DBS Project Code Documentation

Step 1: Inputs

We have created a python code that can normalize until the 5NF.
First we have taken inputs.
Read the table using pandas using:

```
table = pd.read_csv('referenceInputTable.csv')
```

then we took the functional dependencies too read by file these are those fds We have modified the functional dependencies based on the assumptions.
1.Office -> Department
2.Manager -> ManagerEmail
3.SSN -> EmployeeID, FirstName, LastName, JobTitle, Manager, ManagerEmail, Department, PhoneNumber, Office
4.PhoneNumber -> EmployeeID

open ('fds.txt', 'r') -> This line code of reads fds through file instead of user entering each time for consistency
then after it prints fds like this
fds =('Office',): ['Department'], ('Manager',): ['ManagerEmail'], ('SSN',): ['EmployeeID', 'FirstName', 'LastName', 'JobTitle', 'Manager', 'ManagerEmail', 'Department', 'PhoneNumber', 'Office'], ('PhoneNumber',): ['EmployeeID']}

then it prompts users to enter multi valued dependencies
Please enter Multi-Valued Dependencies using format X ->> A, B or X ->> Y:
1.EmployeeID ->> JobTitle
2.Manager ->> Department
3.Manager ->> JobTitle
4.EmployeeID ->> Department
Then after entering these mvds Now enter “exit”.
mvds = { ('EmployeeID'): ['JobTitle', 'Department'],
('Manager'): ['Department', 'JobTitle']}
After this it prompts user to enter options between 1 to 6 for the required normalization form from input tables

Select the highest normal form that the table can achieve (1: 1NF, 2: 2NF, 3: 3NF, 4: BCNF, 5: 4NF, 6: 5NF):
Suppose if user enters 5
Then after again it prompts user if he wants to know highest normal table achieved
Do you want to know the highest normal form of the input table? (1: Yes, 2: No): either yes or no ,If he enters 1
Then again it prompts user to enter primary key
Enter primary keys (for composite keys, separate them with commas): SSN

Step 2: 1NF check

Check for atomicity using following code

```
# Normalize to 1NF
if step >= 1: #here as user entered input is more than 1 so it goes to if loop
```

```
    result = normalizer.transform_to_1NF(table, pk)
#after this step it calls function to check if it in 1NF or not

if relation.empty:
return False
# Ensure each column has a single, consistent data type and no nested structures
for col in relation.columns:
# Check if there is more than one unique type in the column
if relation[col].map(type).nunique() > 1:
return False
# Check if any item in the column is a nested structure (list, dict, set)
if relation[col].apply(lambda item: isinstance(item, (list, dict, set))).any():
return False
# If all checks pass, the relation is in 1NF
return True
```

if it return true then it prints table is in 1NF print("Given input table is already in 1NF.\n")
else it transforms the table into 1NF by following below code function run

```
def transform_to_1NF(relation, pk):
    #Transforms a relation to comply with First Normal Form (1NF) by flattening any nested structures.
        # Iterate through each column to check for nested structures
        for col in relation.columns:
            # If any item in the column is a nested collection, explode the column
            if relation[col].apply(is_nested_collection).any():
                relation = relation.explode(col)  # Flatten the nested structures

        # Print the normalized relation for debugging purposes
        print(f"The relation after transforming into 1NF:\n{relation}\n")
                # Store the normalized relation in the dictionary
        normalized_relations[pk] = relation
        return normalized_relations, is_already_1NF  # Return the normalized relation and the normalization status
```

Step 3: 2NF Check

After transforming the table into 1NF, we proceed to check for the Second Normal Form (2NF). 2NF requires that the relation is already in 1NF and that all non-key attributes are fully functionally dependent on the primary key.
To check for 2NF, we use the following code:
if step >= 2: Proceed if the user selected normalization to 2NF or higher

```
    normalized_result = normalizer.transform_to_2NF(normalized_table_1, pk, fds)
#after this step it calls function to check if it in 2NF or not
def validate_2NF(pk, fds, rel):
    # Collect attributes that are not part of the primary key
```

```
non_primary_attrs = [col for col in rel.columns if col not in pk]

    # Validate the relation against 2NF rules
    for lhs, rhs in fds.items():  # LHS: determinant, RHS: dependents
        if set(lhs).issubset(pk) and set(lhs) != set(pk):
            if any(attr in non_primary_attrs for attr in rhs):
                return False
    return True
```

The validate_2NF function effectively checks for violations of 2NF by:

- Identifying attributes that are not part of the primary key.
- Evaluating each functional dependency to ensure that no non-prime attributes depend on a proper subset of the primary key.
- Returning False if a violation is found or True if the relation satisfies 2NF.

