Group 11 Final Report:
Fake but Realistic Data

CS4221 AY22/23 Semester 2 - Project Research

Amanda Lian (A0221973A), Kenny Chew (A0200016H),
Sie Xiang Yi Jefferson (A0166887E), Xu Zeng (A0194487L)

Github repo: https://github.com/jeffypie369/mock_data_generator

1 Introduction

In today's data-driven world, we cannot understate the importance of the role that data plays. Data is crucial, but we often do not have enough of it [1]. We often face the challenge of acquiring adequate data for testing data applications or pipelines. To simulate real-world production scenarios, we require datasets that are similar in volume and variety. However, manually creating such datasets is a daunting task due to the need of diverse data types, characteristics and constraints involved. Furthermore, human-generated data is susceptible to implicit biases [2].

In this project, our team aims to tackle this issue through the implementation of a Python Command Line Interface (CLI) program that generates mock data that is not only highly realistic, but customisable.

We address the problem in two ways. Firstly, we need to support user-customisable data generation, to create data sets that satisfy constraints and dependencies defined by the user. Secondly, we need to generate data that is realistic, that include common preset columns drawing from real-world data such as names and addresses. These values can then be further constrained by users through the use of text fields and other options to create a data set that meets their requirements. The project focuses on generating data with various constraints for columns and tables, with the capability to create custom columns built on common SQL data types like VARCHAR, INT, DATE, etc. The generated data can be used for testing and development purposes in a wide range of applications.

2 Existing Literature

In this section, we look at several existing literature and how the features of our project compare against current available capabilities.

2.1 Mockaroo

Mockaroo is a web app that allows users to create their data from scratch [3]. It has a wide range of preset data types that users can create realistic data from over a hundred types, such as names, addresses, IP addresses and gender.

Key features of Mockaroo include being able to specify the percentage of data that would be blank, as well as formulas that the user would like applied to the column. Users can also choose the format of data that they need from ten options, including CSV, JSON and XML.

This tool however has several limitations. The biggest limitation is its inflexibility in implementing inter-column constraints. Mockaroo is primarily a tool generating solo tables and not full schemas. As such, users are unable to specify inter-table dependencies for tables created on Mockaroo as well, which is a huge drawback as databases used in data teams in businesses often have related columns across multiple tables. Additionally, users on the free version of Mockeroo can only download 1000 rows of data each time, which may not be sufficient for comprehensive testing efforts.

2.2 Faker

Faker is a Python library that provides an API to generate realistic data for certain data types such as names, addresses and phone numbers. They also provide data catered to different regions since addresses and phone numbers differ across regions [4].

Due to the limited time and resources given for our project, we made the conscious decision as a group to leverage the ready-made Faker library in order to bypass the more preliminary manual steps of data generation in our methodology. Instead, we shift our focus and dedicate our efforts to the implementations of within-table and between-table constraints and dependencies, which are capabilities surprisingly missing from existing tools available on the market and Internet, yet prove to be increasingly important with the prevalence and complexity of big data in industry today [5].

3 Methodology

3.1 Key Features

1. Generating character data with customizable constraints, including length, pattern, and exclusion of specific values. The program uses the Faker Python library to generate names, addresses, and emails.

2. Generating integer data with uniform distribution, allowing users to input parameters such as the number of values, minimum and maximum values, selectivity percentage, as well as values within the range but to be excluded from the dataset.

3. Generating float data with uniform or normal distributions, depending on the user's input. The user can specify the number of decimal points and provide distribution parameters like mean and standard deviation.

4. Generating dates and times with uniform and normal distributions. Users can input the number of dates, times, or date-times to generate and specify lower and upper bounds, mean values, and standard deviations.

5. Supporting composite keys and foreign keys for tables.

6. Supporting functional dependencies.

3.2 Accounting for Constraints

3.2.1 Foreign Key Constraints

Our program allows for the usage of foreign key constraints in data generation. The caveat here is that the table that the foreign key is referencing must be created first. For the second table onwards, the user will be prompted to input whether there are foreign keys in the table. Our program will randomly obtain rows of data from the referenced table containing the foreign keys. This is in contrast to taking random values of each foreign key from the referenced table as doing so would break the functional dependencies that may exist in the referenced table. Therefore, our program checks for all the foreign keys that belong to a referenced table and extracts data by rows.

