These changes are implemented in two of the discussed generators. STBenchmark supports integration scenarios with denormalization, meaning it can join overlapping relations into a single relation [37]. Similarly, eTuner can change the structure of relations by merging two columns within the same relation, if they are neighboring columns or share a prefix [35]. Such joins can create a structure that differs from the original dataset [37].

Finally, we need to consider the output of the generator. If the input dataset is already used as the ground truth, it is sufficient that a generator simply outputs the newly generated datasets [36]. There are also matchers that generate and output precise mappings for the generated datasets, such as iBench and eTuner. Developers can use these mappings to evaluate their integration mechanisms [35],[38].

In addition to the established functionalities of existing generators, it is also necessary that we explore other relevant aspects that can enhance the capabilities of a generator. One important requirement for integration scenarios is schema complexity. The datasets created by generators are often simply structured, although many real datasets have more complex connections between the data [2]. Generators should accommodate this complexity. One way to change the structure of relations is to apply normalization [12]. Normalization can create distinct structures among the relations within the datasets to be integrated. It also introduces more foreign key constraints, which increases the complexity and challenges of the integration process [20].

Another way to change the structure is to shuffle rows or columns. This means that relations representing the same real-world entity no longer maintain the same order of entries. While the information remains unchanged, its order is modified [39]. This can be a significant challenge for matchers that rely on the order of attributes or tuples [13].

We summarize the results of the analysis with regard to the first research question. We can identify a total of eight approaches and methods as suitable components. These include input handling, splitting, joining, normalization, noise insertion in the schema, noise insertion in the data, shuffling, and generating the output. Due to the scope and the different requirements of the data, it is not feasible for us to incorporate all the specific methods of the generators into the design of SYDAG. Instead, our goal is to implement the core components and to choose a useful amount of functionalities. These can include ideas from the other generators as well as self-developed options.

3.2 Input, Output and Configuration Parameters

It is important that we determine the exact functionalities that the individual components of SYDAG provide. To achieve this, we define the input, output, and configuration parameters that the user can pass. This enables us to derive a logical order for the components in the following Section 3.3.

As mentioned above, SYDAG works with relational datasets. The user must select a relation stored in a CSV-file as input. Additionally, they must specify the properties of the relation so that the generator can read it correctly. This includes whether the relation has headers and which characters are used as separator, quote, and escape sequence.

Regarding split types, SYDAG offers a horizontal and a vertical option, allowing the user to choose either one or both. In addition, the user can specify separate percentages for overlapping columns and rows. The configuration also includes the distribution of non-overlapping columns or rows between the new relations, enabling one relation to have more columns than the other. For horizontal splits, the user can further choose whether the overlapping rows should be selected within a block or randomly scattered.

The user can define how SYDAG modifies the structure of the created datasets by choosing one of three options for each dataset. The first option keeps the structure unchanged. The second allows column joining and is inspired by eTuner [35]. If the user selects this option, they must specify the percentage of overlapping columns to join. The third option applies normalization to BCNF and requires the user to specify a percentage that determines how many of the possible decomposition steps should be executed.

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, they must specify the percentage of how many of the attributes overlapping with other datasets the generator will modify. The user can select from nine error methods to introduce the noise. Additionally, they can decide to delete the schema completely, which results in a relation without headers.

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 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. A shuffle option is also available for each created dataset, allowing the user to choose between no changes, shuffling rows, or shuffling columns.

After processing, SYDAG outputs each relation of the created datasets as a CSV-file. If the user chooses only one split option, the generator will create two datasets. Selecting both split options results in four datasets. Depending on the structure changes chosen by the user, these datasets include different numbers of relations. In addition, for each dataset, SYDAG outputs a file containing the key and foreign key relationships. Lastly, the generator creates a mapping file that specifies which attributes in the new relations correspond to the original attributes, meaning they represent the same real-world entity.

Regarding the presented generators, SYDAG shares the greatest similarity with Valentine's dataset generator. Both are based on relational datasets in the form of CSV-files [9]. SYDAG incorporates some of Valentine's functionalities but enhances them by adding structure and error options. Therefore, it is essential that we outline the unique features of SYDAG compared to Valentine. SYDAG automatically handles the determination of keys, while Valentine requires the user to specify the keys. If they are unknown, a separate approach must first determine the keys before Valentine can be used [36]. This makes SYDAG more practical for users who do not know the keys of their datasets. There are also notable differences in the methods available for introducing errors into the schema. SYDAG adds six additional options to the three methods that Valentine provides. SYDAG also allows the user to control how many attributes in the schema should receive noise, while Valentine only allows the user to choose whether or not errors are included [36]. Differences also exist in how the generators introduce errors into the data. Valentine offers two methods for perturbing the entries: one for numeric and one for alphanumeric entries [36]. In contrast, SYDAG offers a selection of twelve methods for alphanumeric entries and two for numeric entries. The user can freely combine these methods, which allows the inclusion of simple or complex errors. Completely new parts of SYDAG are the join, normalization, and shuffle components, which are not part of Valentine [9].

3.3 Architecture of SYDAG

After we determined the useful components and their configuration options, the next step is to effectively combine them to create SYDAG. We require a logical order and implementation of the components. This challenge leads us to the second research question.

Second Research Question:

How can we effectively integrate the various generation components into a single generator for creating data integration scenarios?

When we arrange the components of SYDAG, it is important that we consider how they logically build upon one another. We need to perform certain processing steps before others to allow optimal dataset generation. In addition to input and output, we arrange the other components developed in 3.1 in a logical and effective order. Figure 3 visualizes the sequence of SYDAG's generation steps. We can precisely justify the choice of this sequence. We select the split as the first processing step. It serves as the basis for the generation because it determines the number of new datasets that SYDAG creates. SYDAG splits the relation passed by the user into either two or four new relations, each representing a distinct dataset of the integration scenario. Through normalization, SYDAG can further decompose these relations into subsequent relations, causing the datasets to no longer

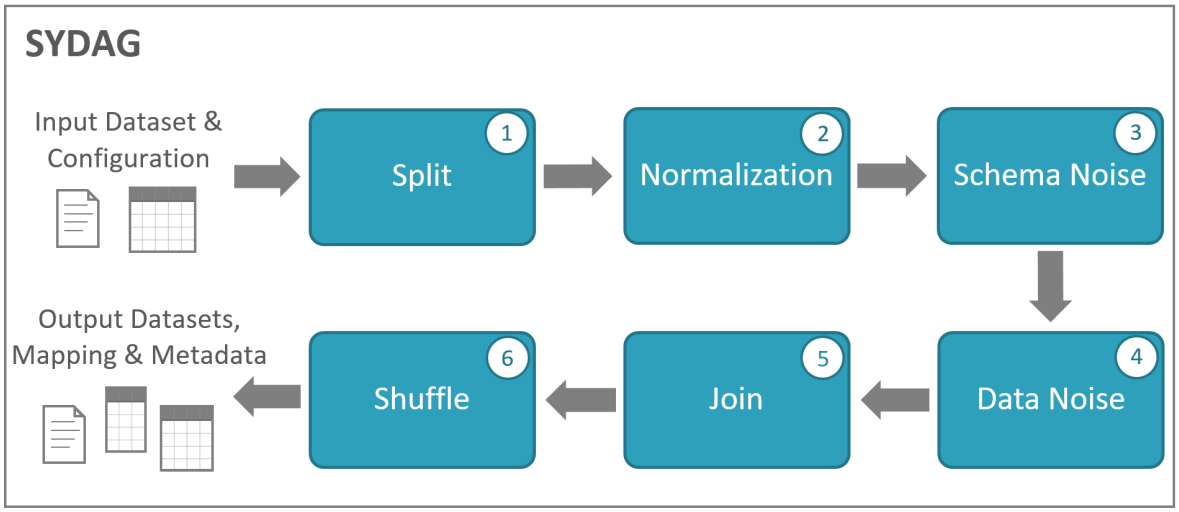

Figure 3: Overall procedure of SYDAG: An input dataset and a user-defined configuration are received. After that, six steps are executed sequentially. The created files are output in the final step.

contain only one but multiple linked relations. However, it is important to note that immediately after the split, each dataset contains only one relation. By placing the split at the start, it is possible to select the error levels and structural changes individually for each newly created dataset. This is particularly useful if the user wants to create heterogeneous datasets. Furthermore, the split component includes key determination, which is helpful because the following components require knowing the keys of each dataset.

Normalization follows as the next component. It can be used to create more relations within the datasets. It is logical to execute this component after the split and before introducing noise because the decomposition process generates new foreign key constraints. SYDAG stores and passes these constraints to the noise components. When SYDAG adds noise, the user can choose to preserve the constraints or allow them to be broken. Therefore, SYDAG should execute normalization before adding noise.

The following component is schema noise. It introduces errors into the relations based on the user's configuration. Because normalization has already been applied, any schema within a dataset can receive errors. The split, which SYDAG performs in a previous component, provides 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 the possible candidates for noise introduction. The information from the split makes it possible to detect those attributes. Whether we add noise to the schema or the data first is irrelevant because these processes are independent of each other.

Data noise serves as the next component. The same arguments as for the schema noise apply here. Once again, SYDAG can perturb all relations of the datasets, because normalization has already been applied. Additionally, SYDAG can use the information from the split to identify overlapping rows or columns and insert errors into their entries depending on the specified percentage.

Next follows the join component. In our case, a join combines two columns of a relation into a single column within that relation. SYDAG can execute multiple of these joins. We visualize an example of the process in Figure 4. Each join creates an attribute that contains an enumeration of multiple values. It combines the column names and individually concatenates all corresponding entry pairs to single values. Because we want to enable different errors within the linked values, we place the join component after noise insertion. This approach also makes it easier to handle joins of numeric and alphanumeric values. If we performed the joins first, SYDAG would no longer recognize the numeric value as such and, therefore, only apply alphanumeric error methods.

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

Join Columns →

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

Figure 4: Example of the result of the join component for two columns of a relation.

As the last step before the output, we apply the shuffle component. SYDAG can shuffle the row or column order. Placing this at the end is particularly helpful for the implementation strategy because it helps prevent the column indexing from getting disrupted. In addition, this component can break up the block structure of overlapping rows by shuffling all rows. If the user selects the block overlap, the shuffling allows the dispersion of the overlapping rows throughout the entire dataset. These rows originally belong to one block and may share some similarities, but when SYDAG shuffles them it makes it harder to find them. 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 shuffling, SYDAG outputs the metadata, mapping, and new datasets.

With regard to the second research question, we conclude that an effective sequence of components exists and is outlined in Figure 3. This component sequence is: Input, Split, Normalization, Schema Noise, Data Noise, Join, Shuffle, and Output. We implement these components to create SYDAG. The chosen component sequence ensures an implementation that minimizes code redundancies.

4 Implementation

In this section, we implement the design components that we describe in Section 3 to create our SYDAG application. In addition to the implementation of the dataset generation algorithm, we also provide a GUI. Therefore, we divide our code into backend and frontend. In Section 4.1 we present the programming languages and tools that we use to create SYDAG. After that, we describe the implementation of the backend in Section 4.2. Finally, in Section 4.3 we present the frontend and its functionalities.

4.1 Selection of Programming Languages and Tools

For the backend, we choose the programming language Java because it offers a number of advantages. One of them is the object orientation, which enables good structuring of the components of SYDAG. Another advantage is the many Java class libraries, which we can include in our project, for example, for CSV or JSON handling [40]. To process HTTP requests from the frontend in the backend, we use Spring Boot. Spring Boot provides the embedded web server Tomcat, which allows us to receive the user's input from the frontend without having to install an external server [41]. Aside from that, we use two Application Programming Interfaces (APIs) within the Noise component [42]. One of them is the Datamuse API, which we use to generate synonyms for data entries [43]. The other is the MyMemory API, which generates translations of entries [44]. If the user selects the methods that generate synonyms or translations, the generation process of SYDAG can take longer than without this selection. The reason is that calling the APIs is more time-consuming than using a self-implemented error method.