Then if it satisfies 2NF it prints print("Given input table is already in 2NF.\n")
Else it decompose into 2NF

```
    non_primary_attrs = [col for col in rel.columns if col not in pk]
        for lhs, rhs in fds.items():
            if set(lhs).issubset(pk) and set(lhs) != set(pk):
                if any(attr in rhs for attr in non_primary_attrs):
                    new_rel = rel[list(lhs) + rhs].drop_duplicates()
                    normalized_relations[tuple(lhs)] = new_rel

                    for attr in rhs:
                        if attr not in lhs and attr not in attributes_to_remove:
                            attributes_to_remove.append(attr)

        rel.drop(columns=attributes_to_remove, inplace=True)
        normalized_relations[pk] = rel
```

Creates a list of non-primary attributes by filtering out the primary key attributes from the columns of rel. Begins a loop over each functional dependency in the fds dictionary. Checks whether the left-hand side (determinant) of the functional dependency is a proper subset of the primary key. Checks if any attributes on the right-hand side (dependent attributes) are non-primary attributes. Identifies and removes partial dependencies by creating new relations for non-prime attributes dependent on subsets of the primary key. Outputs the resulting normalized relations.

Step 4: 3NF Check

If the table is in 2NF, we move to check for Third Normal Form (3NF). The requirements for 3NF are that the relation must be in 2NF and that there are no transitive dependencies.

```
if step >= 3:
    normalized_result = normalizer.transform_to_3NF(normalized_table_2, fds)
#after this step it calls function to check if it in 3NF or not
def validate_3NF(relations_dict, fds):
    return not has_partial_dependencies(relations_dict, fds)
```

```
def has_partial_dependencies(relations_dict, fds):
    pk = set(fds.keys())
    non_key_attributes = {attr for attrs in fds.values() for attr in attrs}

    for rel_name, rel in relations_dict.items():
        for lhs, rhs in fds.items():
            if (set(lhs).issubset(rel.columns) and
                not set(lhs).issubset(pk) and
                set(rhs).issubset(non_key_attributes)):
                return True
```

The has_partial_dependencies function identifies any partial dependencies in the relations, which would disqualify the relations from being in 3NF. The validate_3NF function provides a higher-level interface to check if all relations comply with 3NF rules.So when validate_3NF returns true it prints table is in 3NF else it decomposes into 3NF.

```
for rel_name, rel in relations_dict.items():
        for lhs, rhs in fds.items():
            if (set(lhs).issubset(rel.columns) and
                not set(rhs).issubset(lhs)):
                table1, table2 = create_new_relations(relations_dict, lhs, rhs, rel)

                modified_relations[tuple(lhs)] = table1
                modified_relations[rel_name] = table2
                break
        else:
            modified_relations[rel_name] = rel
```

The code iterates through each relation and its functional dependencies to identify transitive dependencies that prevent 3NF. If such dependencies are found, it creates two new relations: one for the dependent attributes and another for the remaining attributes. Finally, it stores and prints the modified relations, returning them along with a status indicating the transformation to 3NF.

Step 5: BCNF Check

Once in 3NF, we check for Boyce-Codd Normal Form (BCNF), which requires that for every non-trivial functional dependency, the left-hand side is a superkey.

```
if step  >= 4:
    normalized_result_bcnf = normalizer.transform_to_BCNF(normalized_table_3, pk, fds)
#after this step it calls function to check if it in BCNF or not
def validate_bcnf(relations_dict, pk, fds):
    for relation_name, relation in relations_dict.items():
        all_columns = set(relation.columns)
        for lhs, rhs in fds.items():
            for dependent in rhs:
                if dependent not in lhs:
                    if all_columns - attribute_closure(lhs, fds):
```

```
                return False
    return True
```

The **validate_bcnf** function is the main entry point for checking if relations are in BCNF. It uses the **attribute_closure** function to determine the closure of attribute sets, which is crucial for verifying whether each determinant in the functional dependencies is a superkey. If any functional dependency violates the BCNF condition (i.e., the left-hand side is not a superkey), the function returns False, indicating that the relation is not in BCNF.