3.2.2 Functional Dependencies

Our program allows for Functional Dependency constraints to be applied in data generation. The user will be prompted to input functional dependencies (if any), and these functional dependencies would be referenced to every time a new row is being generated, checking if the row about to be added meets the functional dependency constraints with respect to the previous rows generated.

The implementation of data generation with functional dependencies considered is different from that without functional dependencies. With functional dependencies, data are generated row by row. Without functional dependencies, data are generated by columns.

3.2.3 Selectivity Constraints

Selectivity is a parameter in the given functions that determines the level of uniqueness or repetition in the generated data. It is a value between 0 and 1, where a lower value signifies a higher degree of uniqueness, while a higher value signifies more repetition in the generated data.

Selectivity is useful when generating test data with varying degrees of uniqueness. For example, in a database, you might want to test how different levels of uniqueness in the data affect the performance of queries or indexes.

Here's a more detailed explanation of how selectivity works in the functions:

1. If selectivity is 0 or the number of unique elements required (1 / selectivity) is greater than the number of rows, then all the elements generated are unique.

2. If selectivity is greater than 0 and less than 1, the functions calculate the number of unique elements required (unique_data) as the ceiling of (1 / selectivity).

3. The functions generate the specified number of unique elements (unique_data) by iterating and checking if each generated element is not in the exclusion list.

4. The functions create the final list by appending the unique elements in a cyclical manner, ensuring that the selectivity requirement is satisfied.

5. Lastly, the functions shuffle the final list to randomize the order of elements, maintaining the desired selectivity level while providing a more natural-looking dataset.

In summary, selectivity is a measure of how often a value is repeated in the generated data, with a higher value resulting in more repetition, and a lower value resulting in greater uniqueness. This parameter is useful for generating test data that simulates various scenarios for database or application performance testing.

3.2.4 Non-equality Constraints

Non-equality constraints are used to exclude specific values or patterns from the generated data, ensuring that the output adheres to the desired specifications or avoids potential issues during testing or validation.

The exclusion parameter is an optional argument in the functions that allows users to specify a list of values that should not be included in the generated data. By providing a list of values to the exclusion parameter, users can customize the output to meet specific requirements or to prevent known problematic values from appearing in the test data.

For example, when generating a list of email addresses, you might want to exclude addresses from a specific domain or those containing certain keywords to comply with data privacy regulations. Similarly, when generating a list of names, you might want to exclude offensive or inappropriate words.

The non-equality constraints are implemented in the functions using the exclusion parameter, which is a list of values to be excluded from the generated data. The functions check each generated value against the exclusion list and, if there's a match, the value is regenerated until it satisfies the constraint.

Here's a brief overview of how the exclusion parameter is used in the functions:

1. Initialize the exclusion parameter with an empty list or user-provided values.

2. Generate a value using the appropriate generation logic.

3. Check if the generated value exists in the exclusion list. If the value is found in the exclusion list, regenerate the value and repeat the check. If the value is not in the exclusion list, add it to the generated data.

4. Continue this process until the desired number of values is generated.

In conclusion, non-equality constraints, implemented through the exclusion parameter, allow users to control the generated data more precisely and exclude specific values or patterns from the output. This feature is particularly useful for tailoring test data to comply with specific requirements, avoiding known issues, or ensuring data integrity during testing and validation.

3.3 Implementation

The implementation of the project is done using Python, with the main code in a script named "prompt.py". This script is responsible for gathering user inputs and generating the corresponding data based on the constraints provided. The script also takes care of foreign key constraints when not the first table, as well as handling composite keys. It is designed to generate custom data for a user-defined database schema. It allows users to define tables and their attributes, specify data types, constraints, and data distributions. Here's a more detailed explanation and documentation of the script:

main() function:

1. The script prompts the user to input the number of tables they want to create (num_tables).
2. For each table, the user is asked to provide the table name (table_name) and the number of rows of data needed (num_rows).
3. The user specifies the number of columns for each table, except for foreign keys.
4. For each column, the user provides the column name (entity) and selects one of the known data types, or provides a custom data type (CHAR, INT/FLOAT, or DATE/TIME).
5. The script gathers general constraints, such as primary key designation, uniqueness, and selectivity, for each column.
6. Depending on the chosen data type, the script prompts the user to provide specific constraints and distribution parameters.
7. For tables other than the first one, users can specify foreign key relationships.
8. Intra-table constraints, such as functional dependencies, and inter-table constraints can be defined, but their implementation is not provided in the current code.
9. The script utilizes the Faker Python library for generating realistic data such as names, addresses, and emails. The library supports multiple locales for generating data from different regions or languages (e.g., French).

