Programming Project 01 -Documentation

5300 – Database Systems

Group Members-
Prasanna Kancharla
D Shilpa Reddy
Rajlakshmi Maurya
Monica Gungi
Shreyas Shekar Koushik

1.**Objective**

The project's primary goal is to develop a robust program that accepts a dataset and its associated functional dependencies. The core function of the program is to methodically normalize the input relations in accordance with the given functional dependencies, enhancing data integrity and minimizing redundancy. As a part of the normalization process, the program is engineered to automatically generate SQL queries that facilitate the creation of structured, optimized database tables.

2.**Introduction**

The Database Normalization Code is designed to assist users in organizing and optimizing their database structures. It takes a dataset (relation) and a set of functional dependencies as input, offering database normalization and SQL query generation based on the provided dependencies.

3. **System Overview**

The Database Normalization Code consists of four core components: the Input Parser, Normalizer, SQL Query Generator, and Normal Form Finder. These components collaborate to transform unnormalized data into a structured and efficient database.

4. Developer Guide

4.1. Code Structure

The code focuses on parsing the input dataset (relation) and functional dependencies responsible for the normalization of the dataset up to the desired normal form and generating the corresponding SQL queries. Below is the decoded clear explanation of the code where it is explained in 2 halves.

Part 1: Input Parsing and Functional Dependencies Identification

- **Import Statements and Class Definitions**

  The code begins with Python import statements that import necessary libraries such as csv for reading CSV files and sys for accessing system-specific parameters and functions.

Classes are defined to represent Attributes and Functional Dependencies (FDs). The Attribute class encapsulates the properties of a database attribute, while the FunctionalDependency class represents an FD, holding both the determinant and the dependent as sets of Attribute objects.

- **Reading the Input**
  The csv module is utilized to read input CSV files, which contain the dataset to be normalized. The data is read row-by-row and stored for further processing.
  Functional dependencies are read from a text file, where each line represents a single FD. The FDs are parsed and stored as instances of the FunctionalDependency class.
- **Preprocessing and Validation**
  After the initial parsing, the code performs preprocessing steps to ensure data consistency and correctness. This includes validating the format of the input and handling potential errors in the dataset or functional dependencies.

Part 2: Normalization and SQL Query Generation

- **Normalization Functions**
  For each normal form, specific functions are implemented to perform the necessary transformations. The code abstracts the normalization rules into these functions, which progressively decompose the relation into tables that satisfy the conditions of 2NF, 3NF, and BCNF. Each function accounts for the removal of partial, transitive, and other types of dependencies that violate the rules of the respective normal form.
- **SQL Query Generation**
  After normalization, the code generates SQL **CREATE TABLE** statements corresponding to each table in the normalized form. This includes the definition of primary keys and data types inferred from the attributes. The generation of SQL is dynamic, and it ensures that the output can be directly executed in a relational database management system (RDBMS).

4.2. Functionality

This project relies on the following Python libraries:

ordered_set: Utilized for maintaining ordered sets.

collections: Employed for working with ordered dictionaries.

CSV: Used for parsing CSV files.

CSV Parsing: The script parses the input dataset stored in a CSV file and stores it in a Python dictionary.

Functional Dependency Parsing: It also parses the functional dependencies from a text file. Functional dependencies are provided in the form of strings (e.g., "A, B -> C"), where A and B together determine C.
MVD Parsing : For handling 4NF and 5NF, the script can parse Multivalued Dependencies (MVDs) from a

text file. MVDs are in the format "A, B ->> C," indicating that A and B together determine multiple values of C.

4.3 Usage and Flow of the Code

Initialization and Setup

Parsing Data: The code initializes by parsing input data and functional dependencies from CSV (exampleInputTable2.csv) and text files (fd2.txt) respectively. This step involves data ingestion and schema interpretation.

CSV Parsing: read_input_table(): Reads CSV file line by line and converts it into a dictionary where keys are column headers and values are lists containing column data.

Functional Dependencies (FDs) Parsing: parse_functional_dependencies(): Interprets the FDs defined in a text file into a data structure (likely a set or list of tuples) for algorithmic processing.