If it is in BCNF then it prints table is in BCNF if not it decomposes further

```
for relation_name, relation in relations_dict.items():
        for lhs, rhs in fds.items():
            closure_result = attribute_closure(set(lhs), fds)
            if not closure_result.issuperset(relation.columns):
                combined_columns = list(lhs) + rhs
        if set(combined_columns).issubset(relation.columns) and not set(combined_columns) = = set(relation.columns):
                # Create the new relation with lhs and rhs
                new_relation = relation[combined_columns].drop_duplicates()
                updated_bcnf_relations[tuple(lhs)] = new_relation
                # Drop the dependent attributes from the original relation
                relation = relation.drop(columns=rhs)

        # Store the modified original relation
        updated_bcnf_relations[relation_name] = relation

    # Update relations_dict to be the updated relations
    relations_dict = updated_bcnf_relations.copy()
```

The function enters a decomposition process For each relation in relations_dict, it checks each functional dependency (FD) defined by lhs (left-hand side) and rhs (right-hand side). The function calculates the **attribute closure** of lhs using the attribute_closure function. The closure represents all attributes that can be functionally determined by lhs based on the FDs. If the closure of lhs does not include all attributes of the relation, it indicates that this functional dependency violates BCNF. This means that lhs is not a superkey (a key that can uniquely identify tuples in the relation), which is a requirement in BCNF. When a BCNF violation is detected:

- The function creates a new relation consisting of the columns in lhs and rhs combined (these attributes depend on lhs).
- This new relation is added to updated_bcnf_relations.
- The original relation is modified by removing the rhs attributes, which are no longer needed in it since they are now represented in the new relation.
- After decomposing based on all functional dependencies, the modified original relation (with rhs attributes removed) is also stored in updated_bcnf_relations. he process continues by updating relations_dict with updated_bcnf_relations, and the loop repeats until all relations meet BCNF.

Step 6: 4NF Check After achieving BCNF, we proceed to check for Fourth Normal Form (4NF). 4NF addresses multi-valued dependencies (MVDs) that can cause anomalies.

```
if step >= 5:
  normalized_result = normalizer.transform_to_4NF(normalized_table_bcnf, mvds)
#after this step it calls function to check if it is in 4NF or not
def validate_4NF(relations, mvds):
   for relation_name, relation in relations.items():
      for determinant, dependents in mvds.items():
         for dependent in dependents:
            if isinstance(determinant, tuple):
               determinant_cols = list(determinant)
            else:
               determinant_cols = [determinant]

            if all(col in relation.columns for col in determinant_cols + [dependent]):
               grouped = relation.groupby(determinant_cols)[
                  dependent].apply(set).reset_index()
               if len(grouped) < len(relation):
                  print(f"Multi-valued dependency violation: {determinant} ->> {dependent}")
                  return False
   return True
```

This validate_4NF function checks whether each relation in a set of relations satisfies Fourth Normal Form (4NF). In 4NF, a relation should not have multi-valued dependencies (MVDs) unless they are implied by a superkey.
If it returns true prints table is in 4NF if not it decomposes further

```
for relation_name, relation in relations.items():
         for determinant, dependents in mvds.items():
            for dependent in dependents:
               if isinstance(determinant, tuple):
                  determinant_cols = list(determinant)
               else:
                  determinant_cols = [determinant]
               if all(col in relation.columns for col in determinant_cols + [dependent]):
                  grouped = relation.groupby(determinant_cols)[
                     dependent].apply(set).reset_index()
                  if len(grouped) < len(relation):
                     table_1 = relation[determinant_cols + [dependent]].drop_duplicates()
                     four_relations[tuple(determinant_cols)] = table_1
     table_2 = relation[determinant_cols + [col for col in relation.columns if col not  in [dependent]
+ determinant_cols]].drop_duplicates()
                     four_relations[relation_name] = table_2
                     break
               else:
                  continue
               break
            else:
               four_relations[relation_name] = relation
```

```
    if len(four_relations) == len(relations):
        return four_relations, False  # Add False here to maintain the return structure
    else:
        return transform_to_4NF(four_relations, mvds)  # Recursive call
```

tansform_to_4NF function is designed to decompose a set of relations (tables) into Fourth Normal Form (4NF) by eliminating multi-valued dependencies (MVDs) that cause redundancy. If any relation violates 4NF, the function proceeds with decomposition :For each relation, it iterates through the MVDs (mvds).For each MVD, it checks if the MVD applies to the current relation by verifying that the columns in both the determinant and dependent are present in the relation.
If the MVD is relevant to the relation, the function groups the relation by the determinant columns (determinant_cols), collecting the unique values of the dependent attribute as sets.This is used to identify redundancy caused by the MVD.If grouping by the determinant columns results in fewer rows than the original relation, this indicates that the MVD causes redundancy. The function then decomposes the relation into two tables: table_1 contains the determinant columns and the dependent column (capturing the MVD). table_2 contains the determinant columns and the remaining columns (excluding the dependent), thus splitting the multi-valued dependency.
These tables are added to four_relations, effectively splitting the relation to remove the MVD.
If a relation was decomposed, the function calls itself recursively with four_relations as the new set of relations. This ensures that any further MVDs introduced by the decomposition are also handled. If all MVDs are removed after decomposition, the function returns four_relations along with a flag indicating whether the relations satisfy 4NF.