We integrate two algorithms in the form of JAR-files into SYDAG. The first algorithm, called HyUCC, is introduced by Papenbrock [45]. HyUCC efficiently discovers all minimal Unique Column Combinations (UCCs) of a relation. The algorithm begins with a sampling phase in which it identifies non-UCCs by comparing record pairs. It continues with a validation phase, where it uses prefix trees to validate potential UCCs. Due to the combination of approximation and validation techniques, it can handle large datasets. It also outperforms prior methods, including an algorithm called UCC, presented by Heise [46]. We use HyUCC for key determination by selecting one of the smallest UCCs as the key [45]. The second algorithm, called Normalize, is presented by Papenbrock [47]. Normalize transforms relational datasets into BCNF using the decomposition process described in 2.1.2. First, it detects FDs and calculates their extensions. Then it identifies the keys of the extended FDs. After that, it checks which FDs violate BCNF, selects one of them, and decomposes the relation. Normalize repeats these steps until the schema is BCNF-compliant. It works instance-driven and (semi-) automatic [47]. We use it as a base for the normalization component. An advantage is that HyUCC and Normalize are implemented in Java, which allows us to easily integrate them into our implementation [45],[47].

For the frontend, we use TypeScript as the programming language. Its main responsibility is type enforcement [48]. Additionally, we use React to build our application. React is a popular JavaScript library for building reusable functional components [49]. We integrate all of this within the Next.js framework, which provides automatic routing and an extensive feature set that makes it a more viable option for SYDAG compared to other alternatives [50]. While Next.js handles server-side rendering and routing, Node.js executes server-side logic within the frontend architecture [48],[50]. We use Docker with a preconfigured Docker compose file to containerize the entire application of backend and

frontend to ensure simple and consistent deployment [51]. Docker then deploys the frontend and backend services on their respective ports specified in the file. To allow the services to communicate with each other, we need to establish the connection by leveraging Cross-Origin Resource Sharing (CORS), which we implement in the backend [52].

4.2 Backend

Our backend consists of multiple classes that SYDAG utilizes during the generation process. An overview of these classes is shown in Figure 5. We categorize the classes based on their functionality. The API components are responsible for receiving input from the frontend and forwarding it to the generation algorithm. To achieve this, they include the 'Server-Application' class, which manages all HTTP requests sent by the frontend. The 'WebConfig' class configures CORS. Additionally, the 'FormDataWrapper' class maps the received configuration from the frontend to the corresponding backend parameters. The 'GeneratorParameters' class holds the CSV-file provided by the user along with the configuration settings, which we encapsulate in a 'FormDataWrapper' object. The 'FormDataController' processes the frontend's API request. It passes the input relation and the user's configuration as a 'GeneratorParameters' object to the generation algorithm in the 'Generator' class. The 'FormDataController' then generates the results and packages them into a ZIP-file. It sends the ZIP-file to the frontend where the user can download it.

The 'Generator' class contains the generation algorithm, which creates the integration scenario using the provided parameters. For generation, SYDAG uses all the groups that are shown below the 'Generator' class in Figure 5. The Data Structure Components define how our generator stores the datasets. These components include the enumeration 'Type', which determines whether a column is numeric or alphanumeric. To do this, the class contains a method that analyzes all entries in a column. It classifies a column as numeric if at least 75 percent of its entries are numeric. The 'Attribute' class uses this classification by storing a column name and the corresponding 'Type'. To describe a relation and its metadata we use the 'Relation' class. It stores the relations schema as 'Attribute' objects and the data as individual columns. The class also contains the keys of the relation and information about overlapping columns or rows. Finally, the 'Dataset' class stores a list of all 'Relation' objects that belong to the dataset.

Included in the File Processing Components is the 'CSVTool' class. This class provides a method that reads the input CSV-file. The method creates a 'Relation' object from the input file and determines the types of the columns. Additionally, 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

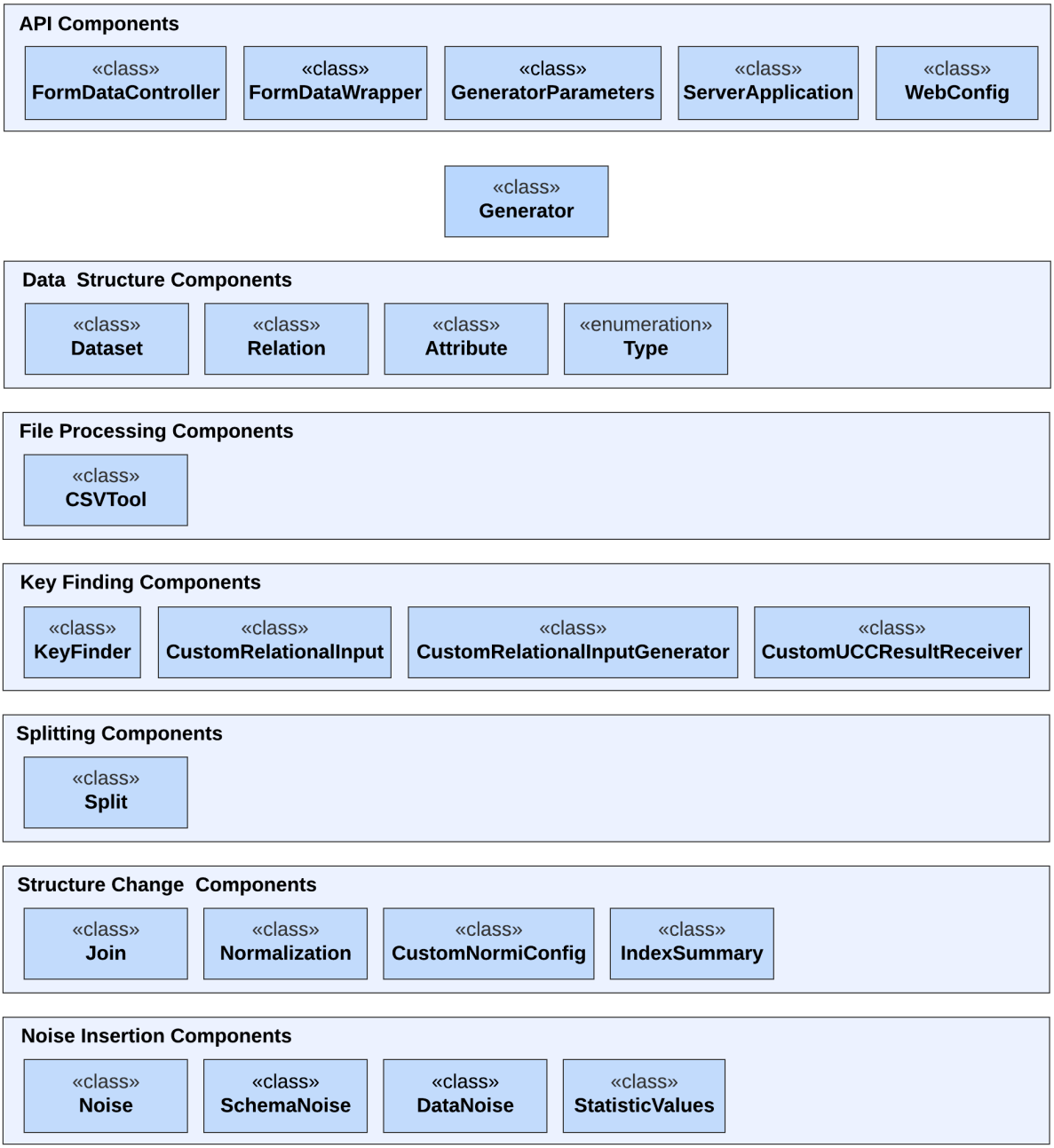

Figure 5: Overview of the components of SYDAG's backend. The classes are grouped by the different processing steps.

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.

SYDAG uses the four classes from the Key Finding Components to identify the keys of a relation. The classes 'CustomRelationalInput', 'CustomReationalInputGenerator', and 'CustomUCCResultReceiver' implement interfaces from HyUCC. The classes define how HyUCC should process a relation and how it should output the results. The 'Keyfinder' class contains a method that takes a relation as input and outputs its key. This method sets up the appropriate input format for HyUCC by using the custom classes. It then executes HyUCC and reads the discovered UCCs from the 'CustomUCCResultReceiver' object [46].

The method chooses one of the combinations with the fewest columns as the key of the relation and outputs the indices of the key columns.

To divide a relation into new ones, SYDAG uses the 'Split' class, which is part of the Splitting Components. For the vertical split, the class provides a method that generates a 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. The method saves the indices of the overlapping columns in a field of the relation. For the horizontal split, there are two different methods. One of them creates a block overlap. First, the method selects a random start index for the overlapping rows. Then, it determines the end index based on the user-selected percentage of overlap. The method 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 method creates a horizontal overlap of random rows. Unlike block overlap, it picks overlapping rows randomly from the entire dataset rather than selecting consecutive rows. To track which rows overlap, both methods save the number of overlapping rows in a field of the relation. It is not necessary to save all indices of the overlapping rows because both methods insert these rows at the top of the new relations. 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.

Among the Structure Change Components are four classes that SYDAG uses to modify the structure of the relations. The 'Join' class provides a method that merges columns within a relation. It relies on the user-defined percentage to determine how many of the overlapping columns should be joined. Then, it calls the method 'executeJoins', which repeats the join process until it reaches the desired number of joined columns. In each step, this method identifies the best column pair for joining 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 joined with another column. The method selects the column pair with the highest score. After that, it merges the attribute names and data entries of the columns into a new column and inserts it into the relation. It also removes the original columns. We use the other classes within the Structure Change Components for normalization. The 'CustomNormiConfig' class implements an interface from Normalize, which defines the input format. The 'Normalization' class provides methods that decompose a relation into BCNF. Its overall method, called 'transformToBCNF', first applies Normalize to find column indices of BCNF-compliant relations [47]. It stores the column and key indices of those relations in 'IndexSummary' objects. Then, the method adjusts the indices to the input relation so that they reference the correct column positions. After that, depending on the user configuration, the

method calculates how many of the decomposition steps that Normalize has detected, it should apply. The final step is the creation of the number of selected relations. To do this, the method extracts the columns corresponding to the indices in the 'IndexSummary' object, removes the overlapping rows, and then outputs the generated relations.

Finally, we explain the Noise Insertion Components. We use the 'StatisticValues' class to store the values of standard deviation and mean of a column, which prevents redundant calculations. The 'Noise' class contains the following error methods:

1. Removal of all vowels from a given string.
2. Abbreviation of the first letter of each word in a string.
3. Shortening of the words in a string to a random length.
4. Shuffling of all letters within a string.
5. Shuffling of all words within a string.
6. Generation of a random string with a length between 1 and 10.
7. Appending a prefix of length 1 to 4 consisting of random characters to a string.
8. Replacing the words in a string with synonyms.
9. Replacing of the words in a string with their German translation.
10. Generation of a missing/ empty value.
11. Insertion of a phonetic error in a string (similar sounding letter).
12. Insertion of an OCR error in a string.
13. Insertion of a random number of typing errors in a string.
14. Changing the format of a string by swapping "-", "_", ".", and spaces.
15. Shortening the words of a string to a random length and adding periods in between.
16. Generation of a random numeric value from the column's normal distribution.
17. Generation of a random numeric outlier for the column.
18. Mapping an entire column to numeric values.

We adopt Valentine's implementation for Methods 1, 3, 7, and 13 with some adjustments, while we develop the other methods independently [36]. SYDAG can apply Methods 1 to 9 to the schema and Methods 5 to 18 to the data. The 'SchemaNoise' class extends the 'Noise' class. We use it to add errors to the attribute names. It includes the 'perturbSchema' 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 randomly selects attributes that will receive noise from the overlapping attributes. For each of the selected attributes, it calls the 'chooseNoise' method, which applies one of the user-selected error methods. It only reuses a method 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 only selected Methods 1 and 5, but neither is applicable, Method 6 or 7 serves as fallback.