Our program allows users to customize data constraints and select data types by providing various generator functions that accept parameters such as the maximum length of the generated data, number of rows, selectivity, and exclusion list. These generator functions are designed to handle different data types and distributions, such as CHAR, VARCHAR, INT, DATE, and various statistical distributions like uniform, normal, and Poisson.

The implementation of various data types and distributions is achieved by using different generator functions for each data type. For example, the name_generator function generates names, the address_generator generates addresses, and the postcode_generator generates postcodes.

3.3.1 Char

We have implemented eight functions in the script to generate mock char data for various types of fields, including names, addresses, postcodes, random strings, ISBNs, IDs, email addresses, and credit card numbers. These functions are designed to help generate synthetic data for testing and development purposes, saving time and effort for developers who need to generate realistic data to work with.

Each of these functions comes with a set of parameters that can be customized to meet specific user requirements. Developers can use these functions to generate a wide variety of data while ensuring that any required exclusions, selectivity factors, or other constraints are met.

Customization of character data is facilitated by providing specific parameters to the generator functions. For character data types, the script allows customization of the length, pattern, and exclusion list. For instance, the generate_random_strings function accepts a pattern parameter that dictates the composition of the generated strings, such as whether they contain only letters, only digits, or a combination of both. By utilizing these customizable parameters, developers can generate highly tailored synthetic data that meet their specific needs.

Functions

1. **name_generator(max=50, num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of names using the Faker library. It considers the maximum length of the names, number of rows required, selectivity constraint, and a list of names to exclude.

   Parameters:

   - max (int): The maximum length of the names. Default is 50.
   - num_rows (int): The number of names to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2.
   - exclusion (list): A list of names to exclude from the generated output. Default is None.

   Returns: A list of generated names.

2. **address_generator(max=50, num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of addresses using the Faker library. It considers the maximum length of the addresses, number of rows required, selectivity constraint, and a list of addresses to exclude.

   Parameters:

   - max (int): The maximum length of the addresses. Default is 50.
   - num_rows (int): The number of addresses to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2.
   - exclusion (list): A list of addresses to exclude from the generated output. Default is None.

   Returns: A list of generated addresses.

3. **postcode_generator(num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of postcodes using the Faker library. It considers the number of rows required, selectivity constraint, and a list of postcodes to exclude.

   Parameters:

   - num_rows (int): The number of postcodes to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2.
   - exclusion (list): A list of postcodes to exclude from the generated output. Default is None.

   Returns: A list of generated postcodes.

4. **generate_random_strings(length=10, pattern=None, num_rows=1, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of random strings based on the specified length, pattern, number of rows required, selectivity constraint, and a list of strings to exclude.

   Parameters:

   - length (int): The length of the random strings to generate. Default is 10.

- A list of characters representing the pattern for the random strings. Use 'l' for letters, 'd' for digits, and 'x' for any character. Default is None.
- num_rows (int): The number of random strings to generate. Default is 10.
- selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2.
- exclusion (list): A list of strings to exclude from the generated output. Default is None.

Returns: A list of generated random strings.

5. **isbn_generator(num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of random ISBN-10 numbers. It considers the number of rows required, selectivity constraint, and a list of ISBNs to exclude.

   Parameters:

   - num_rows (int): The number of ISBN-10 to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2. If the selectivity is 0 or greater than the number of rows, all generated ISBNs will be unique.
   - exclusion (list): A list of ISBNs to exclude from the generated output. Default is None.

   Returns: A list of random ISBN-10 numbers.

6. **id_generator(min=1, max=50, num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of random IDs. It considers the minimum length, maximum length, number of rows required, selectivity constraint, and a list of IDs to exclude.

   Parameters:

   - min (int): The minimum length of the generated ID (default: 1).
   - max (int): The maximum length of the generated ID (default: 50).
   - num_rows (int): The number of IDs to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2. If the selectivity is 0 or greater than the number of rows, all generated IDs will be unique.
   - exclusion (list): A list of IDs to exclude from the generated output. Default is None.

