**CS5300: DATABASE SYSTEMS**

**PROGRAMMING PROJECT:**

**RDMS NORMALIZER**

**Objective:**

To develop a program that takes a database (relations) and functional dependencies as input, normalizes the relations based on the provided functional dependencies, produces SQL queries to generate the normalized database tables, and optionally determines the highest normal form of the input table.

**Group Members:**

* **Supraja Yadav Rajaboina - 12619504**
* **Pujitha Ganamukula - 12608977**

**Input –**

1. Database Table (.csv) file path.
2. Functional Dependencies
3. Multi Valued Dependencies
4. Primary Key
5. Choice of the highest normal form to reach (1: 1NF, 2: 2NF, 3: 3NF, 3.5: BCNF, 4: 4NF, 5: 5NF).

**Output –**

1. If the input table is not in 2NF then the table after converting to 1NF would be written to the **Console** and to **converted2NF.csv** file and similar.
2. **convertedxNF.csv** file which stores the tables generated after being normalized to the xth Normal Form
3. **SQL Queries** after normalization are generated and output to the Console.

**Requirements –**

1. Database Dataset Table must be stored in a **.csv** File.

The entire project can be divided into 3 stages:

Stage 1: Taking inputs from the user.

Stage 2: Converting the table to the target Normal Form as per user request.

Stage 3: Writing the output to **convertedxNF.csv** file and generating **SQL Queries** to the console.

**Stage 1: Taking Inputs**

The code initially asks the user for inputs.

1. Prompting for the CSV File Path:

* We prompt the user to provide the path to the CSV file that contains the relational data.
* This file will be read into a DataFrame, forming the base table we aim to normalize.
* Loaded as a DataFrame **(df)** using **pd.read\_csv()**

1. Prompting for the Primary Key:

* The user provides the primary key columns, separated by commas.
* These are crucial for defining how data is uniquely identified in the table.
* This is given in a format separated by commas (M, N, etc)
* The primary key columns are stored as a list of strings **(primary\_key\_columns)**, which is obtained by splitting the **primary\_key** input on commas and removing extra whitespace.

1. Target Normal Form:

* The target normalization form specifies the desired level of normalization (e.g., 1NF, 2NF, 3NF, etc.).
* This guides the normalization process, allowing us to stop once the data reaches the specified form.
* It is stored as an integer in the **target\_nf** variable.

1. Functional Dependencies (FDs):

* Functional dependencies represent the relationships between columns.
* The user inputs FDs in the format "A -> B; C,D -> E", where "A -> B" signifies that B is functionally dependent on A, meaning A determines B.
* The FDs are collected as a string from the user
* It is then parsed by **parse\_functional\_dependencies**, which splits each dependency by **->** to create a list of tuples, where each tuple is a dependency pair (LHS and RHS).
* Then, each dependency is stored as a tuple of lists in **fds**

1. Multi-Valued Dependencies (MVDs):
   * If the target NF is 4 or higher, we prompt for MVDs.
   * An MVD, denoted as A ->> B, indicates that B is multi-valued with respect to A.
   * This is particularly relevant for 4NF and 5NF.
   * We parse MVDs similarly to FDs, storing them in a structured list.
   * Similar to FDs, MVDs are collected as a string and parsed by **parse\_multivalued\_dependencies**
   * Each MVD is stored as a tuple of lists in **mvd**

**Stage 2: Converting the Table to the Target Normal Form**

* When the user specifies the target NF as an integer (1, 2, 3, 3.5, 4, or 5), this value becomes a key control parameter that directs the **normalize\_csv** function.
* This input specifies how far the program should go in the normalization process, from ensuring basic atomicity (1NF) to eliminating join dependencies (5NF).
* Each normalization level (NF) builds upon the previous one, meaning a higher NF automatically requires compliance with all lower NFs.
* The conditional branching is processed as follows:
  1. 1: 1NF
  2. 2: 2NF
  3. 3: 3NF
  4. 3.5: BCNF
  5. 4: 4NF
  6. 5: 5NF
* The target NF input is received and stored as an integer in **target\_nf**, which is then passed to the **normalize\_csv** function.
* Within **normalize\_csv**, this input drives the conditional execution of specific normalization functions, effectively controlling the flow of the entire process.