The 'Data Noise' class extends the Noise class. We use it to insert errors into the data. It provides the 'perturbData' method, which receives a relation and the noise parameters. The method calls the 'perturbColumnData' method in case of a vertical split, the 'perturbRowData' method in case of a horizontal split, and both methods in case of a double split. The 'perturbRowData' method is shown in Algorithm 1.

Algorithm 1 Noise Insertion in Row Data

```
1: procedure PERTURBROWDATA(relation, noisePercentage, noiseInsidePercentage,
   dataNoiseInKeys, columnPerturbation)
2:     schema ← relation.getSchema()
3:     data ← relation.getData()
4:     numOfOverlappingRows ← relation.getNumOfOverlappingRows()
5:     numToPerturb ← round((noisePercentage / 100) * numOfOverlappingRows)
6:     if numToPerturb = 0 then
7:         return relation
8:     end if
9:     indicesToPerturb ← pickUniqueRandomIndices(numOfOverlappingRows,
                          numToPerturb)
10:    selectableIndices ← list(data.keys())
11:    if not dataNoiseInKeys then
12:        selectableIndices.removeAll(relation.getKeyIndices())
13:        selectableIndices.removeAll(relation.getForeignKeyIndices())
14:        selectableIndices.removeAll(relation.getKeysBeforeNormalization())
15:    end if
16:    if columnPerturbation then
17:        selectableIndices.removeAll(relation.getOverlappingColumnsIndices())
18:    end if
19:    for each rowIndex in indicesToPerturb do
20:        numOfValues ← length(selectableIndices)
21:        numOfErrors ← round((numOfValues * noiseInsidePercentage) / 100)
22:        shuffle(selectableIndices)
23:        for i = 0 to numOfErrors - 1 do
24:            columnIndex ← selectableIndices[i]
25:            entry ← data[columnIndex][rowIndex]
26:            replacement ← chooseNoise(entry, schema[columnIndex], columnIndex,
                           data[columnIndex])
27:            data[columnIndex][rowIndex] ← replacement
28:        end for
29:    end for
30:    return relation
31: end procedure
```

The ‘perturbRowData’ method 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 randomly selects that many row indices. This is done using an adapted version of the iterative Floyd algorithm, where we adjust the indices to start from 0 instead of 1. The algorithm generates k unique random numbers within the range $[0, N-1]$. For each $i \in [N-k, N-1]$, a random number t is chosen from the range $[0, i]$ and added to the set S. If t is already present in S, i is added to the set [53]. The set S contains the chosen row indices from the overlapping rows. 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 ‘perturbColumnData’ method. 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 and calls the ‘chooseNoise’ method for each of those entries. That method checks which of the user-selected error methods are applicable and randomly selects and executes one of them. The new entry replaces the original entry in the relation. The ‘perturbColumnData’ method uses 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 in the selected numeric columns and the other in 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.

To illustrate the workflow of the components during generation, we present an activity diagram of the ‘execute’ method from the ‘Generator’ class in Figure 6. This method connects all components of SYDAG. It receives a ‘GeneratorParameters’ object as input, which contains the configuration of the user and the CSV-file. It also receives the output path for the result. The method executes the first steps sequentially. It reads the user’s configuration parameters from the ‘FormdataWrapper’ object, which is contained in the ‘GeneratorParameters’ object. Next, it creates a ‘Relation’ object from the transferred CSV-file. After that, it determines the key of the ‘Relation’ object and then executes the split, which creates the new datasets. The following steps are executed iteratively for each of these generated datasets. The method applies normalization, schema noise, data noise, and joins. If the user has not selected noise or structural changes, these components return the unchanged dataset. After those steps are completed, the method shuffles the relations of the datasets and writes them to CSV-files at the same time. Additionally, it creates files that describe the key and foreign key relationships. Lastly, the method creates a mapping and stores it in the transferred output path, along with the other generated files.

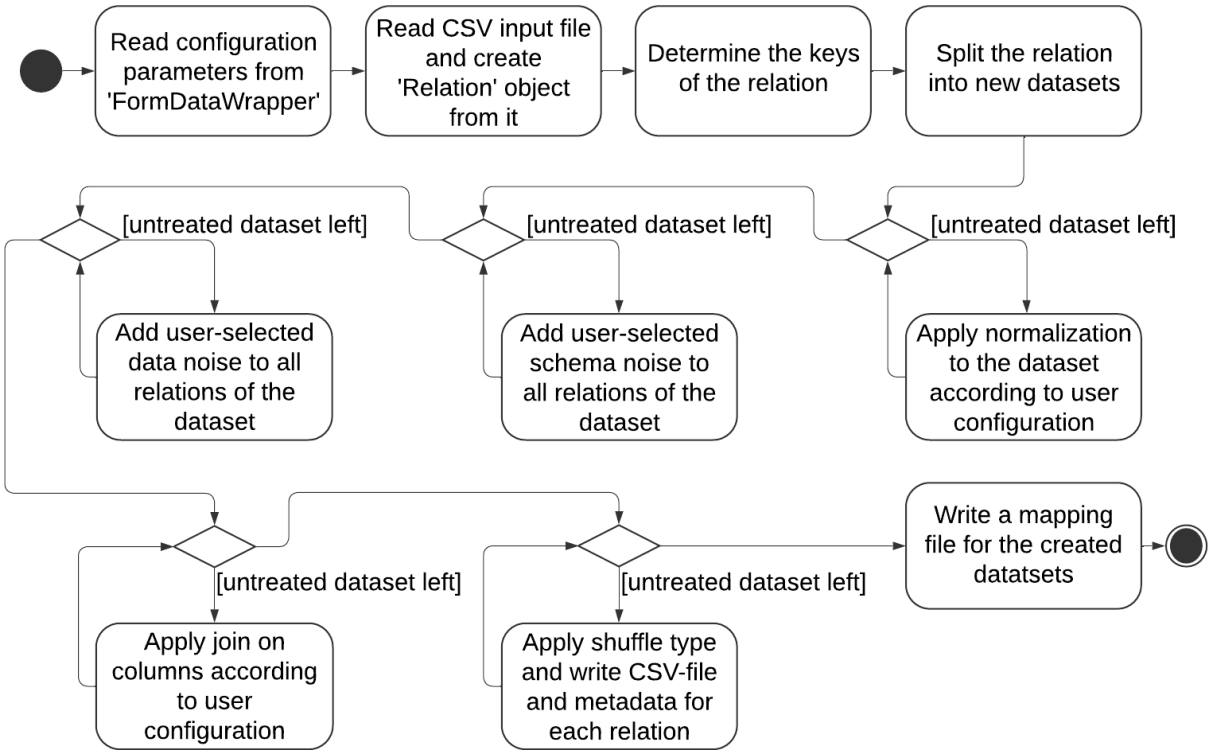

Figure 6: Activity diagram for the 'execute' method of the 'Generator' class of SYDAG.

4.3 Frontend

The GUI of SYDAG is based on a project by Fitzsimons, which is available on GitHub [54]. He provides a multi-step form implementation that guides the user through the individual steps of an interface. The SYDAG frontend is based on the core framework of Fitzsimons' implementation. However, we strongly adapt it to the characteristics of SYDAG. For instance, we add validations to all fields to ensure a correct input for the algorithm. Our implementation consists of ten separate steps. For each step of the multi-step form, we utilize 'StepProbs' as input and output parameters. These variables reflect the current values that the user entered in the GUI. We implement dedicated functions that update these variables according to the user's settings.

In the first step of the multi-step form, the user uploads a CSV-file, specifies whether it includes headers, and sets the separator, quote, and escape characters. In the following step, the user chooses between manual configuration or the upload of a JSON configuration file. They can also download a template that includes an example of a JSON configuration. Next, the user defines the split type, as well as the overlap and distribution of rows and columns. In the fourth step, the user selects how SYDAG should modify the structure of each created dataset. In case of normalization or joining, the user must also specify a percentage. In the fifth step, for each dataset the user selects whether, how much, and which schema noise SYDAG should generate. Step six allows the same choices for the data. Figure 7 shows a screenshot of the GUI for the Data Noise configurations. Next, the user defines the shuffle type for each dataset. After all these steps are com-

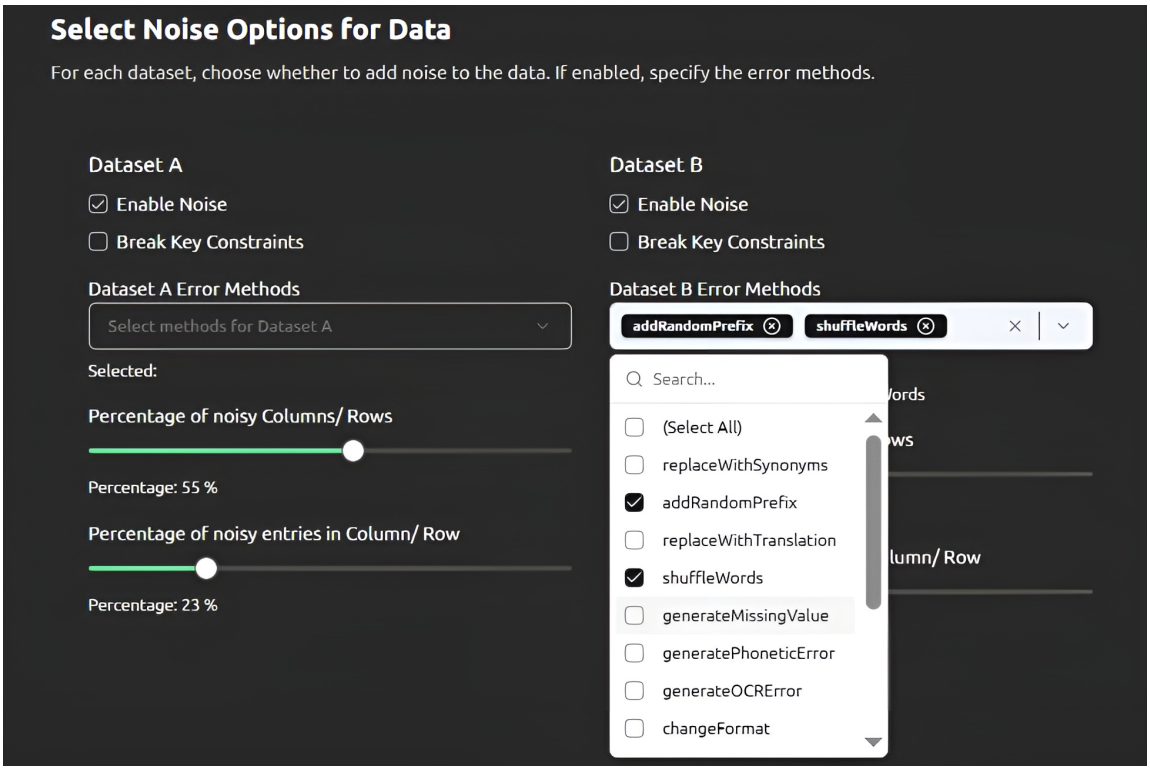

Figure 7: Example of SYDAG's graphical user interface.

pleted, step eight shows a summary of all selected parameters. If the user uploads a JSON configuration, the summary appears directly after step two. The user can confirm the configurations in the summary. Once the user confirms the summary of their input and submits the request, the frontend processes all 'StepProbs' values into the second part of the multipart JSON object. It combines the second part with the first part, which is the uploaded CSV-file. Then the frontend sends the multipart JSON object to the backend for further processing. While the backend generates the results, a message informing the user that the generation process is running is shown as step nine. As soon as the frontend receives the response from the backend with the created ZIP file, the GUI displays a success message. Then, the user can save the datasets via a download button.

5 Evaluation

In the previous sections, we provide an overview of the design and implementation of SYDAG. Following this, we need to evaluate our generator. The goal of our evaluation is to assess SYDAG's ability to generate integration scenarios of different complexity that challenge schema matchers. To achieve this, we start with the presentation of the datasets used for the evaluation in Section 5.1. Then, Section 5.2 explains how users can alter the configurations of SYDAG to create integration scenarios of different complexity. We use these configurations to generate multiple integration scenarios from our chosen datasets. Section 5.3 introduces three schema matchers that we run on our generated scenarios. It

also explains the metrics that we use to evaluate the matching results. Finally, we measure and evaluate the performance of the matchers on the integration scenarios in Section 5.4.