   Returns: A list of random IDs.

7. **email_generator(max=50, num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of random email addresses. It considers the maximum length, number of rows required, selectivity constraint, and a list of IDs to exclude.

   Parameters:

   - max (int): The maximum length of the generated ID (default: 50).
   - num_rows (int): The number of email addresses to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2. If the selectivity is 0 or greater than the number of rows, all generated email addresses will be unique.
   - exclusion (list): A list of email addresses to exclude from the generated output. Default is None.

```
names = name_generator()
addresses = address_generator()
postcodes = postcode_generator()
random_strings = generate_random_strings(length=5, pattern=['l', 'd', 'x', 'x',
'x'])

print(names)
print(addresses)
print(postcodes)
print(random_strings)
```

Figure 1: CHAR types example

```
['Emma Smith', 'Oliver Johnson', 'Sophia Williams', 'Liam Brown', ...]
['1234 Elm St, Chicago, IL 60601', '5678 Oak St, Los Angeles, CA 90001', ...]
['90210', '10001', '60601', '94102', ...]
['a3b5c', 'p9q8r', 'f6g7h', ...]
```

Figure 2: CHAR types output

Returns: A list of random email addresses.

8. **credit_card_number_generator(num_rows=10, selectivity=0.2, exclusion=None)**

   Description:

   This function generates a list of random credit card numbers. It considers the number of rows required, selectivity constraint, and a list of credit card numbers to exclude.

   Parameters:

   - num_rows (int): The number of credit card numbers to generate. Default is 10.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.2.
   - exclusion (list): A list of credit card numbers to exclude from the generated output. Default is None.

   Returns: A list of generated credit card numbers.

Figure 1 shows an example of some of the CHAR data types being called and Figure 2 shows the output of the generated data.

3.4 Numeric

We have implemented four functions in the script to generate mock integer and float data. In designing these functions, we made a conscious effort to maximize the customisability of the numeric data generation process, by providing multiple different options and approaches for the target audience to tailor the data best to their needs. These include the choice to specify integers that lie in the user-defined range yet users may want to exclude from generation. Customisability options also include the flexibility to generate floats in a non-uniform distribution, such as the normal distribution or the Poisson distribution.

Float Distributions We expect the most commonly used distribution for floats to be the uniform distribution, which generates all possible floats values (made finite by the user-specified number of decimal places) within the user-defined range with the same likelihood. This would be good for float

columns where the statistical distribution has little impact on the user's planned application of the mock data, or in situations where the user lacks either sufficient information on or strong mathematical understanding of the actual data they are trying to simulate.

Non-uniform distributions can give users a more accurate modelling and representation of the data they are trying to simulate and conduct testing on, by following a statistical distribution similar to what is expected of the actual data. Use cases of float generation in Poisson distribution include situations where users need to simulate count data with a random and constant rate of occurrence, such as analyzing the frequency of events. On the other hand, normal distribution is commonly used to model continuous data that are symmetrically distributed around a mean value. For instance, it could model the heights of people in a population, the weights of objects produced by a manufacturing process, or the scores of students on an exam.

Functions

1. **int_generator(n, min, max, exclusion=None, unique=False, selectivity=0)**

   Description:

   This function generates a list of integers with a discrete uniform distribution, where all integers including and between the user-defined lower and upper bound integer values have the same likelihood of being generated. Users may also optionally specify exclusion values, uniqueness, as well as selectivity of values.

   Parameters:

   - n (int): Total count of integers to generate.
   - min (int): Minimum value of list of integers.
   - max (int): Maximum value of list of integers.
   - exclusion (list): A list of unique values in range (min, max) to be excluded from generation. Default is None.
   - unique (bool): True if integers generated must be unique, tops up with NULL values if the number of distinct integers available is less than the number of integers to generate. Default is False.
   - selectivity (float): Percentage in range (0,1] that represents the probability that any row will be a particular value. Default is 0, meaning no selectivity constraint enforced.

   Returns: A list of generated integers.

2. **float_generator(n, decimals=2, distribution='uniform', min=0, max=10, exclusion=None, unique=False, selectivity=0, mean=0, sd=1)**

   Description:

   This function generates a list of floats with the user-specified number of decimal places, and in a distribution chosen by the user. Subsequent function arguments used are dependent on the distribution used.