target\_nf = int(input("Enter the target normal form (1, 2, 3, 3.5, 4, or 5): "))

normalize\_csv(input\_csv, output\_2NF, output\_3NF, output\_bcnf, output\_4nf, output\_5nf, primary\_key, target\_nf)

* The **normalize\_csv** function contains conditional checks (if **target\_nf >= x**) for each NF level. These checks ensure that only the necessary steps are executed up to the specified target NF.
* **1NF:**
  + Ensure atomicity by making each cell contain only single values, removing any multi-valued attributes.

if target\_nf >= 1:

df\_1nf, decomposed\_tables\_1nf, converted\_to\_1nf = check\_and\_convert\_to\_1nf(df, primary\_key\_columns, fds)

* + This conditional statement executes if **target\_nf >= 1**
    - **check\_1NF**: Checks if each cell contains a single, atomic value.
    - **decompose\_multivalued\_attributes**: If multi-valued attributes are found, they are decomposed into separate tables.
    - **check\_and\_convert\_to\_1nf**: Wraps the above functions, returning the main table in 1NF and any additional tables with decomposed attributes.
* **2NF:** 
  + Eliminate partial dependencies, where non-key columns depend only on part of a composite primary key.

if target\_nf >= 2:

tables\_in\_2NF\_result = convert\_to\_2NF(table\_df, ",".join(pk), relevant\_fds, output\_2NF)

* + This conditional statement executes if **target\_nf >= 2**
    - **find\_partial\_dependencies**: Identifies FDs where non-key attributes are only partially dependent on the primary key.
    - **convert\_to\_2NF**: Uses identified partial dependencies to create new tables, ensuring non-key attributes fully depend on the primary key.
* **3NF:**
  + Eliminate transitive dependencies, where non-key attributes depend on other non-key attributes

if target\_nf >= 3:

tables\_in\_3NF = convert\_to\_3NF(table\_df, ",".join(pk), relevant\_fds, output\_3NF, tables\_2nf\_dict)

* + This conditional statement executes if **target\_nf >= 3**
    - **find\_transitive\_dependencies**: Identifies FDs where non-key attributes determine other non-key attributes, indicating transitive dependencies.
    - **convert\_to\_3NF**: Uses identified transitive dependencies to create new tables, ensuring all attributes directly depend only on the primary key.
* **BCNF:**
  + Ensure that every determinant in an FD is a superkey

if target\_nf >= 3.5:

tables\_in\_bcnf = check\_and\_convert\_to\_bcnf(tables\_in\_3NF, relevant\_fds, output\_bcnf)

* + This conditional statement executes if **target\_nf >= 3.5**
    - **is\_in\_bcnf**: Verifies if each determinant (left-hand side of FD) is a superkey.
    - **check\_and\_convert\_to\_bcnf**: Uses non-superkey determinants to decompose tables, ensuring BCNF compliance.
* **4NF:**
  + Eliminate multi-valued dependencies (MVDs) unless the left-hand side is a superkey.

if target\_nf >= 4:

mvd\_input = input("Enter the multi-valued dependencies (use 'A ->> B' format, separated by semicolons): ")

mvds = parse\_multivalued\_dependencies(mvd\_input)

tables\_in\_4nf = decompose\_to\_4nf(tables\_in\_bcnf, mvds, output\_4nf)

* + This conditional statement executes if **target\_nf >= 4**
    - **is\_in\_4nf**: Checks if MVDs are correctly structured so that each LHS is a superkey.
    - **decompose\_to\_4nf**: Uses MVDs to decompose tables, removing independent multi-valued attributes that create redundancy.
* **5NF:**
  + Eliminate join dependencies, ensuring no redundancy in projections

if target\_nf == 5:

tables\_in\_5nf = []

for table\_name, table, pk\_columns in tables\_in\_4nf:

result\_tables, sql\_queries = check\_and\_convert\_to\_5nf(table, pk\_columns)

tables\_in\_5nf.extend(result\_tables)