User Input: input(): Requests the highest normal form to decompose the dataset into. This is an interface for user-driven specification of database design objectives.

Decomposition Process

The code appears to use a not-shown function get_normal_form_choice() to decompose into the desired normal form. The decomposition process is central to database normalization, reducing redundancy, and preventing update anomalies.

Normalization:

The script offers functionality to normalize the dataset to 1NF, 2NF, 3NF, BCNF, 4NF. . It decomposes tables based on functional dependencies and provides the option to work with composite keys.These algorithms iteratively remove partial, transitive, and join dependencies, as well as multivalued and join dependencies to achieve higher normal forms.

Multivalued Dependencies (MVDs): For 4NF and 5NF, the code handles MVDs, which are dependencies where one attribute determines multiple disjoint attributes.

Normalization Functions:

- Generate_3NF_queries(): It encapsulates the logic for transforming relations into Third Normal Form by eliminating transitive dependencies.
- is_bcnf(): It is a predicate function that tests whether a relation meets the Boyce-Codd Normal Form (BCNF) conditions, which are stricter than 3NF.
- decompose_to_bcnf(): It embodies the algorithm for decomposing relations into BCNF, resolving anomalies that 3NF does not address.

- decompose_to_4NF(): It applies the principles for removing MVDs, producing a dataset that is free from non-trivial multivalued dependencies without loss of data.

SQL Query Generation

The code translates the decomposed relations into SQL queries, essential for actual database schema implementation.

generate_bcnf_query(): Constructs SQL CREATE TABLE statements based on the BCNF-decomposed relations.

generate_query(): General function for generating SQL queries, indicating a possible factory or template method pattern for differing SQL generation strategies.

Output and File Management

After processing, the code writes the results into text files for documentation or further manipulation.

Write_to_text_file(): It is a utility function to output the results, likely normalization decisions, or intermediate forms, into a human-readable text file.

write_to_query_file(): It appends the generated SQL queries into an output file which could be used to execute schema changes on a database.

Main Execution Block

The if __name__ == "__main__" block orchestrates the execution flow when the script is run as the main program.

Main Logic: The parsed input is printed for verification. A function get_composite_keys() (not provided) presumably calculates the composite keys for the relation, which are critical for understanding full dependencies. The script processes the input to generate a normalized database schema based on user-specified normal form criteria.It saves the decomposed schema and generates SQL commands to output files for persistence and auditability.

Technical Methodology:

The code employs standard relational database normalization techniques to iteratively decompose a dataset schema.