Step 7: 5NF Check

Finally, we check for Fifth Normal Form (5NF), which ensures that every join dependency in the relation is implied by the candidate keys.

```
if step >= 6:
 normalized_result = normalizer.transform_to_5NF(normalized_table_4nf, pk, fds)
```

it checks whether it is 5NF or not
is_superkey function:This helper function checks if a given set of attributes (the determinant) is a superkey for a relation.It groups the relation by the determinant attributes and counts the occurrences for each group.If each group appears only once (count is 1 for all groups), then the determinant is a superkey, and the function returns True.
validate_5NF function:This function checks if each relation in the relations dictionary satisfies 5NF.**Candidate Key Input**:For each relation, it prompts the user to input candidate keys (sets of attributes) in the format (A, B), (C, D).The user's input is parsed and stored in candidate_keys_dict with each relation name as a key.**Project and Check for 5NF**:For each subset of columns (attrs), it:**Projects** the data onto the selected subset (attrs) and its complement,**Checks if attrs is a superkey** by verifying if it contains any candidate key as a subset. **Performs a Join Test**:It constructs the joined data by combining the projections on attrs and its complement. If the join result doesn't match the original data, it means the table fails the 5NF test for that subset (attrs). If any subset fails the test, it returns False with the candidate keys. If all subsets pass the join test, the function returns True, indicating the relation satisfies 5NF, along with the candidate keys.
The main code that checks for 5NF is in the **join test section** within validate_5NF:

```
joined_data = {(row1 + row2) for row1 in projected_data for row2 in complement_data}
```

```
if set(data_tuples) != joined_data:
    print("Failed 5NF check for attributes:", attrs)
    return False, candidate_keys_dict
```

This part compares the result of joining projected_data (projection on attrs) and complement_data (projection on the complement of attrs) with the original data (data_tuples). If they don't match, it indicates a failure to satisfy 5NF for that subset.

So if a relation violates 5NF it further decomposes using above code

```
for relation_name, relation in relations:
            candidate_keys = candidate_keys_dict[relation_name]
            decomposed_relations = decompose_into_5NF(
                relation_name, relation, candidate_keys)
            five_relations.append(decomposed_relations)
def decompose_into_5NF(relation_name, dataframe, candidate_keys):
def project(df, attributes):
        return df[list(attributes)].drop_duplicates().reset_index(drop=True)

    def is_lossless(df, df1, df2):
        common_columns = set(df1.columns) & set(df2.columns)
        if not common_columns:
            return False
        joined_df = pd.merge(df1, df2, how='inner', on=list(common_columns))
        return df.equals(joined_df)
    decomposed_rel = [dataframe]
    for key in candidate_keys:
        new_tables = []
        for table in decomposed_rel:
            if set(key).issubset(set(table.columns)):
                table1 = project(table, key)
                remaining_columns = set(table.columns) - set(key)
                table2 = project(table, remaining_columns | set(key))

                if is_lossless(table, table1, table2):
                    new_tables.extend([table1, table2])
                else:
                    new_tables.append(table)
            else:
                new_tables.append(table)
        decomposed_rel = new_tables

    return decomposed_rel
```

transform_to_5NF calls decompose_into_5NF to break the relation down into smaller tables that eliminate join dependencies, ensuring that each decomposition is lossless.The function attempts to decompose the relation in a way that satisfies 5NF by removing any non-trivial join dependencies.

The result is a set of tables that, when joined, can recreate the original relation without introducing redundancy or violating 5NF.

Conclusion: Through this structured approach, we systematically check and transform the input table through normalization forms up to 5NF, ensuring a robust database schema that minimizes redundancy and enhances data integrity.