* + This conditional statement executes if **target\_nf == 5**
    - **find\_join\_dependencies\_and\_check\_5nf**: Identifies join dependencies and checks if further decomposition is required to eliminate redundancy.
    - **check\_and\_convert\_to\_5nf**: Decomposes tables based on join dependencies, creating projections as separate tables.
* The **normalize\_csv** function controls the conditional execution of each normalization function based on the **target\_nf** input, with each step addressing a specific type of dependency violation.

1. **check\_1NF:**

def check\_1NF(df):

for col in df.columns:

for value in df[col]:

if isinstance(value, str) and ',' in value:

return False # Not in 1NF

return True # In 1NF

* 1. Determines if a table is in 1NF by checking if any column contains multiple values within a single cell.
     1. **Column Scan**: Iterates over each column, checking if values contain commas (indicating multiple values).
     2. **Returns Boolean**: Returns **False** if a multi-valued attribute is detected; otherwise, it returns **True**.

1. **check\_and\_convert\_to\_1nf:**

def check\_and\_convert\_to\_1nf(df, primary\_key\_columns, fds):

if check\_1NF(df):

print("Data is already in 1NF.")

return df, {}, False

print("Converting to 1NF by decomposing multi-valued attributes.")

decomposed\_tables = decompose\_multivalued\_attributes(df, fds)

main\_table\_columns = [col for col in df.columns if col not in decomposed\_tables.keys()]

df\_1nf = df[main\_table\_columns].drop\_duplicates().reset\_index(drop=True)

return df\_1nf, decomposed\_tables, True

* 1. Checks if a DataFrame is in 1NF and decomposes it if necessary.
     1. **Checks 1NF Compliance**: Calls **check\_1NF** to verify 1NF compliance.
     2. **Decomposes Multi-Valued Attributes**: If not in 1NF, it decomposes the DataFrame using **decompose\_multivalued\_attributes**.
     3. **Returns Updated Data**: Returns the main table in 1NF, any decomposed tables, and a boolean indicating if conversion was applied.

1. **convert\_to\_2NF:**

def convert\_to\_2NF(df, primary\_key, fds, output\_csv):

primary\_key\_columns = [col.strip() for col in primary\_key.split(',')]

partial\_dependencies = find\_partial\_dependencies(fds, primary\_key\_columns)

tables\_in\_2NF = []

for lhs, rhs in partial\_dependencies:

table\_name = '\_'.join(lhs) + "\_relation"

new\_table = df[lhs + rhs].drop\_duplicates()

tables\_in\_2NF.append((table\_name, new\_table, lhs))

print(f"Generated SQL for 2NF Partial Dependency Table: {table\_name}")

print(generate\_sql\_query(table\_name, new\_table.columns, lhs))

moved\_columns = {col for \_, table, \_ in tables\_in\_2NF for col in table.columns}

remaining\_columns = [col for col in df.columns if col in primary\_key\_columns or col not in moved\_columns]

if remaining\_columns:

main\_table\_name = f"{'\_'.join(primary\_key\_columns)}\_main\_relation"

df\_2NF = df[remaining\_columns].drop\_duplicates()

tables\_in\_2NF.append((main\_table\_name, df\_2NF, primary\_key\_columns))

print(generate\_sql\_query(main\_table\_name, df\_2NF.columns, primary\_key\_columns))

return tables\_in\_2NF

* 1. Converts a DataFrame to 2NF by removing partial dependencies.
     1. **Identifies Partial Dependencies**: Calls **find\_partial\_dependencies** to identify partial dependencies.
     2. **Decomposes Table**: Creates separate tables for each partial dependency.
     3. **Returns 2NF Tables**: Returns a list of tables in 2NF, including the main table and tables for each partial dependency.