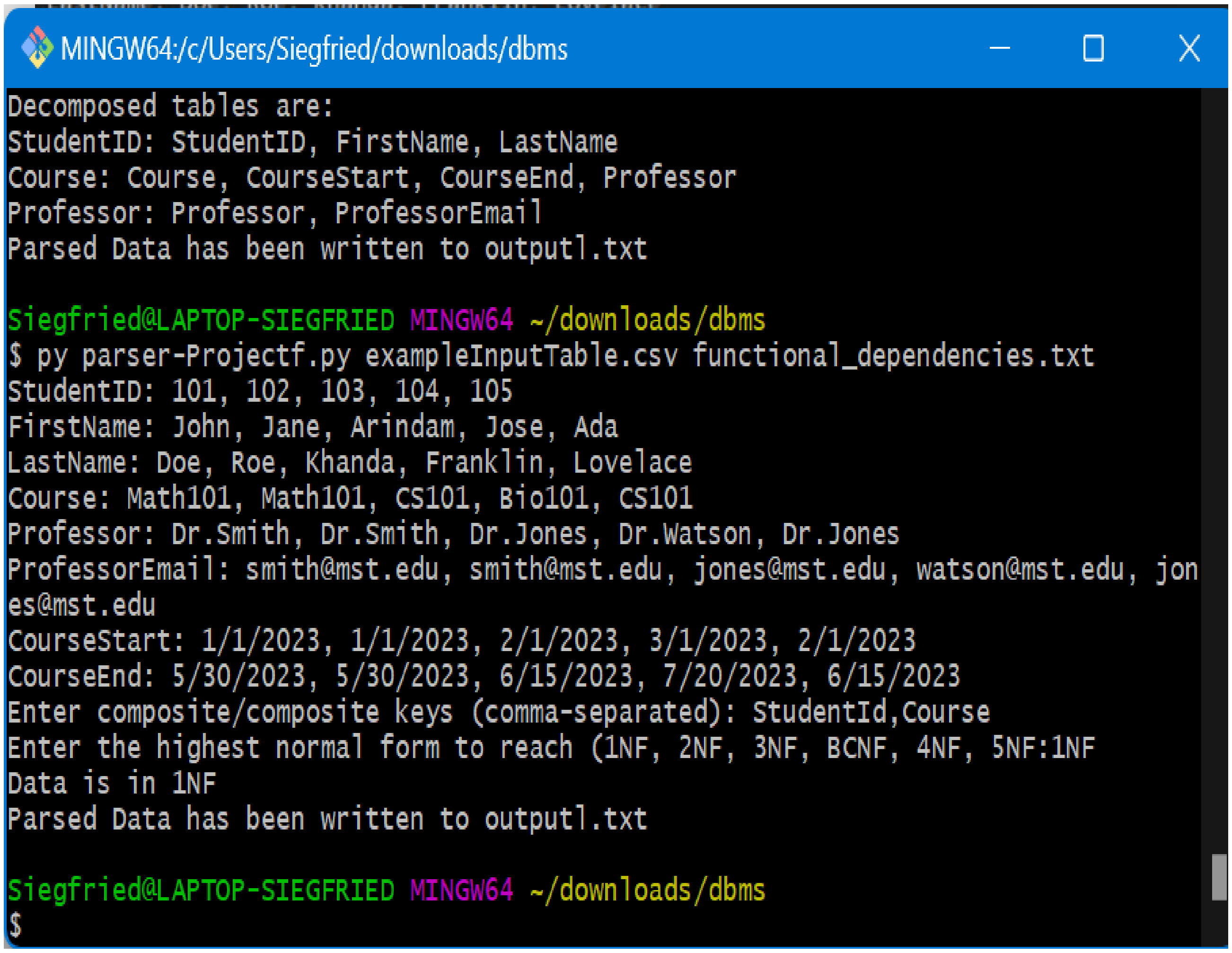
- Elimination of Partial Dependencies: Ensures that non-prime attributes are dependent on the whole of a candidate key, not part of it (2NF).

- Elimination of Transitive Dependencies: Ensures that non-prime attributes are directly dependent on super keys and not on other non-prime attributes (3NF).
- Removal of Functional Dependency Anomalies: Ensures that every determinant is a candidate key (BCNF).
- Elimination of Multivalued Dependencies: Ensures that attributes have a one-to-one relationship within a table unless there is a multivalued dependency (4NF).
- Projection-Join Normalization: Ensures lossless join and dependency preservation in the database schema (5NF).