   Parameters:

   For all distributions:

   - n (int): The total count of floats to generate.
   - decimals (int): The number of digits after the decimal point. Default is 2.
   - distribution (string): The type of distribution that floats will be generated in. Values are 'uniform', 'normal', 'normal_minmax' or 'poisson'.

   For uniform distribution:

   - min (float): Minimum value of list of floats.
   - max (float): Maximum value of list of floats.

- exclusion (list): A list of unique values in range (min, max) to be excluded from generation. Default is None.
- unique (bool): True if floats generated must be unique, tops up with NULL values if the number of distinct floats available is less than the number of floats to generate. Default is False.
- selectivity (float): Percentage in range (0,1] that represents the probability that any row will be a particular value. Default is 0, meaning no selectivity constraint enforced.

For normal distribution:

- mean (float): Mean value of floats. Default is 0.
- sd (float): Standard value of floats. Default is 1.

For normal_minmax distribution:

- min (float): Estimated minimum value of list of floats. This value will be generated with 0.1% probability on the left tail of the statistical normal distribution.
- max (float): Maximum value of list of floats. This value will be generated with 0.1% probability on the right tail of the statistical normal distribution.

For poisson distribution:

- mean (float): Mean value of floats. Default is 0, which is illegal in Poisson distributions, so user-input value must be obtained.

Returns: A list of generated floats.

3. **int_generator_single(min, max, exclusion=None)**

   Description:

   This function generates one integer from the user-defined range with equal probabilities of all integers except for values specified to be excluded.

   Parameters:

   - min (int): Minimum value of integer.
   - max (int): Maximum value of integer.
   - exclusion (list): A list of unique values in range (min, max) that are not to be generated. Default is None.

   Returns: A single generated integer.

4. **float_generator_single(decimals=2, distribution='uniform', min=0, max=10, exclusion=None, mean=0, sd=1)**

   Description:

   This function generates one float with the user-specified number of decimal places, from a range with probabilities corresponding to the distribution chosen by the user. Subsequent function arguments used are dependent on the distribution used.

   Parameters:

   For all distributions:

   - decimals (int): The number of digits after the decimal point. Default is 2.
   - distribution (string): The type of distribution that the float will be generated from. Values are 'uniform', 'normal', 'normal_minmax' or 'poisson'.

   For uniform distribution:

- min (float): Minimum value of float.
- max (float): Maximum value of float.
- exclusion (list): A list of unique values in range (min, max) that are not to be generated. Default is None.

For normal distribution:

- mean (float): Mean value of distribution from which float is generated. Default is 0.
- sd (float): Standard value of distribution from which float is generated. Default is 1.

For normal_minmax distribution:

- min (float): Estimated minimum value of float. This value can be generated with 0.1% probability on the left tail of the statistical normal distribution.
- max (float): Estimated maximum value of float. This value can be generated with 0.1% probability on the right tail of the statistical normal distribution.

For poisson distribution:

- mean (float): Mean value of distribution from which float is generated. Default is 0, which is illegal in Poisson distributions, so user-input value must be obtained.

Returns: A single generated float.

3.5 DateTime

We have implemented three functions in the script to generate mock datetime data.

Each of these functions comes with a set of parameters that can be customized to meet specific user requirements. Developers can use these functions to generate a wide variety of data while ensuring that any required exclusions, selectivity factors, or other constraints are met.

Functions

1. **generate_time(lower_bound_time=[0,0,0], upper_bound_time=[23,59,59], number_rows=10, exclusion=None, selectivity=0)**

   Description:

   This function generates a list of times using the Faker library. It considers the lower and upper bound times, number of rows required, selectivity constraint, and a list of times to exclude.

   Parameters:

   - lower_bound_time (int[]): Lower bound of time. Default is [0,0,0] (00:00:00)
   - upper_bound_time (int[]): Upper bound of time. Default is [23,59,59] (23:59:59)
   - num_rows (int): The number of times to generate. Default is 10.
   - exclusion (list): A list of times to exclude from the generated output. Default is None.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.

   Returns: A list of generated times.

2. **generate_date(lower_bound_date=datetime.today() - timedelta(days=30*365), upper_bound_date=datetime.today(), number_rows=10, exclusion=None, selectivity=0)**

   Description:

   This function generates a list of dates using the Faker library. It considers the lower and upper bound dates, number of rows required, selectivity constraint, and a list of dates to exclude.