5.1 Information on Datasets

Because SYDAG relies on real data sources as input, it is necessary to select datasets for the evaluation. These should contain as many diverse properties as possible to test SYDAG across different requirements. We select four datasets in CSV format that are all available through public Internet sources. These datasets each include distinct characteristics.

The first dataset, which we name *'Bridges'*, is available in the UCI Machine Learning Repository [55]. This dataset contains information about bridges with each row representing a bridge and its attributes. It consists of 13 columns and 108 rows and stands out from the other datasets because it does not include headers. We want to include this feature in the datasets because SYDAG can process datasets both with and without headers. The dataset includes four columns with numeric entries, while the rest are alphanumeric. In addition, the primary key consists of only one column.

On Kaggle, one can find the second dataset, which we title *'Diabetes'* [56]. It provides medical data to predict the likelihood of patients to develop diabetes. Each row represents a female patient and each column describes a health indicator. The dataset consists of nine columns, 768 rows of data, and a row of headers. A feature that sets this dataset apart from the others is that it contains only numeric data entries. This ensures that we can adequately evaluate the numerical noise methods provided by SYDAG. Additionally, this dataset contains a primary key that includes three columns. This is the largest key among the four datasets, which makes it ideal for checking if SYDAG processes keys correctly.

The third dataset, which we name *'Mental'*, is available on Kaggle [57]. The dataset contains information about the mental health of people who work remotely. Each row represents an employee with attributes detailing personal characteristics and mental health factors. The dataset consists of 20 columns and 5000 rows, which makes it significantly larger than the others. This dataset also features a single-column key but unlike 'Bridges' it includes headers. Because 'Mental' contains seven numeric columns and 13 alphanumeric columns, it challenges SYDAG to effectively handle a mix of both value types.

Our fourth dataset, which we title *'Gym'*, can be found on Kaggle [58]. It provides information on gym members. Each row represents a person with columns detailing fitness attributes. The dataset comprises 15 columns, 973 rows of data, and a header row, making it the second largest of the datasets. Its key consists of two columns. Similar to 'Diabetes' this dataset contains a high proportion of numeric values, 13 out of the 15 columns to be exact. This combination is useful for our evaluation because it ensures that the system handles alphanumeric columns correctly, despite their small number. In addition, the dataset contains many decimal numbers, which SYDAG should perturb correctly.

Table 1 summarizes the characteristics of the four datasets. This includes the number of rows and columns, the number of columns in the key, the distribution of value types, and whether the dataset contains headers. These different characteristics make the datasets useful for our evaluation of SYDAG.

Dataset	Headers	Rows	Columns	Numeric Columns	Alphanumeric Columns	Key Columns
Bridges	No	108	13	4	9	1
Diabetes	Yes	768	9	9	0	3
Mental	Yes	5000	20	7	13	1
Gym	Yes	973	15	13	2	2

Table 1: Summary of the characteristics of the datasets used for evaluation.

5.2 Creation of different Complexity Levels

In order to examine whether SYDAG can generate integration scenarios that challenge schema matchers, it is essential to understand how the configuration influences the scenario generation. Therefore, it is necessary that we analyze how the user must adjust the parameters to achieve a certain level of complexity in the output. Due to the many different settings that are available, there are various possibilities of integration scenarios that we need to consider. This leads us to the third research question.

Third Research Question:

How can users configure the synthetic dataset generator SYDAG to produce data integration scenarios with varying levels of complexity?

To address this question, we must take the configuration parameters of SYDAG into account. The user can adjust different settings to control the number of errors and structural changes to produce integration scenarios of varying complexities. It is important to note that each data integration tool perceives the complexity differently because the tools use different characteristics of the data during the integration process [13]. When users utilize SYDAG, they can tailor the configurations to the requirements of the integration tool that they are testing. A user can achieve the desired complexity by fine-tuning. For our evaluation, we use three different schema matchers and therefore only aim for a broad categorization into three complexity levels. This provides a balance between the necessary granularity and the flexibility of the levels to be applicable to multiple matchers. We give attention to ensuring that each complexity level contains potential challenges for all different matchers. Some challenges can be specific to individual matchers, but there are others that overlap and challenge all the matchers simultaneously. This enables the distinction between the three complexity levels. We define the levels by the possible values

of the configuration parameters that generate such a scenario. Therefore, we summarize a selection of possible parameter assignments for each complexity level. We use the full configuration possibilities of SYDAG. This means that in the complex levels, we include a high number of errors, which leads to extremely difficult integration scenarios. We do this to test how matchers react to very complicated scenarios that SYDAG can generate.

Parameters of the First Complexity Level

Split: We allow the horizontal or vertical split. A combination of both splits requires more integration effort and, therefore, we use it only in higher complexity levels [21]. We also require a high degree of overlapping columns or rows with a minimum of 60 percent. The more columns or rows overlap, the easier it is to find matches during integration [35]. Additionally, we require a relatively even distribution of the non-overlapping columns and rows. One result relation of the split must include 40 to 60 percent of them. Regarding the selection of overlapping rows, we only allow block overlap. This approach creates simpler scenarios compared to random overlap because the search space decreases. Matchers that rely on groupings also benefit from block overlap [36],[39].
Structural Changes: We only allow the 'No Change' option. This ensures that there are no major structural differences between the datasets of the integration scenario. The differences would complicate the matching process, but we want to keep it simple [13].
Noise: For the noise parameters, we require SYDAG to not introduce errors into the schema or the data of all datasets. This way, we achieve a scenario in which there are no challenges due to differently named values, which greatly simplifies the integration process [21], [36].
Shuffle: The shuffle type must also remain unchanged because a shuffled order of rows or columns increases the complexity [13],[39].

Parameters of the Second Complexity Level

Split: Similar to the first level, we require the user to select either a horizontal or vertical split. To increase complexity, we reduce the overlap percentage. It must lie between 50 to 60 percent [35]. In addition, the distribution percentage of non-overlapping columns or rows should range from 30 to 40 or 60 to 70 percent. This means that one of the created relations includes more columns or rows than the other. In addition to block overlap, we also permit random overlap, which can further increase complexity [36].
Structural Changes: The user can select from all options. However, to ensure that the structures of the datasets do not differ too much, we require small percentages up to a maximum of 35 percent for join and normalization. This means that SYDAG only joins a few columns and performs a small number of decomposition steps. Additionally, the user may alter only one of the two generated datasets' structures. This increases heterogeneity compared to the first complexity level but leaves options for a higher level [2].
Noise: Regarding schema noise, we require the user to insert errors in one of the two generated datasets. We expect the percentages of noisy attributes to range from 40 to 60

percent. We do not allow the deletion of the schema because this poses major challenges for matchers that work with attribute names [13]. The user should mainly choose from schema noise methods that keep the attribute names recognizable. These include abbreviations, a random prefix, or letter shuffling. Regarding data noise, the user must enable it for at least one of the two generated datasets while preserving the key constraints. The percentage of noisy rows or columns should range from 30 to 60 percent, while the percentage of noisy values inside such a row or column should lie between 10 and 40 percent. The selected data noise methods must primarily include those that keep the value recognizable, such as typing errors or word shuffling [36].
Shuffle: The shuffle type remains unchanged because shuffling significantly challenges matchers that take the order of entries into account [13],[39].

Parameters of the Third Complexity Level

Split: We allow the user to select from all split types. The double split increases the complexity of the integration scenario because it generates four datasets instead of two [2], [21]. However, we can also make a scenario with one split more challenging by using a smaller overlap [35]. This overlap must lie under 50 percent. The distribution of the non-overlapping columns and rows should not be balanced.
Structural Changes: We increase the structural changes by requiring the user to select different structure options for at least two of the datasets. In addition, we expect percentages between 30 to 50 for the number of joins or decomposition steps. As a result, we can achieve a high heterogeneity within the integration scenario [2].
Noise: For schema noise, the user must choose at least two of the generated datasets for error insertion, with one containing at least 70 percent of noisy attributes. Alternatively, the user can delete the schema of the dataset. The noise percentages for the remaining datasets must exceed 40 percent to ensure a lot of errors. The greater the number of errors, the more challenging it can be for matchers to identify identical attributes [13]. The selected methods should include the generation of a random string because this makes the actual column name unrecognizable. For the data noise, the user must enable error insertion for at least two of the generated datasets. In addition, at least one dataset must contain more than 60 percent of columns or rows with errors. The other datasets that receive noise must include at least 20 percent of noisy columns or rows. The percentages for how many entries SYDAG changes within a column should vary between 10 and 40. For intense complexity, the user can choose that SYDAG breaks the key constraints [21].
Shuffle: Regarding the shuffle type, at least one of the datasets must shuffle the rows or columns. This results in different sequences of data that increase complexity [13], [39].

We generate integration scenarios of the three complexity levels for each of the four test datasets presented in 5.1. We do not use the same configuration for multiple datasets. Instead, for each dataset, we select different parameters from the listed complexity levels.

Additionally, we make sure to cover all possible parameter settings that correspond to the levels. For each dataset we create an easy, medium, and difficult integration scenario, which leads us to a total of 12 scenarios. Overall, the variations in our integration scenarios provide possibilities to test the performance of schema matchers in a versatile way. Interested users can access the different configuration files via the GitHub project [59].

Regarding the third research question, we summarize how users can apply SYDAG to generate data integration scenarios with different complexity levels. Despite the different capabilities of integration tools, 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 [21]. Another aspect is the structural changes. As SYDAG applies more diverse structural changes or shuffle options, the integration of the datasets becomes increasingly heterogeneous and challenging [2]. Furthermore, data and schema noise play an important role. The greater the number of noisy datasets and the higher the noise percentages, the more challenging it becomes for matchers to identify overlapping attributes and recognize that they represent the same real-world entity [7],[13]. Overall, we can state that when we increase the percentages, it results in a more complex integration scenario.

5.3 Schema Matchers and Metrics

To evaluate SYDAG, we want to determine whether the various complex integration scenarios generated by the tool can effectively challenge schema matchers. To achieve this, we measure how different matchers perform on the integration scenarios. For that, we select three suitable matchers. Because there is a difference in the approach of schema-based and instance-based matchers, it is logical to use both groups for our evaluation [21]. We choose the schema-based Levenshtein Matcher to determine whether SYDAG can modify attribute names in a challenging way [13]. Data scientists often use this matcher to detect editing errors, which makes it suitable for SYDAG's type of error insertion [21]. We also include an instance-based matcher to determine whether SYDAG can modify data in a challenging manner. We choose to apply the Jaccard-Instance Matcher, which uses subword-level tokenization with 2-gram generation and the Jaccard measure to match the data entries [13]. It determines the similarity of two columns by summarizing the tokens of all entries for each column and comparing the two sets of tokens using the Jaccard measure. It is useful to include this matcher because tokenization allows flexibility in string recognition [21]. Finally, we choose a third matcher that uses metadata from the datasets for the matching process. We select the Distinct-Count Matcher, which calculates the similarity of two columns by counting their distinct entries and dividing the smaller count by the larger count [28],[60]. Overall, the choice of the three matchers enables us to address different aspects of schema matching, which improves the evaluation.