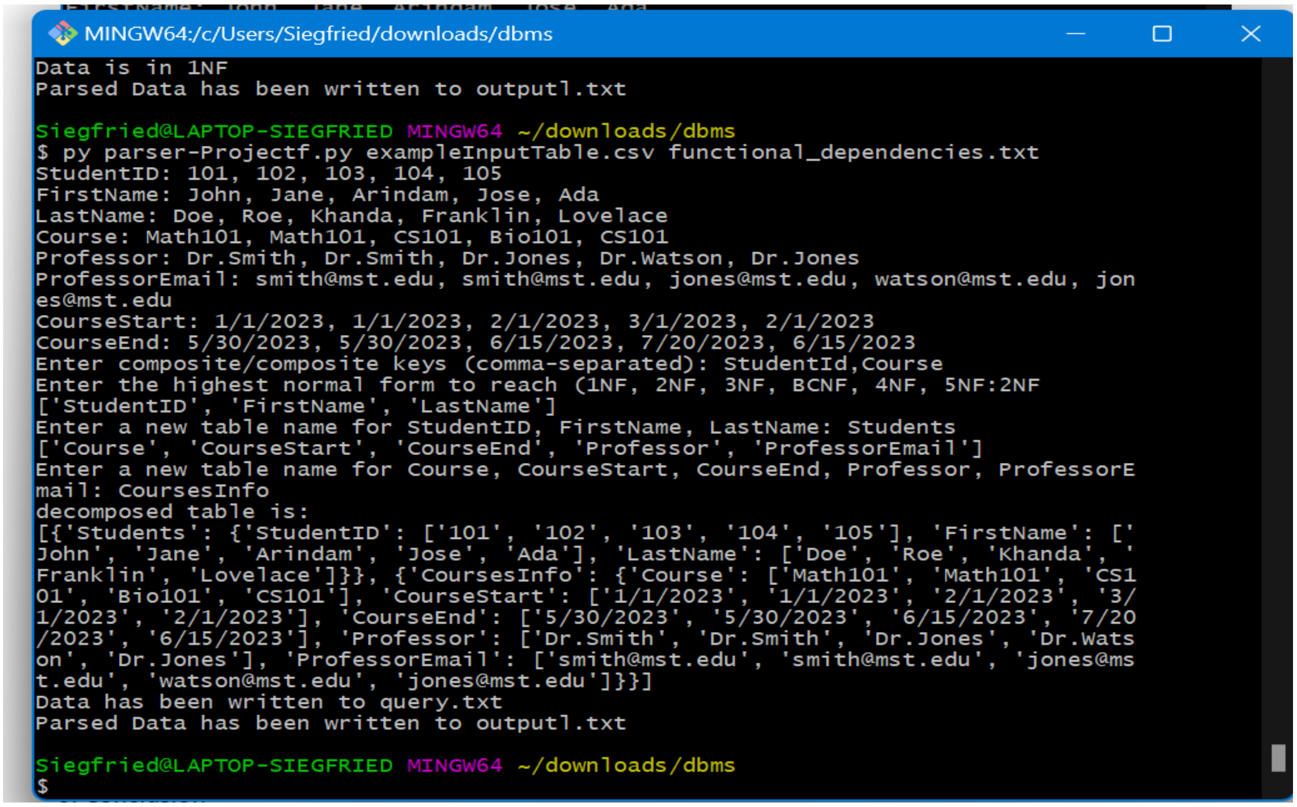
Each of these steps is crucial to reduce redundancy, avoid update anomalies, ensure data integrity, and optimize the performance of the database operations.The technical flow indicates a structured approach to relational database design, translating business rules encapsulated in FDs and MVDs into a well-defined database schema expressed in SQL.