   Parameters:

- lower_bound_date (int[]): Lower bound of date. Takes in an array of integers in the format [YYYY, MM, DD]. Default is set to 30 years before the current date.
- upper_bound_datetime (int[]): Upper bound of date. Takes in an array of integers in the format [YYYY, MM, DD]. Default is set to the current date.
- num_rows (int): The number of dates to generate. Default is 10.
- exclusion (list): A list of dates to exclude from the generated output. Default is None.
- selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.

Returns: A list of generated dates.

3. **generate_datetime(lower_bound_datetime=datetime.today() - timedelta(days=30*365), upper_bound_datetime=datetime.today(), number_rows=10, exclusion=None, selectivity=0)**

   Description:

   This function generates a list of datetimes using the Faker library. It considers the lower and upper bound datetimes, number of rows required, selectivity constraint, and a list of datetimes to exclude.

   Parameters:

   - lower_bound_date (int[]): Lower bound of datetime. Takes in an array of integers in the format [YYYY, MM, DD, hh, mm, ss]. Default is set to 30 years before the current datetime.
   - upper_bound_date (int[]): Upper bound of datetime. Takes in an array of integers in the format [YYYY, MM, DD, hh, mm, ss]. Default is set to the current datetime.
   - num_rows (int): The number of datetimes to generate. Default is 10.
   - exclusion (list): A list of datetimes to exclude from the generated output. Default is None.
   - selectivity (float): The selectivity constraint. Value should be between 0 and 1. Default is 0.

   Returns: A list of generated datetimes.

4 Future Works

The existing implementation of our program includes a number of preset data types for realistic data, including name, ID, address, email, postcode, etc. However, the available preset options are rather limited in the current skeletal implementation of our mock data generation tool. For instance, we currently only store and support name, address and email data fields in English and French. In the future, we want to expand to include even more data fields such as colors, latitude and longitude coordinates. We also intend to expand the language options to provide multilingual support to more users internationally.

There is also the potential to include support for more schema types such as the star schema, as well as inter-table join constraints, that may help users model their datasets and simulate the interactions between tables more accurately and realistically. Additionally, adding the capability for the program to understand and evaluate open-ended string inputs for check constraints will also go a long way in improving user customisability and data realism in our tool.

Lastly, in the aspect of user experience, we aim to speed up data generation to cater to larger datasets with more rows of data. Potentially, we can incorporate more types of output download formats for users to choose from, such as JSON, XML or Excel, that may provide convenience for users in terms customizing the generated data for use in unit testing and modelling in their existing systems. We also intend to include the function to allow users to correct any erroneous inputs that they had made in an earlier field. This can be accomplished through developing an interactive and visually appealing user interface that also serves to provide a more pleasant experience for users.

5 Conclusion

The CLI program we have implemented addresses the limitations of existing data generation tools and allows for the creation of complex and varied datasets that accurately reflect real-world scenarios. With its ability to produce data that adheres to the cardinality, participation, and selectivity constraints of relational databases, this program can help database developers and researchers create more reliable and realistic database models for their applications. As future work, the program could be extended to include more advanced features, such as the ability to incorporate domain-specific knowledge or to have a more interactive user interface. In summary, this program is a valuable tool that can improve the quality and reliability of database systems.

6 Bibliography

1. Hamilton, R. (2016, February 8). Data mocking - a way to test the untestable. Scott Logic: Altogether Smarter. Retrieved April 3, 2023, from https://blog.scottlogic.com/2016/02/08/data-mocking.html

2. Leung, K. (2021, September 23). Free Resources for generating Realistic Fake Data. Medium. Retrieved April 3, 2023, from https://towardsdatascience.com/free-resources-for-generating-realistic-fake-data-da63836be1a8

3. Mockaroo APIs and Documentation. Mockaroo. (n.d.). Retrieved April 3, 2023, from https://www.mockaroo.com/docs

4. Faker 18.3.2 documentation. (2022, April 20). Retrieved April 3, 2023, from https://faker.readthedocs.io/en/master/

5. Mancas, Christian. (2015). On the Paramount Importance of Database Constraints. Journal of Information Technology & Software Engineering. 05. 10.4172/2165-7866.1000e125.