1. **convert\_to\_3NF:**

def convert\_to\_3NF(df, primary\_key, fds, output\_csv, tables\_in\_2NF):

primary\_key\_columns = [col.strip() for col in primary\_key.split(',')]

transitive\_dependencies = find\_transitive\_dependencies(fds, primary\_key\_columns)

tables\_in\_3NF = []

for table\_name, table, pk\_columns in tables\_in\_2NF:

moved\_columns = set()

for lhs, rhs in transitive\_dependencies:

if set(lhs).issubset(table.columns) and any(col in table.columns for col in rhs):

transitive\_table\_name = '\_'.join(lhs) + "\_relation"

transitive\_table = table[lhs + rhs].drop\_duplicates()

tables\_in\_3NF.append((transitive\_table\_name, transitive\_table, lhs))

moved\_columns.update(rhs)

remaining\_columns = [col for col in table.columns if col not in moved\_columns]

if remaining\_columns:

final\_relation\_table = table[remaining\_columns].drop\_duplicates()

tables\_in\_3NF.append((table\_name, final\_relation\_table, pk\_columns))

return tables\_in\_3NF

* 1. Converts a DataFrame to 3NF by removing transitive dependencies. Transitive dependencies exist when a non-key attribute depends on another non-key attribute rather than directly on the primary key.
     1. **Identifies Transitive Dependencies**: Uses **find\_transitive\_dependencies** to find dependencies where a non-key attribute depends on another non-key attribute.
     2. **Decomposes Each Dependency**: For each transitive dependency, it creates a new table with the left-hand side (LHS) and right-hand side (RHS) attributes of the dependency.
     3. **Updates Main Table**: After extracting transitive dependencies, the original table is updated to remove these dependencies, ensuring that all remaining columns depend directly on the primary key.
     4. **Returns 3NF Tables**: It returns a list of tables in 3NF, including both the main table and any tables created for transitive dependencies.

1. **is\_in\_bcnf:**

def is\_in\_bcnf(table, fds, pk\_columns):

for lhs, rhs in fds:

lhs\_is\_superkey = set(pk\_columns).issubset(lhs) or set(lhs) == set(pk\_columns)

if not lhs\_is\_superkey:

return False

return True

* 1. Checks if a table is in **BCNF (Boyce-Codd Normal Form)**.
  2. In BCNF, every determinant (LHS of an FD) must be a superkey, meaning that it uniquely identifies each row.
     1. **Superkey Check**: For each FD, it checks if the LHS is a superkey by comparing it to the primary key columns.
     2. **Returns Boolean**: If any FD has a determinant that is not a superkey, the function returns **False**, indicating that the table violates BCNF. If all FDs are superkey-determined, it returns **True**.

1. **check\_and\_convert\_to\_bcnf:**

def check\_and\_convert\_to\_bcnf(tables\_in\_3NF, fds, output\_sql\_file="bcnf\_queries.sql"):

tables\_in\_bcnf = []

with open(output\_sql\_file, 'w') as sql\_file:

for table\_name, table, pk\_columns in tables\_in\_3NF:

relevant\_fds = [(lhs, rhs) for lhs, rhs in fds if set(lhs).issubset(table.columns)]

if is\_in\_bcnf(table, relevant\_fds, pk\_columns):

tables\_in\_bcnf.append((table\_name, table, pk\_columns))

sql\_query = generate\_sql\_query(table\_name, table.columns, pk\_columns)

sql\_file.write(sql\_query + "\n\n")

else:

for lhs, rhs in relevant\_fds:

bcnf\_table\_name = '\_'.join(lhs) + "\_BCNF\_relation"

decomposed\_table = table[lhs + rhs].drop\_duplicates()

tables\_in\_bcnf.append((bcnf\_table\_name, decomposed\_table, lhs))

sql\_query = generate\_sql\_query(bcnf\_table\_name, decomposed\_table.columns, lhs)

sql\_file.write(sql\_query + "\n\n")

return tables\_in\_bcnf

* 1. Converts a table to BCNF by decomposing any FDs where the LHS is not a superkey. This ensures that every determinant is a superkey.
     1. **Identifies BCNF Violations**: Uses **is\_in\_bcnf** to check if each table is in BCNF.
     2. **Decomposes Tables**: For any table with FDs that violate BCNF, it creates a new table containing the LHS and RHS of the violating FD.
     3. **Generates SQL**: Writes **CREATE TABLE** statements for each decomposed table into a **.sql** file.
     4. **Returns BCNF Tables**: It returns a list of tables in BCNF, including both compliant tables and any newly created tables to resolve BCNF violations.