5. Output and Results

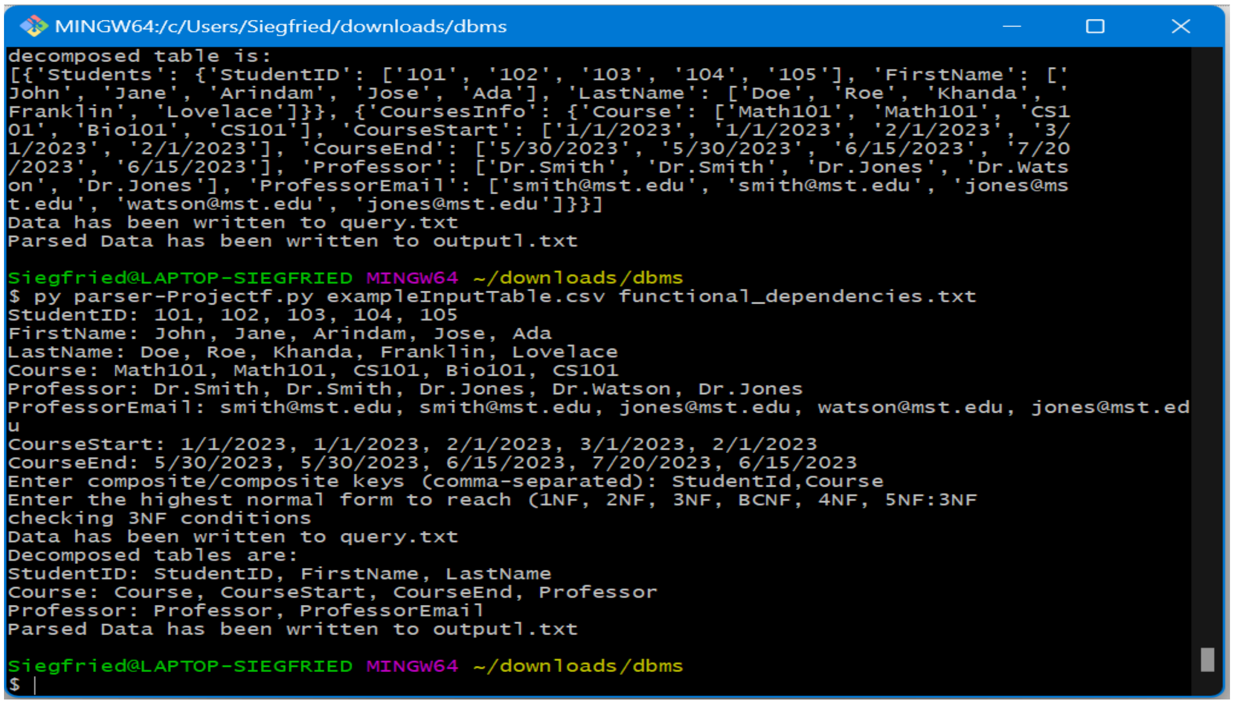
5.1. Normalization


MINGW64:/c/Users/Siegfried/downloads/dbms

```
Decomposed tables are:
StudentID: StudentID, FirstName, LastName
Course: Course, CourseStart, CourseEnd, Professor
Professor: Professor, ProfessorEmail
Parsed Data has been written to output1.txt

Siegfried@LAPTOP-SIEGFRIED MINGW64 ~/downloads/dbms
$ py parser-Projectf.py exampleInputTable.csv functional_dependencies.txt
StudentID: 101, 102, 103, 104, 105
FirstName: John, Jane, Arindam, Jose, Ada
LastName: Doe, Roe, Khanda, Franklin, Lovelace
Course: Math101, Math101, CS101, Bio101, CS101
Professor: Dr.Smith, Dr.Smith, Dr.Jones, Dr.Watson, Dr.Jones
ProfessorEmail: smith@mst.edu, smith@mst.edu, jones@mst.edu, watson@mst.edu, jon
es@mst.edu
CourseStart: 1/1/2023, 1/1/2023, 2/1/2023, 3/1/2023, 2/1/2023
CourseEnd: 5/30/2023, 5/30/2023, 6/15/2023, 7/20/2023, 6/15/2023
Enter composite/composite keys (comma-separated): StudentId,Course
Enter the highest normal form to reach (1NF, 2NF, 3NF, BCNF, 4NF, 5NF:1NF
Data is in 1NF
Parsed Data has been written to output1.txt

Siegfried@LAPTOP-SIEGFRIED MINGW64 ~/downloads/dbms
$
```