The schema matching tool Schematch provides access to implementations of the three matchers that we use for the evaluation. Schematch is provided by Vielhauer via GitHub [60]. As input the tool accepts datasets that store their relations in the form of CSV-files. This allows us to directly transfer the created datasets from SYDAG to Schematch. As ground truth, Schematch requires similarity matrices that represent the correct matches of the different relations. We create these matrices from the mapping file that SYDAG outputs and pass them to Schematch. In addition to a high number of matchers, Schematch also offers a clear output of the matching results. They are output in the form of similarity matrices that summarize the detected similarities. Additionally, Schematch includes metrics to evaluate the performance of the matchers. It calculates them using the ground truth similarity matrices as well as the similarity matrices generated in the matching process. An advantage is that the user can add personalized metrics, which enables a customized evaluation [60]. This allows us to select and implement our own metrics for the evaluation of SYDAG. We decide on the calculation of precision, recall, and F-measure. To calculate these metrics, we first need to pick a threshold above which a match should be classified. We define and use two different thresholds to make the evaluation more accurate. The first threshold we pick is already implemented in Schematch [60]. The threshold is described as the lowest relevant score from the similarity matrix. Schematch determines this value by examining all entries in the similarity matrix where the corresponding ground truth matrix contains a 1. Schematch selects the smallest value from these entries. This means that we can define Threshold 1 as the smallest similarity value that describes an actual match [60]. We use a self-selected statistical value as the second threshold. First, we detect all values in the similarity matrix that are marked with 1 in the ground truth. Then, we calculate the mean value and the standard deviation from these values. We define Threshold 2 as the mean value minus the standard deviation. The corresponding calculation is based on Equation 5 and 7 and shown in Equation 11.

$$T_2 = \bar{x} - s \tag{11}$$

We use two different thresholds to enable a more comprehensive analysis of the matchers' performances. Because we define Threshold 1 as the smallest value of an actual match, no true matches remain undetected. This results in a recall of 1 for all integration scenarios. Therefore, Threshold 1 is useful for integration processes in which no matches should be overlooked [20]. However, this can cause low precision. With Threshold 2 we intend to create a better balance between recall and precision than with Threshold 1. The advantage of Threshold 2 is that it shows how well the matcher performs in an application where sufficient precision and recall are required. It can be useful in cases where missing some true matches is not problematic [13].

To use Schematch, we add the implementations for precision, recall, and F-measure. We adopt the implementation of the first threshold because Vielhauer already implemented this in Schematch. Additionally, we add our own implementation for the second threshold. This makes it possible to use the matchers from Schematch and evaluate them on our own metrics [60].

5.4 Analysis of Results

In Sections 5.1 to 5.3 we define the basis of our evaluation. We select four datasets from each of which we generate three integration scenarios of varying complexity. We evaluate these scenarios by using our defined thresholds and metrics on the matching results of three different matchers: the Levenshtein Matcher, the Jaccard-Instance Matcher, and the Distinct-Count Matcher. These conditions for the evaluation enable us to analyze the influence of the different complexity levels, which SYDAG can generate, on the specific matchers. We want to discover whether the complex scenarios of SYDAG pose a challenge for the matchers because this is the goal of a useful generator [2],[36]. We construct the fourth research question from these objectives.

Fourth Research Question:

What effect do different levels of complexity in data integration scenarios created with SYDAG have on the performance of the Levenshtein Matcher, Jaccard-Instance Matcher, and Distinct-Count Matcher?

To answer this research question, we first make the necessary measurements. We transfer our 12 data integration scenarios and their ground truth similarity matrices to Schematch. In Schematch's configuration, we specify the three matchers we want to test. Schematch executes each of the matchers on the 12 integration scenarios. It stores the matching results in separate similarity matrices. In addition to matching, Schematch also evaluates the performance of the matcher on the corresponding integration scenario. For this, we use the metrics we added to Schematch: precision, recall, and F-measure. Schematch calculates the performance metric values under our two different thresholds. After the successful execution, it outputs the similarity matrices along with the performance measure values, which are calculated from them. These calculations are performed individually for each combination of matcher and integration scenario. The created integration scenarios and matching results are available on GitHub [59].

We present the results of Schematch in Table 2. The table organizes the results by the three matchers. For each of the matchers, it shows the four datasets and their three respective integration scenarios of different complexity levels. The table includes the results of precision, recall, and F-measure under Thresholds 1 and 2. Overall, it provides

Levenshtein Matcher							
		Threshold 1			Threshold 2		
Dataset	**Complexity**	**Precision**	**Recall**	**F-Measure**	**Precision**	**Recall**	**F-Measure**
Bridges	1	0.0	0.0	0.0	0.0	0.0	0.0
	2	0.0	0.0	0.0	0.0	0.0	0.0
	3	0.0	0.0	0.0	0.0	0.0	0.0
Diabetes	1	1.0	1.0	1.0	1.0	1.0	1.0
	2	0.888889	1.0	0.941177	1.0	0.75	0.857143
	3	0.107143	1.0	0.193548	0.272727	0.75	0.4
Mental	1	1.0	1.0	1.0	1.0	1.0	1.0
	2	0.077465	1.0	0.143791	0.888889	0.727272	0.8
	3	0.052350	1.0	0.099492	0.063177	0.714286	0.116086
Gym	1	1.0	1.0	1.0	1.0	1.0	1.0
	2	0.522727	1.0	0.686567	0.888889	0.695652	0.780488
	3	0.064815	1.0	0.121739	0.064815	1.0	0.121739
Jaccard-Instance Matcher							
		Threshold 1			Threshold 2		
Dataset	**Complexity**	**Precision**	**Recall**	**F-Measure**	**Precision**	**Recall**	**F-Measure**
Bridges	1	1.0	1.0	1.0	1.0	1.0	1.0
	2	0.391304	1.0	0.5625	0.589744	0.851852	0.696970
	3	0.3125	1.0	0.476191	0.4375	0.7	0.538462
Diabetes	1	0.818182	1.0	0.9	1.0	0.888889	0.941177
	2	0.142857	1.0	0.25	0.21875	0.875	0.35
	3	0.131868	1.0	0.233010	0.140845	0.833333	0.2409638
Mental	1	0.75	1.0	0.857143	0.75	1.0	0.857143
	2	0.174603	1.0	0.297297	0.333334	0.636364	0.4375
	3	0.052350	1.0	0.099492	0.220779	0.693878	0.334975
Gym	1	1.0	1.0	1.0	1.0	1.0	1.0
	2	0.127072	1.0	0.225490	0.333333	0.782609	0.467533
	3	0.101449	1.0	0.184211	0.316239	0.880952	0.465409
Distinct-Count Matcher							
		Threshold 1			Threshold 2		
Dataset	**Complexity**	**Precision**	**Recall**	**F-Measure**	**Precision**	**Recall**	**F-Measure**
Bridges	1	0.346154	1.0	0.514286	0.346154	1.0	0.514286
	2	0.126582	1.0	0.224719	0.183673	0.9	0.305084
	3	0.084640	1.0	0.156069	0.127907	0.814815	0.221106
Diabetes	1	0.818182	1.0	0.9	0.8	0.888889	0.842105
	2	0.129032	1.0	0.228571	0.318182	0.875	0.466667
	3	0.116505	1.0	0.208696	0.196078	0.833333	0.317461
Mental	1	0.315789	1.0	0.48	0.315789	1.0	0.48
	2	0.053922	1.0	0.102326	0.064	0.727273	0.117647
	3	0.055682	1.0	0.105489	0.064639	0.693878	0.118261
Gym	1	0.789474	1.0	0.882353	0.75	0.8	0.774194
	2	0.194915	1.0	0.326241	0.246575	0.782609	0.375
	3	0.079696	1.0	0.147627	0.206704	0.880952	0.334842

Table 2: Performance results of the Levenshtein Matcher, Jaccard-Instance Matcher, and Distinct-Count Matcher on the four test datasets and their respective complexity levels. The values are measured under Threshold 1 (Lowest Relevant Score) and Threshold 2 (Mean minus Standard Deviation).

an overview of the performance metrics for all matchers across the integration scenarios. In the following sections, we analyze the results to gain information about the matchers' performances. We individually look at the three matchers and evaluate their results with regard to both thresholds. We analyze the results of the Levenshtein Matcher in Section 5.4.1, the Jaccard-Instance Matcher in Section 5.4.2, and the Distinct-Count Matcher in Section 5.4.3. Lastly, we perform an overall comparison of the matchers in Section 5.4.4.

5.4.1 Levenshtein Matcher

The first matcher we analyze in terms of performance is the Levenshtein Matcher. Figure 8 visualizes the results for this matcher separated by the two different thresholds. The complexity degree of the integration scenario is shown on the x-axis. For each complexity degree, the integration scenarios we generated using the four test datasets are considered. The performance metric values are measured on the y-axis. They include precision, recall and F-measure. Precision and recall are shown in the form of bars, while the F-measure is represented by connected points. The F-measure provides the best indication of performance, which allows us to observe performance changes based on the plotted line [13].

For the Levenshtein Matcher, it is important to note that all performance-measures for the 'Bridges' dataset have a value of 0. The reason is that the label-based Levenshtein Matcher compares attribute names, but the 'Bridges' dataset does not include a header line. For this reason, the Levenshtein Matcher cannot be applied and we only analyze the performance on the remaining three datasets.

We begin by examining the first complexity level, where we observe identical values under both thresholds. The Levenshtein Matcher achieves a perfect score of 1 for precision, recall, and f-measure across all integration scenarios. This demonstrates optimal performance by the matcher. The reason is that in the first complexity level, SYDAG does not include any errors in the attribute names. Therefore, the Levenshtein Matcher is able to find the exact matches among all attributes of the integration scenarios.

For the second complexity level, we observe different values under both thresholds. We start by analyzing Threshold 1. The definition of this threshold ensures that matchers achieve a recall of 1 for all integration scenarios. It is therefore interesting to analyze the precision values that result from a high recall. We observe a strong variation in the precision values for the three integrations scenarios. Especially for the integration scenario 'Mental', the matcher achieves an extremely low precision value of 0.08. This implies that the smallest relevant score is lower than many similarity scores of column pairs that are not a true match. As a result, these column pairs are incorrectly recognized as matches. We can identify the column pair with the lowest relevant score. The columns 'myHt' and 'Physical_Activity' have a high Levenshtein distance although they represent the same real-world entity. For the scenario 'Gym', the precision is 0.52, which implies some false positives. However, for 'Diabetes', the Levenshtein Matcher achieves a high precision of

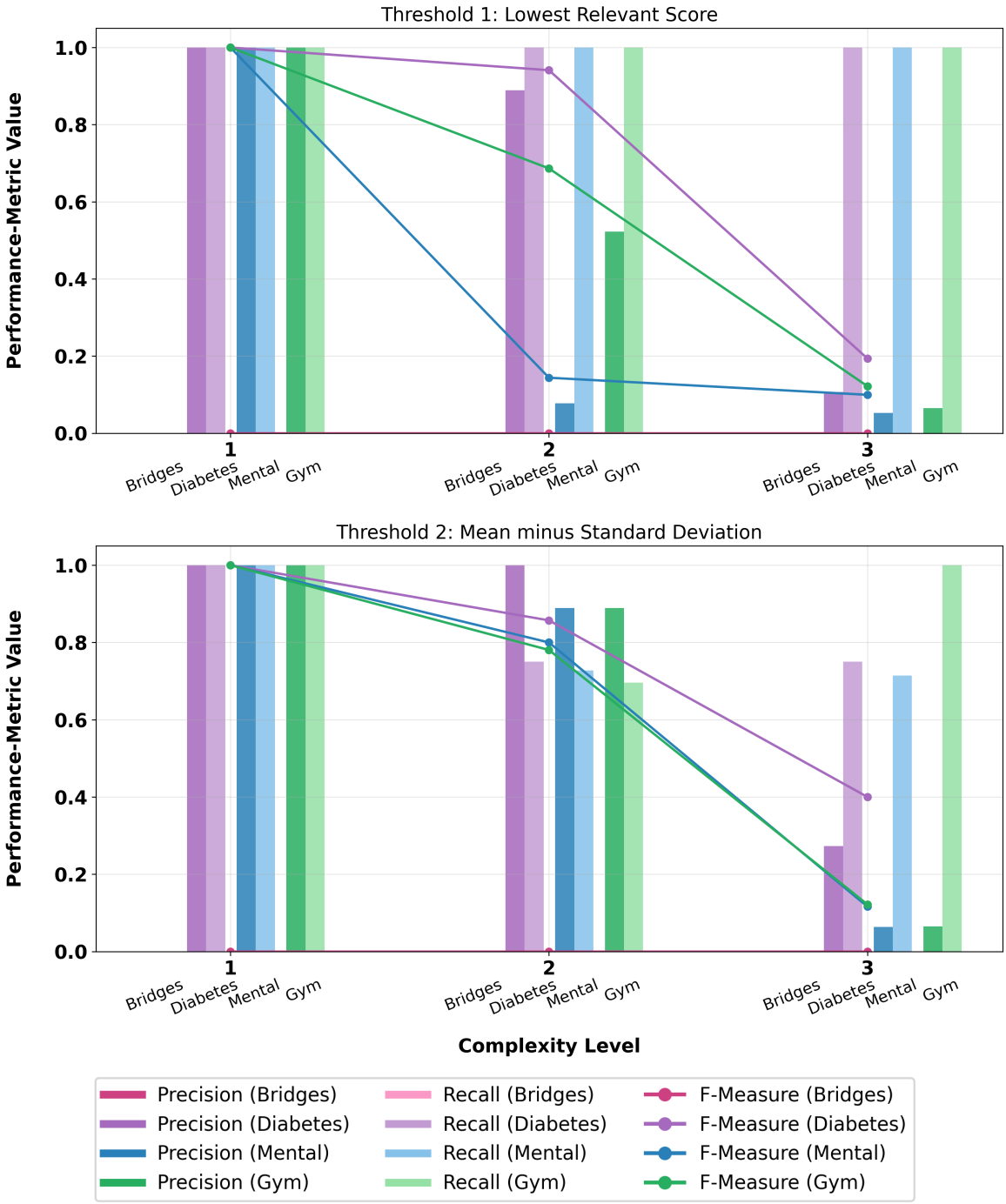

Figure 8: Comparison of the performance metrics of the Levenshtein Matcher at two different thresholds.