1. **is\_in\_4nf:**

def is\_in\_4nf(table, mvds, pk\_columns):

for lhs, rhs in mvds:

if not set(lhs).issuperset(pk\_columns) and validate\_mvd(table, lhs, rhs):

return False

return True

* 1. Checks if a table is in **4NF (Fourth Normal Form)**. In 4NF, there should be no multi-valued dependencies unless the LHS is a superkey.
     1. **Validates MVDs**: For each MVD, it uses **validate\_mvd** to check if the MVD holds.
     2. **Superkey Check**: If the LHS of the MVD is not a superkey and the MVD holds, this violates 4NF.
     3. **Returns Boolean**: Returns **True** if the table is in 4NF; otherwise, it returns **False**.

1. **decompose\_to\_4nf:**

def decompose\_to\_4nf(tables\_in\_bcnf, mvds, output\_sql\_file="4nf\_queries.sql"):

tables\_in\_4nf = []

with open(output\_sql\_file, 'w') as sql\_file:

for table\_name, table, pk\_columns in tables\_in\_bcnf:

relevant\_mvds = [(lhs, rhs) for lhs, rhs in mvds if set(lhs).issubset(table.columns)]

if is\_in\_4nf(table, relevant\_mvds, pk\_columns):

tables\_in\_4nf.append((table\_name, table, pk\_columns))

sql\_query = generate\_sql\_query(table\_name, table.columns, pk\_columns)

sql\_file.write(sql\_query + "\n\n")

else:

for lhs, rhs in relevant\_mvds:

new\_table\_name = '\_'.join(lhs) + "\_4NF\_relation"

decomposed\_table = table[lhs + rhs].drop\_duplicates()

tables\_in\_4nf.append((new\_table\_name, decomposed\_table, lhs))

sql\_query = generate\_sql\_query(new\_table\_name, decomposed\_table.columns, lhs)

sql\_file.write(sql\_query + "\n\n")

return tables\_in\_4nf

* 1. Decomposes a table to 4NF by separating any MVDs where the LHS is not a superkey.
     1. **Identifies 4NF Violations**: Uses **is\_in\_4nf** to check if the table violates 4NF.
     2. **Decomposes MVDs**: For each MVD violation, it creates a new table containing the LHS and RHS of the MVD.
     3. **Generates SQL**: Writes **CREATE TABLE** statements for each decomposed table into a **.sql** file.
     4. **Returns 4NF Tables**: Returns a list of tables in 4NF, including both compliant tables and any newly created tables for MVD violations.

1. **find\_join\_dependencies\_and\_check\_5nf:**

def find\_join\_dependencies\_and\_check\_5nf(df):

dp\_results = {}

inferred\_jds = []

attributes = df.columns.tolist()

for subset\_size in range(2, len(attributes)):

for subset in combinations(attributes, subset\_size):

subset\_key = tuple(sorted(subset))

remaining\_attributes = list(set(attributes) - set(subset))

if set(subset).intersection(remaining\_attributes):

projection\_dfs = [df[list(projection)].drop\_duplicates() for projection in [subset, remaining\_attributes]]

dp\_results[subset\_key] = can\_reconstruct(df, projection\_dfs)

if dp\_results[subset\_key]:

inferred\_jds.append((list(subset), remaining\_attributes))

is\_5nf = all(dp\_results[key] for key in dp\_results if dp\_results[key])

return is\_5nf, inferred\_jds

* 1. Identifies join dependencies (JDs) in a table, which cause redundancy when joining projections, and checks if the table is in **5NF (Fifth Normal Form)**
     1. **Subsets of Attributes**: Examines all attribute subsets to identify potential JDs.
     2. **Checks for Redundancy**: For each JD, it projects subsets of attributes and tests if joining them recreates the original table without redundancy.
     3. **Returns JD Information**: If the table is not in 5NF, it returns the inferred JDs for further decomposition.