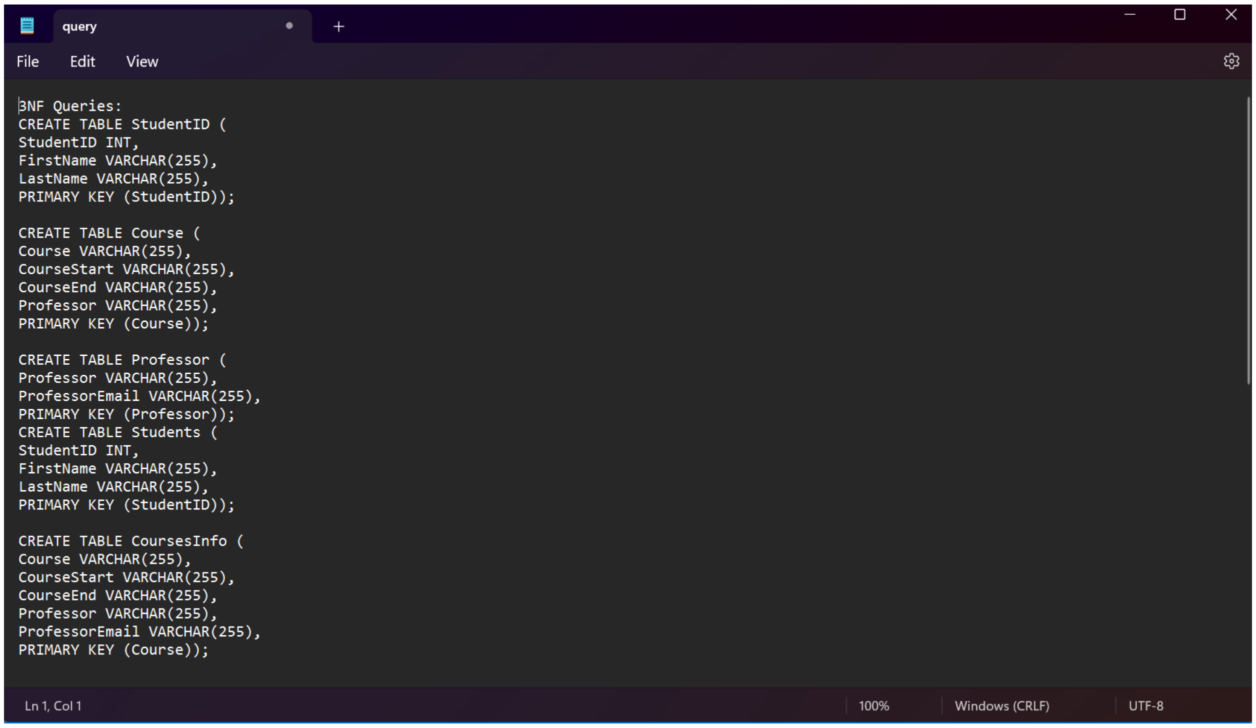


```
Data is in 1NF
Parsed Data has been written to output1.txt

Siegfried@LAPTOP-SIEGFRIED MINGW64 ~/downloads/dbms
$ py parser-Projectf.py exampleInputTable.csv functional_dependencies.txt
StudentID: 101, 102, 103, 104, 105
FirstName: John, Jane, Arindam, Jose, Ada
LastName: Doe, Roe, Khanda, Franklin, Lovelace
Course: Math101, Math101, CS101, Bio101, CS101
Professor: Dr.Smith, Dr.Smith, Dr.Jones, Dr.Watson, Dr.Jones
ProfessorEmail: smith@mst.edu, smith@mst.edu, jones@mst.edu, watson@mst.edu, jon
es@mst.edu
CourseStart: 1/1/2023, 1/1/2023, 2/1/2023, 3/1/2023, 2/1/2023
CourseEnd: 5/30/2023, 5/30/2023, 6/15/2023, 7/20/2023, 6/15/2023
Enter composite/composite keys (comma-separated): StudentId,Course
Enter the highest normal form to reach (1NF, 2NF, 3NF, BCNF, 4NF, 5NF:2NF
['StudentID', 'FirstName', 'LastName']
Enter a new table name for StudentID, FirstName, LastName: Students
['Course', 'CourseStart', 'CourseEnd', 'Professor', 'ProfessorEmail']
Enter a new table name for Course, CourseStart, CourseEnd, Professor, ProfessorE
mail: CoursesInfo
decomposed table is:
[{'Students': {'StudentID': ['101', '102', '103', '104', '105'], 'FirstName': ['
John', 'Jane', 'Arindam', 'Jose', 'Ada'], 'LastName': ['Doe', 'Roe', 'Khanda', '
Franklin', 'Lovelace']}}, {'CoursesInfo': {'Course': ['Math101', 'Math101', 'CS1
01', 'Bio101', 'CS101'], 'CourseStart': ['1/1/2023', '1/1/2023', '2/1/2023', '3/
1/2023', '2/1/2023'], 'CourseEnd': ['5/30/2023', '5/30/2023', '6/15/2023', '7/20
/2023', '6/15/2023'], 'Professor': ['Dr.Smith', 'Dr.Smith', 'Dr.Jones', 'Dr.Wats
on', 'Dr.Jones'], 'ProfessorEmail': ['smith@mst.edu', 'smith@mst.edu', 'jones@ms
t.edu', 'watson@mst.edu', 'jones@mst.edu']}}]
Data has been written to query.txt
Parsed Data has been written to output1.txt

Siegfried@LAPTOP-SIEGFRIED MINGW64 ~/downloads/dbms
$
```



```
decomposed table is:
[{'Students': {'StudentID': ['101', '102', '103', '104', '105'], 'FirstName': ['
John', 'Jane', 'Arindam', 'Jose', 'Ada'], 'LastName': ['Doe', 'Roe', 'Khanda', '
Franklin', 'Lovelace']}}, {'CoursesInfo': {'Course': ['Math101', 'Math101', 'CS1
01', 'Bio101', 'CS101'], 'CourseStart': ['1/1/2023', '1/1/2023', '2/1/2023', '3/
1/2023', '2/1/2023'], 'CourseEnd': ['5/30/2023', '5/30/2023', '6/15/2023', '7/20
/2023', '6/15/2023'], 'Professor': ['Dr.Smith', 'Dr.Smith', 'Dr.Jones', 'Dr.Wats
on', 'Dr.Jones'], 'ProfessorEmail': ['smith@mst.edu', 'smith@mst.edu', 'jones@ms
t.edu', 'watson@mst.edu', 'jones@mst.edu']}}]
Data has been written to query.txt
Parsed Data has been written to output1.txt

Siegfried@LAPTOP-SIEGFRIED MINGW64 ~/downloads/dbms
$ py parser-Projectf.py exampleInputTable.csv functional_dependencies.txt
StudentID: 101, 102, 103, 104, 105
FirstName: John, Jane, Arindam, Jose, Ada
LastName: Doe, Roe, Khanda, Franklin, Lovelace
Course: Math101, Math101, CS101, Bio101, CS101
Professor: Dr.Smith, Dr.Smith, Dr.Jones, Dr.Watson, Dr.Jones
ProfessorEmail: smith@mst.edu, smith@mst.edu, jones@mst.edu, watson@mst.edu, jones@mst.ed
u
CourseStart: 1/1/2023, 1/1/2023, 2/1/2023, 3/1/2023, 2/1/2023
CourseEnd: 5/30/2023, 5/30/2023, 6/15/2023, 7/20/2023, 6/15/2023
Enter composite/composite keys (comma-separated): StudentId,Course
Enter the highest normal form to reach (1NF, 2NF, 3NF, BCNF, 4NF, 5NF:3NF
checking 3NF conditions
Data has been written to query.txt
Decomposed tables are:
StudentID: StudentID, FirstName, LastName
Course: Course, CourseStart, CourseEnd, Professor
Professor: Professor, ProfessorEmail
Parsed Data has been written to output1.txt

Siegfried@LAPTOP-SIEGFRIED MINGW64 ~/downloads/dbms
$ |
```