0.89. The phenomenon of low precision highlights the disadvantage of Threshold 1. Although we can identify all true matches, the precision is negatively impacted, which affects the F-measure. It leads to a noticeable decline of the matcher's F-measure in the 'Gym' and 'Mental' scenarios from the first to the second complexity level.

Under Threshold 2, we observe different results for the second complexity level than under Threshold 1. All scenarios show a high precision above 0.88. Particularly for the 'Mental' scenario, we notice that the precision is 0.81 higher than under Threshold 1. The

reason is that we define Threshold 2 as the mean minus the standard deviation, which leads to fewer false positives. Additionally, the true matches with extremely small similarity values no longer fall into the detected matches, which is why the recall for all scenarios drops to values ranging from 0.69 and 0.75. Based on the F-measure, we observe that Threshold 2 enables a better balance between precision and recall. The values all lie above 0.78, representing a relatively high performance of the Levenshtein Matcher.

For the third complexity level, we first analyze Threshold 1. All integration scenarios show precision values below 0.11. This is due to the fact that in the third complexity level, SYDAG incorporates a large number of errors and uses methods that change the attribute names completely. This causes the lowest score of a true match to be very small, which is why many other column pairs are incorrectly recognized as matches. The low precision also results in F-measure values, which all lie below 0.2. This indicates a poor performance by the matcher. We can observe similar results under Threshold 2. For the integration scenarios 'Diabetes' and 'Mental', the recall values lie at 0.75 and 0.71, which implies that some actual matches are missed. Regarding precision, for the 'Diabetes' scenario, we obtain a value of 0.27, which is above those of the other two scenarios, whose values lie at 0.06. We also observe an F-measure value of 0.4 for the 'Diabetes' scenario, which is twice as high as that under Threshold 1. This indicates a better balance between precision and recall under Threshold 2. The F-measure values of the other integration scenarios differ only minimally from those under Threshold 1, which highlights the poor performance of the matcher.

By analyzing the F-measure, we can observe how the performance of the Levenshtein Matcher changes across the different complexity levels. Under Threshold 2, the performance decreases by 0.14 to 0.22 between Complexity Levels 1 and 2 across all integration scenarios. This indicates that even small percentages of errors 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, maintaining F-measure values above 0.78. Between Complexity Levels 2 and 3, there is a more significant drop in performance compared to the decrease between Complexity Levels 1 and 2. All F-measure values decrease by at least 0.45. This indicates that the high number of errors and the use of complex error methods complicate matching. Structural changes also contribute to the performance decrease. For instance, the join creates complex joined attribute names that are difficult to match. With regard to Threshold 1, we can confirm this observation for all scenarios except 'Mental'. Here, the performance drops by 0.86 from the first to second complexity level and changes only minimally by 0.04 from the second to the third. This indicates that if the focus of matching lies on identifying all matches, the second complexity level presents a great challenge when it includes error methods that make attributes unrecognizable. However, if the focus lies on a balance of recall and precision, the Levenshtein Matcher is strongly challenged only by Complexity Level 3.

5.4.2 Jaccard-Instance Matcher

Next, we analyze the performance of the Jaccard-Instance Matcher. We visualize the measured results in Figure 9, using the same axis labeling as in Figure 8. We begin by analyzing the first level of complexity. Under Threshold 1, we observe that for the integration scenarios 'Bridges' and 'Gym', both precision and recall are 1. This means that all matches are correctly identified. However, for the scenarios 'Diabetes' and 'Mental', the precision lies at 0.82 and 0.75. The reason is the tokenization of numeric values. Numbers often contain recurring sequences of digits, which leads the matcher to assign high similarity scores even when the numbers are unrelated. The 'Diabetes' scenario only consists of numeric columns and the 'Mental' scenario includes seven numeric columns in the overlap. Therefore, the matcher performs worse on these scenarios. Under Threshold 2, the 'Diabetes' scenario is the only one where the results differ from those under Threshold 1. The precision reaches 1 but recall lies at 0.89. This shows that with a higher threshold, we do not detect false matches, but miss some true ones. Nevertheless, the F-measure values differ only slightly between the thresholds and remain above 0.85 across all scenarios.

Next, we analyze Threshold 1 for the second complexity level. Across all integration scenarios, the precision decreases by more than 0.58 from the first to the second complexity level. This occurs because the inserted errors alter the token sets of true matches, resulting in small similarity scores. However, in the 'Bridges' scenario, we use error methods that only slightly distort the values, causing a precision of 0.39. This exceeds the precision values of the other scenarios, which range from 0.12 to 0.17. We observe a similar trend under Threshold 2, with the difference that the balance between recall and precision improves. For all scenarios, the recall ranges from 0.63 to 0.88 but the precision lies between 0.21 and 0.59, which is higher than for Threshold 1. Overall, the F-measure values decrease significantly between Complexity Level 1 and 2 across all scenarios for both thresholds. Threshold 2 results in only slightly higher F-measures ranging from 0.35 to 0.7 than Threshold 1, whose values lie between 0.22 and 0.56.

From the second to the third complexity level, we observe a decrease in precision for the 'Bridges' and 'Mental' scenarios. Under Threshold 1, the precision values drop to 0.31 and 0.05. This indicates that the inclusion of a large number of errors results in many different tokens, which challenge the matcher. In contrast, the precision values for the 'Diabetes' and 'Gym' scenarios decrease by less than 0.03 between these complexity levels. The small decline is due to the application of normalization. SYDAG can introduce less noise into the data of decomposed relations if we prohibit noise in keys. Under Threshold 2, precision for 'Bridges' and 'Mental' also decreases from the second to the third complexity level. The decreases are 0.15 and 0.11, which means they are greater than those under Threshold 1. However, this comes with a cost of recall values ranging from 0.69 to 0.88. For the 'Gym' scenario, precision remains nearly unchanged, meaning it only

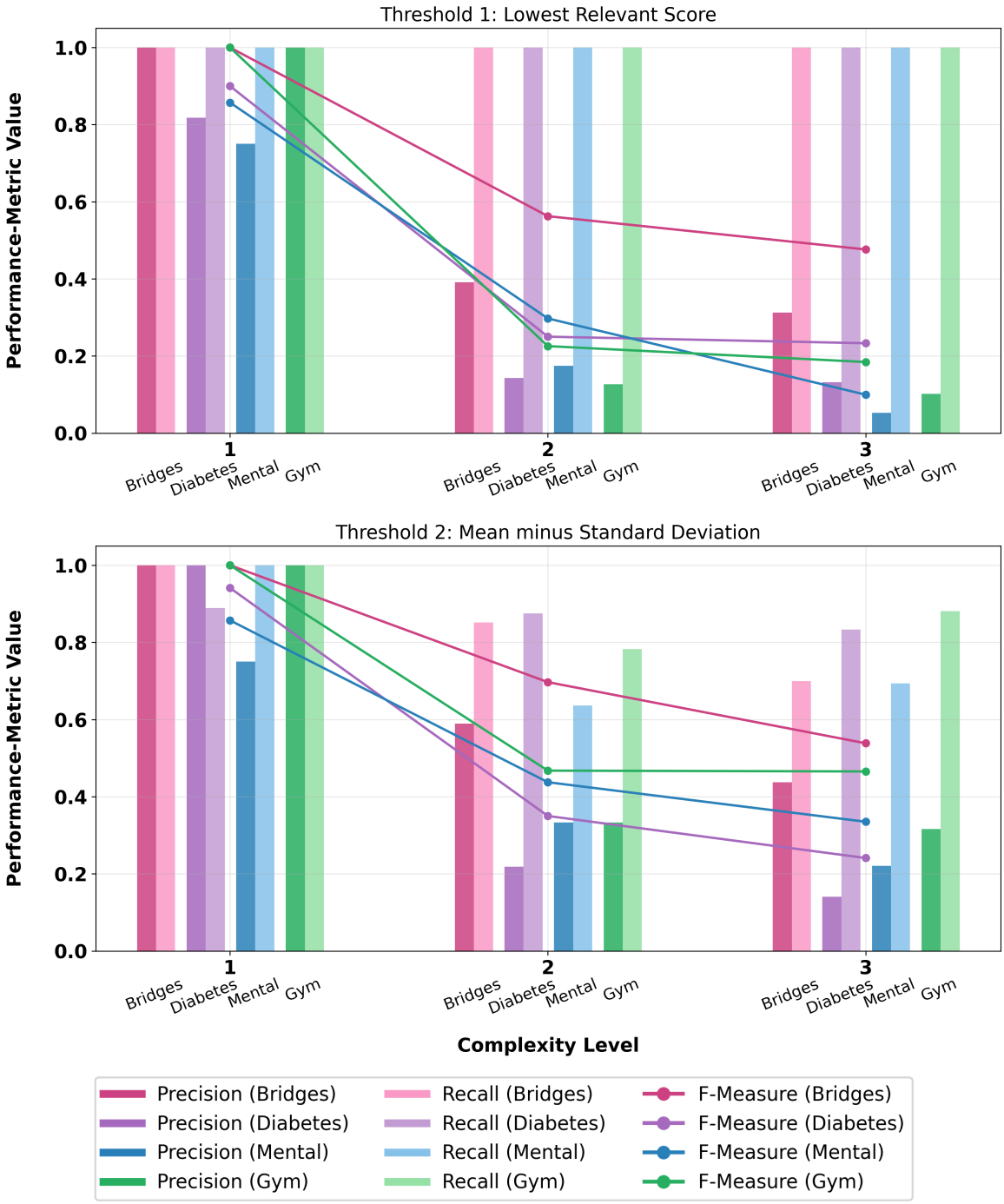

Figure 9: Comparison of the performance metrics of the Jaccard-Instance Matcher at two different thresholds.

decreases by 0.002 from the second to the third complexity level. A difference from the results under Threshold 1 appears in the 'Diabetes' scenario. Under Threshold 2, we observe a decrease in precision by 0.08 between Complexity Levels 2 and 3, which is a greater decrease than under Threshold 1. This means that if some true matches can be missed, the third complexity level leads to a greater drop in precision compared to when all matches must be found. The trend is also reflected in the F-measure values.