1. **decompose\_to\_5nf:**

def decompose\_to\_5nf(df, inferred\_jds, original\_pk):

decomposed\_tables = []

for left, right in inferred\_jds:

projection\_df = df[left + right].drop\_duplicates()

pk\_for\_projection = left if set(left).issuperset(set(original\_pk)) else original\_pk

decomposed\_tables.append((left + right, projection\_df, pk\_for\_projection))

print(f"Decomposed projection for JD {left} -> {right}:\n{projection\_df}\n")

return decomposed\_tables

* 1. Decomposes tables to 5NF by creating separate projections based on identified JDs, ensuring that no redundancy exists from joining subsets.
     1. **Creates Projections**: For each JD, it creates a new table containing the projection of attributes based on the JD.
     2. **Adds Primary Key**: Determines an appropriate primary key for each projection.
     3. **Returns 5NF Tables**: Returns a list of decomposed tables that are in 5NF.

**Stage 3: Writing Output and Generating SQL Queries**

* After the normalization process completes up to the specified target NF, the program will:
  + **Save each decomposed table** to CSV files, clearly indicating which normal form each table meets.
  + **Generate SQL CREATE TABLE statements** for each table, specifying the structure (columns and primary keys) of each normalized table.
* These outputs provide the user with:
  + **Data Storage**: CSV files that contain each normalized table.
  + **Database Creation**: SQL queries that can be used to recreate the schema in a relational database.
* **Saving Decomposed Tables to CSV Files:**
  + Each normalization function in Stage 2 saves its decomposed tables to a CSV file labeled with the target normal form. For example:
    - **output\_2NF**: Stores the tables decomposed into 2NF.
    - **output\_3NF**: Stores the tables decomposed into 3NF.
    - **output\_bcnf**: Stores tables decomposed to BCNF.
    - **output\_4nf**: Stores tables decomposed to 4NF.
    - **output\_5nf**: Stores tables decomposed to 5NF.
* What’s Stored in Each output File?
  + Each output file for a given NF contains:
    - **Table Names**: A header with the name of each decomposed table.
    - **Table Data**: The columns and rows of each table, saved using **table.to\_csv**.
    - **Primary Key Columns**: These columns are preserved for each table, ensuring that each decomposed table retains a primary key.
* This structure makes it easy to visually inspect the decomposed tables and provides ready-to-use data files for importing into a relational database or further analysis.
* Example: **convert\_to\_2nf**

def convert\_to\_2NF(df, primary\_key, fds, output\_csv):

primary\_key\_columns = [col.strip() for col in primary\_key.split(',')]

partial\_dependencies = find\_partial\_dependencies(fds, primary\_key\_columns)

tables\_in\_2NF = []

for lhs, rhs in partial\_dependencies:

table\_name = '\_'.join(lhs) + "\_relation"

new\_table = df[lhs + rhs].drop\_duplicates()

tables\_in\_2NF.append((table\_name, new\_table, lhs))

print(f"Generated SQL for 2NF Partial Dependency Table: {table\_name}")

print(generate\_sql\_query(table\_name, new\_table.columns, lhs))

moved\_columns = {col for \_, table, \_ in tables\_in\_2NF for col in table.columns}

remaining\_columns = [col for col in df.columns if col in primary\_key\_columns or col not in moved\_columns]

if remaining\_columns:

main\_table\_name = f"{'\_'.join(primary\_key\_columns)}\_main\_relation"

df\_2NF = df[remaining\_columns].drop\_duplicates()

tables\_in\_2NF.append((main\_table\_name, df\_2NF, primary\_key\_columns))

print(generate\_sql\_query(main\_table\_name, df\_2NF.columns, primary\_key\_columns))

return tables\_in\_2NF

# Write 2NF tables to the output CSV

with open(output\_csv, 'w') as f:

f.write("=== 2NF Normalized Tables ===\n\n")

for name, table, pk in tables\_in\_2NF:

f.write(f"=== {name} ===\n") table.to\_csv(f, index=False)

f.write("\n\n")