5.2. Generated Sql Queries



```
3NF Queries:
CREATE TABLE StudentID (
StudentID INT,
FirstName VARCHAR(255),
LastName VARCHAR(255),
PRIMARY KEY (StudentID));

CREATE TABLE Course (
Course VARCHAR(255),
CourseStart VARCHAR(255),
CourseEnd VARCHAR(255),
Professor VARCHAR(255),
PRIMARY KEY (Course));

CREATE TABLE Professor (
Professor VARCHAR(255),
ProfessorEmail VARCHAR(255),
PRIMARY KEY (Professor));
CREATE TABLE Students (
StudentID INT,
FirstName VARCHAR(255),
LastName VARCHAR(255),
PRIMARY KEY (StudentID));

CREATE TABLE CoursesInfo (
Course VARCHAR(255),
CourseStart VARCHAR(255),
CourseEnd VARCHAR(255),
Professor VARCHAR(255),
ProfessorEmail VARCHAR(255),
PRIMARY KEY (Course));
```

6. Conclusion

The code encapsulates a comprehensive normalization toolchain for relational databases, transforming raw datasets into structures adhering to user-specified normal forms ranging from 1NF to 5NF. Through parsing inputs, analyzing functional and multivalued dependencies, and generating SQL creation scripts, it systematically reduces redundancy while maintaining data integrity. Its modular design ensures flexibility and adaptability to different schemas, making it a valuable asset for database design and optimization. The parsed data will be saved in output.txt and can be used for further analysis and derivations of higher Normal forms.