Finally, we evaluate the overall performance of the matcher using the F-measure. Under both thresholds, we observe a performance decrease of more than 0.3 between Complexity Levels 1 and 2 across all integration scenarios. In contrast, between Complexity Levels 2 and 3, the performance changes only slightly by at most 0.19. This suggests that error insertion presents a significant challenge, even when only minor structural changes are made. Additionally, we observe smaller performance decreases under Threshold 1 than under Threshold 2, especially between Complexity Levels 1 and 2. Between those levels, the smallest performance decrease under Threshold 1 is 0.44, whereas under Threshold 2 it is 0.31. This indicates that when the focus lies on identifying all matches, the second complexity level already poses a great challenge. However, when we aim for a balance between precision and recall, the third complexity level often increases the challenges.

5.4.3 Distinct-Count Matcher

The final matcher we analyze is the Distinct-Count Matcher. Its results are shown in Figure 10, using the same axis labeling as in Figure 8. For the first complexity level, we observe similar results under both thresholds. Under Threshold 1, the matcher achieves precision values exceeding 0.78 for the integration scenarios 'Diabetes' and 'Gym'. The reason is that both datasets are mostly numerical. Columns with numeric entries include a broader range of distinct elements because the numeric entries show greater variability compared to the alphanumeric entries in the other two datasets. Consequently, the numbers of distinct elements in the columns typically differ, which allows the matcher to identify true matches. In contrast, the precision for 'Bridges' and 'Mental' lies at 0.35 and 0.32. This is the case because the two datasets contain many columns with a limited number of possible entries. As a result, there are multiple columns with identical counts of distinct elements that the matcher incorrectly labels as matches. Under Threshold 2, the recall and precision values for the 'Bridges' and 'Mental' scenarios are identical to those under Threshold 1. For the 'Diabetes' and 'Gym' scenarios, the precision values differ slightly, measuring 0.8 and 0.75, while the recall values lie at 0.89 and 0.8. However, the F-measure values differ only minimally under both thresholds. For 'Bridges' and 'Mental,' the F-measure values are relatively low under both thresholds, ranging from 0.48 to 0.51.

For the second complexity level, under Threshold 1 we observe a precision between 0.05 and 0.19 across all integration scenarios. The reason is that the number of distinct elements in the columns changes because of error insertion. Even a small number of errors in the dataset has a significant impact in this case. Under Threshold 2, the recall values range from 0.72 to 0.9. However, the precision values for the 'Diabetes' and 'Gym' scenarios measure 0.32 and 0.25, meaning they are greater than those under Threshold 1. This indicates that despite the errors, the matcher is able to identify true matches based on the distinct element counts. For the 'Diabetes' and 'Gym' scenarios, this is due to their high number of numeric columns. For integration scenarios with large differences in the num-

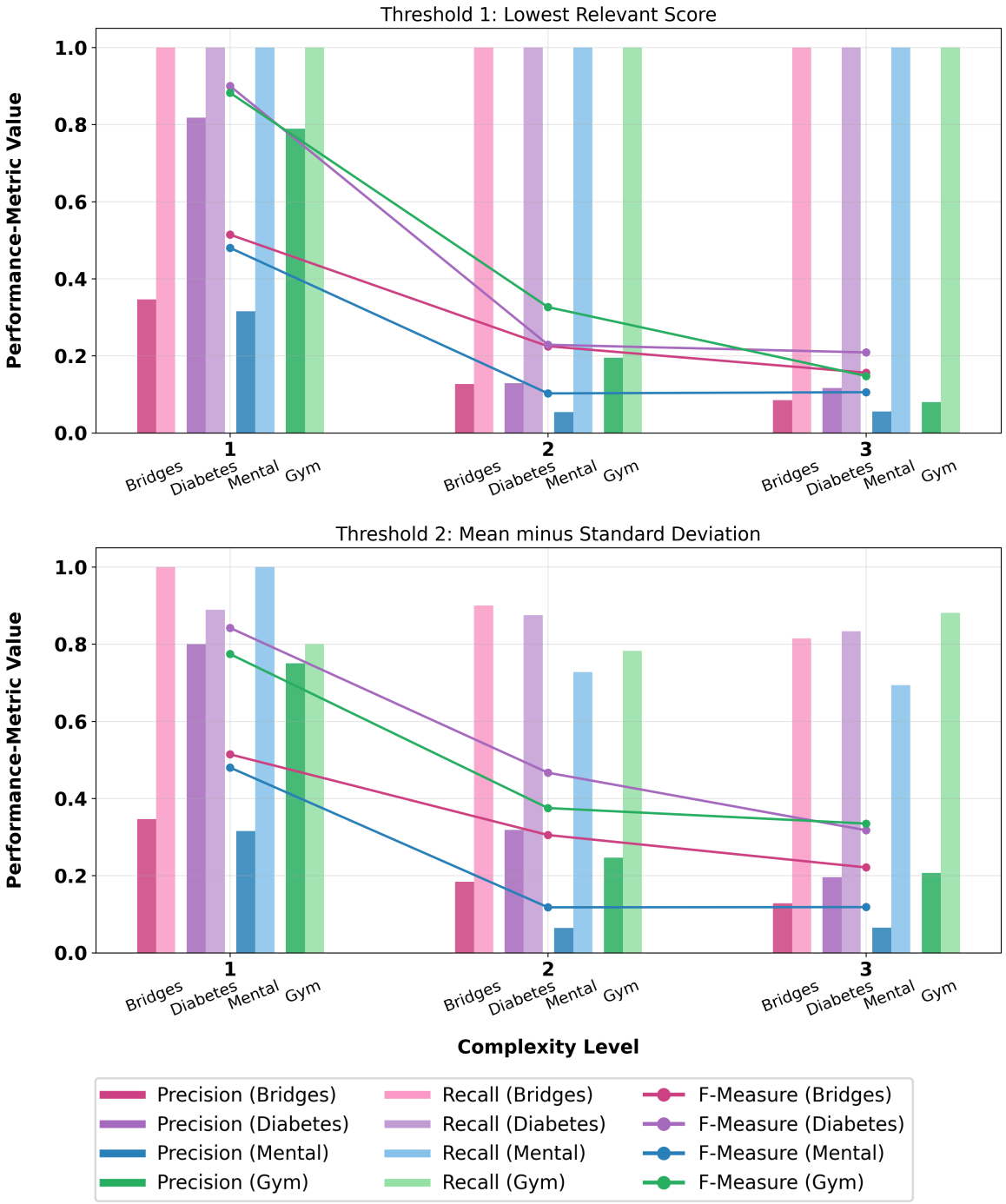

Figure 10: Comparison of the performance metrics of the Distinct-Count Matcher at two different thresholds.

ber of distinct elements in a column, adding a view errors has less impact on the matches because the differences remain prominent. In contrast, scenarios that include multiple columns with similar numbers of distinct elements are more affected by errors. In such cases, even minor changes disrupt the matches because the distinct element counts are closely aligned. Another aspect that limits performance is the distribution of rows during the horizontal split. An uneven distribution of rows causes one created dataset to con-

tain more distinct entries than the other. This complicates finding true matches. The F-measure values reflect the drop in performance under both thresholds. From the first to the second complexity level they drop by at least 0.21.

Under Threshold 1 we observe a slight decrease in precision of the 'Bridges', 'Diabetes', and 'Gym' scenarios from the second to the third complexity level. The decrease ranges from 0.01 to 0.11. For the 'Mental' scenario precision even minimally increases by 0.002. This indicates that under Threshold 1, the third complexity level only adds small challenges. Under Threshold 2, we observe higher precision values than under Threshold 1 for all scenarios except 'Mental'. The values range from 0.12 to 0.2. However, the recall values across all scenarios range from 0.69 to 0.89, meaning some true matches are missed. Overall, we observe a small decrease in the F-measure values. Under both thresholds, the decrease ranges from 0.02 to 0.18. For 'Mental' the F-measures even show a minimal increase of less than 0.003. The reason is that due to normalization, SYDAG adds fewer errors to the integration scenarios. The slight decrease in the matchers performance occurs when we significantly increase the error percentages to compensate for the phenomenon.

For the final analysis, we examine the performance based on the F-measure. Under both thresholds, we observe a more significant performance drop between Complexity Levels 1 and 2 than between Levels 2 and 3. This applies to all integration scenarios. Under Threshold 1, the F-measure values drop by at least 0.29 between Complexity Levels 1 and 2. In contrast, the decrease from Complexity Level 2 to 3 is smaller, ranging from 0.003 to 0.18. This implies that when the aim is to find all matches, the second complexity level already poses a considerable challenge for the Distinct-Count Matcher. Under Threshold 2, we observe higher performances than under Threshold 1. The F-measure decreases range from 0.21 to 0.4 between Complexity Levels 1 and 2. However, the transition to the third complexity level also results in smaller performance decreases, which do not exceed 0.15. This also indicates that the second complexity level poses a great challenge. Overall, we conclude that even small percentages of inserted errors challenge the Distinct-Count Matcher because these errors distort the number of distinct elements in the columns. The impact of additional errors is mitigated through the application of normalization with key preservation and, therefore, only slightly increases the difficulty for the matcher.

5.4.4 Overall Comparison

To complete the answer to the fourth research question, we compare the overall performance of the three matchers by summarizing their results. Specifically, we calculate the average F-measure across all integration scenarios for each complexity level. This provides a performance overview for each matcher under both thresholds. We present the results of the calculations in Table 3. It is important to note that we exclude the F-measure values of the 'Bridges' integration scenario from the performance calculation of the Levenshtein Matcher because it cannot be applied to integration scenarios without headers.

Matcher	Complexity 1	Complexity 2	Complexity 3
F-Measure for Threshold 1			
Levenshtein Matcher	1.0000	0.5905	0.1383
Jaccard-Instance Matcher	0.9393	0.3338	0.2482
Distinct-Count Matcher	0.6942	0.2205	0.1545
F-Measure for Threshold 2			
Levenshtein Matcher	1.0000	0.8125	0.2126
Jaccard-Instance Matcher	0.9496	0.4880	0.3950
Distinct-Count Matcher	0.6526	0.3161	0.2479

Table 3: Presentation of the average F-measure values across the different datasets for each matcher and complexity level. A distinction is made between the two thresholds.

In Figure 11, we visualize the results of the average performances of the three matchers. The complexity levels are shown on the x-axis and the F-measure values on the y-axis. We observe a similar trend for the Distinct-Count Matcher and the Jaccard-Instance Matcher. For both, the performance drop is higher from the first to the second complexity level than from the second to the third. We observe this pattern under both thresholds. Under Threshold 1, 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. For the Jaccard-Instance 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. Under Threshold 2, the performance drops are slightly smaller for both matchers but still exceed a decrease of at least 0.33 between Complexity Levels 1 and 2. Between Complexity Levels 2 and 3, the decreases are less than 0.1, similar to Threshold 1. This leads to the conclusion that even when not all matches need to be found, the second complexity level already presents a major challenge. In contrast, the Levenshtein Matcher shows a different behavior. Under Threshold 1, we observe a drop of 0.41 between the first and second complexity levels and a decrease of 0.45 between the second and third. This indicates that Complexity Levels 2 and 3 represent a similar rise in difficulty when we attempt to find all matches. Under Threshold 2, the drop in performance between Complexity Levels 1 and 2 is only about one-third as large as the drop between Levels 2 and 3. This means that if some matches are allowed to remain undetected, the second complexity level poses a smaller challenge, whereas the third complexity level becomes a significant challenge for the Levenshtein Matcher.

Overall, we observe that the matchers' performances are strongly influenced by the different complexity levels. For all matchers, we see a decline in performance across the complexity levels. This means that users can apply the complexity levels to create challenging integration scenarios with SYDAG. They can also choose to create more individual challenges by fine-tuning the difficulty of their scenarios to the weaknesses and strengths of the integration tool they are testing. This makes SYDAG very versatile.