* **Generating SQL Queries for Table Creation**
  + SQL **CREATE TABLE** statements are generated for each decomposed table using the generate\_sql\_query function. Each SQL statement specifies:
    - **Table Name**: The name of the table, which often reflects the columns that were decomposed.
    - **Column Definitions**: Each column in the table, typically set to **VARCHAR(255)** for simplicity.
    - **Primary Key Constraints**: A primary key constraint that enforces the unique identifier(s) for each table.
  + This function constructs SQL statements by iterating over the columns and adding a primary key constraint at the end:

def generate\_sql\_query(table\_name, columns, primary\_key\_columns=None):

sql\_query = f"CREATE TABLE {table\_name} (\n"

for col in columns:

sql\_query += f" {col} VARCHAR(255),\n"

if primary\_key\_columns:

pk\_str = ", ".join(primary\_key\_columns)

sql\_query += f" PRIMARY KEY ({pk\_str}),\n"

return sql\_query.rstrip(',\n') + "\n);"

* **Where SQL Statements Are Generated**
  + For each normalization function, SQL statements are printed to the console and, in certain cases, saved to an SQL file.
  + After decomposing a table to 2NF, **generate\_sql\_query** is called for each table to print the SQL statement:

for name, table, pk in tables\_in\_2NF:

print(f"Generated SQL for 2NF Partial Dependency Table: {name}")

print(generate\_sql\_query(name, table.columns, pk))

* + The SQL output serves as a blueprint for reconstructing the normalized tables in a relational database. Each table’s SQL query includes the **CREATE TABLE** command, column definitions, and the primary key constraint, ensuring that:
    - **Column Structure**: The table’s columns and data types are defined.
    - **Primary Keys**: Each table enforces primary keys, essential for maintaining referential integrity.
* **Conditional Execution Based on Target NF**
  + Each normalization level generates its own SQL queries and CSV files, and this process is controlled by the **target\_nf** input. The program conditionally saves each normalized table only up to the specified target NF. This prevents unnecessary decompositions or output generation for higher NFs than needed.
  + In **normalize\_csv**, tables are saved only up to the target NF, using conditional statements to control the flow.
  + If **target\_nf = 2**:
    - Only 1NF and 2NF tables are saved to CSV and SQL.
    - Higher NFs (3NF, BCNF, etc.) are not processed, as they exceed the target NF.
  + If **target\_nf = 4**:
    - The program saves tables normalized up to 4NF, including 1NF, 2NF, 3NF, and BCNF tables.
    - The program skips 5NF processing since it exceeds the specified target.
  + Example:
    - if target\_nf >= 1:

# Decompose to 1NF and save output

* + - if target\_nf >= 2:

# Decompose to 2NF and save output

* + - if target\_nf >= 3:

# Decompose to 3NF and save output

* + - if target\_nf >= 3.5:

# Decompose to BCNF and save output

* + - if target\_nf >= 4:

# Decompose to 4NF and save output

* + - if target\_nf == 5:

# Decompose to 5NF and save output

* Each normalization step builds upon the previous one, ensuring that data is only normalized and saved up to the specified target NF, avoiding extra processing.
* In conclusion,
  + After Stage 3 completes, the user has:
    - **CSV Files**: Each CSV file (converted2NF.csv, converted3NF.csv, etc.) contains normalized tables up to the specified NF.
    - **SQL Queries**: SQL CREATE TABLE statements for each table are printed to the console and/or saved to SQL files (output\_2NF.sql, output\_3NF.sql, etc.).
    - **Logical and Physical Data Structure**:
      * The normalized data is organized into tables, each enforcing referential integrity through primary keys.
      * SQL queries provide a structured schema for database creation.