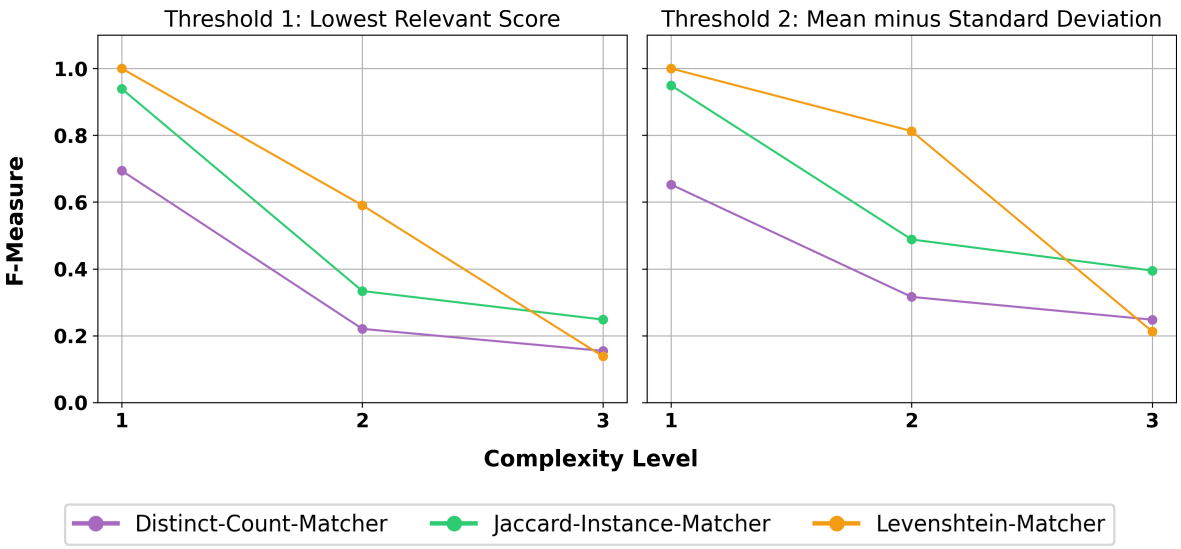

Figure 11: Overview of the performance of three matchers, averaged across all four test datasets.

6 Conclusion

We introduced the synthetic dataset generator SYDAG, which developers can use to create customized data integration scenarios. We identified eight useful components and successfully implemented them in our generator. SYDAG is available on GitHub [59]. Additionally, we examined how SYDAG can be configured to generate data integration scenarios of three different complexity levels. We observed that when we increase the percentages for error insertion and structural modification, the integration scenarios become more complex. Our evaluation of the performance of three schema matchers showed that SYDAG is able to generate integration scenarios that challenge different matchers.

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 different percentages for the individual functionalities, which control the extent of data modifications. The user can also select the combination of error methods that the generator applies, enabling specific customization. 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. In addition, it provides a user-friendly interface. Despite its advantages, SYDAG has a few limitations. It is designed for relational datasets and only supports the input of one relation saved in CSV format. Other data models are not supported. Additionally, SYDAG cannot function without input because it does not generate data itself. It only restructures existing data or inserts errors into it. To enhance SYDAG's capabilities, we could add a connection to relational databases to allow input beyond CSV files. Another possible improvement could be extending the generator to support other data models. Lastly, another potential area for development could be the exact analysis of SYDAG's runtime, which could lead to improvements in its implementation.

References

[1] C. Manzano, A. Miskolczi, H. Stiele., V. Vybornov, T. Fieseler, and S. Pfalzner, "Learning from the present for the future: The Jülich LOFAR long-term archive," in *Astronomy and Computing*, vol. 48, 2024, pp. 1–11.

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

[3] N. Marz and J. Warren, *Big Data : Principles and best practices of scalable real-time data systems*. Manning Publications Co., 2015, ISBN: 9781617290343.

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

[5] A. Bogatu, N. W. Paton, M. Douthwaite, and 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.

[6] V. Crescenzi, A. D. Angelis, D. Firmani, *et al.*, *Alaska: A flexible benchmark for data integration tasks*, 2021. arXiv: 2101.11259.

[7] F. Panse, M. Klettke, J. Schildgen, and 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, and 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.

[9] C. Koutras, K. Psarakis, G. Siachamis, *et al.*, "Valentine in action: Matching tabular data at scale," in *Proceedings of the VLDB Endowment*, vol. 14, 2021, pp. 2871–2874.

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

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

[12] J. L. Harrington, *Relational database design and implementation*, 4th ed. Morgan Kaufmann, 2016, ISBN: 9780128499023.

[13] U. Leser and F. Naumann, *Informationsintegration : Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*, 1st ed. dpunkt.verlag, 2007, ISBN: 9783898644006.

[14] G. Saake and K.-U. Sattler, *Datenbanken und Java : JDBC, SQLJ, ODMG und JDO*, 2nd ed. dpunkt.verlag, 2003, ISBN: 3898642283.

[15] A. H. Kemper and A. Eickler, *Datenbanksysteme: eine Einführung*, 10th ed. De Gruyter, 2015, ISBN: 9783110443752.

[16] G. Saake, K.-U. Sattler, and A. Heuer, *Datenbanken : Konzepte und Sprachen*, 6th ed. mitp Verlag, 2018, ISBN: 9783958457782.

[17] T. Teorey, S. Lightstone, T. Nadeau, and H. Jagadish, *Database Modeling and Design*, 5th ed. Morgan Kaufmann, 2011, ISBN: 9780123820204.

[18] G. Vossen, *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*, 5th ed. R. Oldenbourg Verlag, 2008, ISBN: 9783486275742.

[19] R. F. van der Lans, *Data Virtualization for Business Intelligence System: Revolutionizing Data Integration for Data Warehouses*. Morgan Kaufmann, 2012, ISBN: 97801239 44252.

[20] I. Rossak, *Datenintegration: Integrationsansätze, Beispielszenarien, Problemlösungen, Talend Open Studio*. Hanser, 2013, ISBN: 9783446434912.

[21] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012, ISBN: 9780124160446.

[22] S. Halder and S. Ozdemir, *Hands-On Machine Learning for Cybersecurity*, 1st ed. Packt Publishing, 2018, ISBN: 9781788992282.

[23] S. Tarride and C. Kermorvant, *Revisiting n-gram models: Their impact in modern neural networks for handwritten text recognition*, 2024. arXiv: 2404.19317.

[24] S. Fletcher and M. Z. Islam, "Comparing sets of patterns with the Jaccard Index," in *Australasian Journal of Information Systems*, vol. 22, 2018, pp. 1–17.

[25] K. Bosch, *Basiswissen Statistik : Einführung in die Grundlagen der Statistik mit zahlreichen Beispielen und Übungsaufgaben mit Lösungen*. R. Oldenbourg Verlag, 2007, ISBN: 9783486582536.

[26] L. Fahrmeir, C. Heumann, R. Künstler, I. Pigeot, and G. Tutz, *Statistik : der Weg zur Datenanalyse*, 8th ed. Springer Spektrum, 2016, ISBN: 9783662503713.

[27] E. Alpaydin, *Maschinelles Lernen*, 2nd ed. De Gruyter, 2019, ISBN: 9783110617887.

[28] N. Golov, A. Filatov, and S. Bruskin, "Efficient exact algorithm for count distinct problem," in *Computer Algebra in Scientific Computing*, 2019, pp. 67–77.

[29] J. R. Talburt, *Entity Resolution and Information Quality*, 1st ed. Morgan Kaufmann, 2011, ISBN: 9780123819727.

[30] A. M. Cabrera, C. J. Faber, K. Cepeda, *et al.*, "DIBS: A data integration benchmark suite," in *Proceeding of the ACM/SPEC International Conference on Performance Engineering Companion (ICPE)*, 2018, pp. 25–28.

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

[32] J. Hammer, M. Stonebraker, and 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.

[33] K. Hildebrandt, F. Panse, N. Wilcke, and N. Ritter, "Large-scale data pollution with Apache Spark," in *IEEE Transactions on Big Data*, vol. 6, 2020, pp. 396–411.

[34] E. Ioannou and Y. Velegrakis, "EMBench++: Data for a thorough benchmarking of matching-related methods," in *Semantic Web*, vol. 10, 2019, pp. 435–450.

[35] Y. Lee, M. Sayyadian, A. Doan, and A. S. Rosenthal, "ETuner: Tuning schema matching software using synthetic scenarios," in *VLDB Journal*, vol. 16, 2007, pp. 97–122.

[36] C. Koutras, G. Siachamis, A. Ionescu, *et al.*, "Valentine: Evaluating matching techniques for dataset discovery," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2021, pp. 468–479.

[37] B. Alexe, W. C. Tan, and Y. Velegrakis, "STBenchmark: Towards a benchmark for mapping systems," in *Proceedings of the VLDB Endowment*, vol. 1, 2008, pp. 230–244.

[38] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller, "The IBench integration metadata generator," in *Proceedings of the VLDB Endowment*, vol. 9, 2015, pp. 108–119.

[39] T. Zhong, J. Zhao, X. Guo, Q. Su, and G. Fox, *Rinas: Training with dataset shuffling can be general and fast*, 2023. arXiv: 2312.02368.

[40] B. Steppan, *Einstieg in Java mit Eclipse*. Hanser, 2020, ISBN: 9783446459106.

[41] S. Musib, *Spring Boot in Practice*. Manning Publications, 2022, ISBN: 9781617298813.

[42] K. Spichale, *API-Design: Praxishandbuch für Java- und Webservice-Entwickler*, 1st ed. dpunkt.verlag, 2017, ISBN: 9783864903878.

[43] *Datamuse api*, last accessed 10-March-2025. URL: https://www.datamuse.com/api/.

[44] *Mymemory*, last accessed 10-March-2025. URL: https://mymemory.translated.net/doc/spec.php.

[45] T. Papenbrock and 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.

[46] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann, "Scalable discovery of unique column combinations.," in *Proceedings of the VLDB Endowment*, vol. 7, 2013, pp. 301–312.

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

[48] D. Maharry, *TypeScript Revealed*. Apress, 2013, ISBN: 9781430257257.

[49] B. Cherny, *Programming TypeScript: Making Your JavaScript Applications Scale*, 1st ed. O'Reilly Media, 2019, ISBN: 9781492037651.

[50] M. Riva, *Real-World Next.js: Build Scalable, High-Performance, and Modern Web Applications Using Next.js, the React Framework for Production*, 1st ed. Packt Publishing, 2022, ISBN: 9781801073493.

[51] K. Matthias and S. P. Kane, *Docker Praxiseinstieg*, 2nd ed. mitp Verlag, 2020, ISBN: 9783958459403.

[52] I. Grigorik, *High Performance Browser Networking*, 1st ed. O'Reilly Media, 2013, ISBN: 9781449344764.

[53] J. Bentley and B. Floyd, "Programming pearls: A sample of brilliance," in *Communications of the ACM*, vol. 30, 1987, 754–757.

[54] M. Fitzsimons, *Multi-step form*, last accessed 10-March-2025. URL: https://github.com/Marcosfitzsimons/multi-step-form?tab=readme-ov-file#author.

[55] Y. Reich and S. Fenves, *Pittsburgh Bridges*, UCI Machine Learning Repository, last accessed 10-March-2025. URL: http://archive.ics.uci.edu/dataset/18/pittsburgh+bridges.

[56] H. Rahman, *Diabetes Dataset*, last accessed 10-March-2025. URL: https://www.kaggle.com/datasets/hasibur013/diabetes-dataset.

[57] I. Ramzan, *Remote Work & Mental Health*, last accessed 10-March-2025. URL: https://www.kaggle.com/datasets/iramshahzadi9/remote-work-and-mental-health.

[58] V. Khorasani, *Gym Members Exercise Dataset*, last accessed 10-March-2025. URL: https://www.kaggle.com/datasets/valakhorasani/gym-members-exercise-dataset.

[59] A. Marschner, *SYDAG*, last accessed 10-March-2025. URL: https://github.com/anne-marschner/SYDAG.

[60] A. Vielhauer, *Schematch*, last accessed 10-March-2025. URL: https://github.com/avielhauer/schematch.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Die digital eingereichte Version ist mit der vorliegenden Arbeit identisch.

Unterschrift (Marschner, Anne)	Ort, Datum