#####So here in Main.py file into sections to clarify its purpose and functionality below :

1. Imports and File Reading:

```
import pandas as pd
import csv
import normalizer
import re
# Reading the input csv file and the FDs text file
try:
   table = pd.read_csv('referenceInputTable.csv')
   print(f"Provided sample input Table:\n{table}\n")
except FileNotFoundError:
   print("Error: 'referenceInputTable.csv' not found.")
   exit(1)

# Enter FDs as input by reading file
try:
   with open('fds.txt', 'r') as file:
      lines = [line.strip() for line in file]
except FileNotFoundError:
   print("Error: 'fds.txt' not found.")
   exit(1)

# Creating a dictionary to store functional dependencies (FDs)
fds = {}
for line in lines:
   determinant, dependent = line.split(" -> ")
   determinant = determinant.split(", ")
   fds[tuple(determinant)] = dependent.split(", ")
print(f"fds=\n{fds}\n")
```

Explanation: This section imports necessary libraries and reads a CSV file for the data table and a text file for functional dependencies (FDs). It handles errors if the files are not found and structures the FDs into a dictionary for easy access later.

2. Input Parsing and Multi-Valued Dependencies (MVDs):

```
# Helper function to check if any value in the series contains a comma
def contains_comma(series):
   return any(series.str.contains(','))
# Helper function to parse the main data table for comma-separated values
```

```
def inputparser(table):
    table = table.astype(str)  # Ensure all data is treated as strings
    columns_with_commas = list(filter(lambda col: contains_comma(table[col]), table.columns))
    # Split values in columns with commas
    for col in columns_with_commas:
        table[col] = table[col].apply(lambda x: list(map(str.strip, x.split(','))) if isinstance(x, str) else x)
    return table
# Enter MVDs as input
mvds = {}
while True:
    inp = input("Please enter Multi-Valued Dependencies using format X ->> A, B or X ->> Y. After
done, type exit or finish: ")
    if inp.lower() in ["exit", "finish"]:
        break
    try:
        determinant, dependent = inp.split(" ->> ")
        determinants_list = [d.strip() for d in determinant.split(',')]
        dependents_list = [d.strip() for d in dependent.split(',')]
        mvds.setdefault(tuple(determinants_list), []).extend(dependents_list)  # Use setdefault for
cleaner logic
    except ValueError:
        print("Invalid input format, enter input format as X ->> A, B or X ->> A.")
```

Explanation: The inputparser function cleans the data table by handling any comma-separated values found in its columns. The section for MVDs allows user input to build a dictionary of multi-valued dependencies, checking the format to ensure it's entered correctly.

3. User Inputs for Normal Forms and Primary Keys:

```
# Print user-entered MVDs
print('\nmvds = {')
for determinant, dependents in mvds.items():
    dependents_str = ', '.join(f"'{dep}'" for dep in set(dependents))  # Avoid duplicates
    determinant_str = ', '.join(f"'{d}'" for d in determinant)  # Format determinants
    print(f"    ({determinant_str}): [{dependents_str}],")
print('}')
# Choose any normal form as input to normalize the input table
step = int(input("\nSelect the highest normal form that the table can achieve (1: 1NF, 2: 2NF, 3: 3NF,
4: BCNF, 5: 4NF, 6: 5NF): "))
current = int(input("\nDo you want to know the highest normal form of the input table? (1: Yes, 2:
No): "))
highest_normal_form = 0  # Track the highest normal form achieved
# Enter primary or composite keys as input
pk = input("\nEnter primary keys (for composite keys, separate them with commas): ").split(', ')
pk = tuple(pk)  # Convert list to tuple
```

Explanation: This part collects user inputs for the desired highest normal form, whether they want to know the highest achieved normal form, and the primary keys. The keys are stored as a tuple for later processing.

4. SQL Data Type Determination and Table Creation:

```
# Function to determine SQL datatype
def determine_sql_datatype(dtype):
    if pd.api.types.is_integer_dtype(dtype):
        return "INT"
    elif pd.api.types.is_float_dtype(dtype):
        return "FLOAT"
    elif pd.api.types.is_bool_dtype(dtype):
        return "BOOLEAN"
    elif pd.api.types.is_datetime64_any_dtype(dtype):
        return "DATETIME"
    elif pd.api.types.is_string_dtype(dtype):
        max_length = dtype.str.len().max()  # Get maximum length of string values
        return f"VARCHAR({max_length})" if max_length is not None else "VARCHAR(255)"
    else:
        return "VARCHAR(255)"  # Default for unknown types

# Function to generate SQL for creating the 1NF table
def generate_1nf_table_sql(pk, dataframe):
    pk = list(dataframe.keys())[0]  # Assume the first column is the primary key
    table_name = "_".join(pk)
    dataframe = dataframe[pk]
    create_query = f"CREATE TABLE {table_name} (\n"
    for column in dataframe.columns:
        dtype = determine_sql_datatype(dataframe[column])
        if column in pk:
            create_query += f"  {column} {dtype} NOT NULL PRIMARY KEY,\n"
        else:
            create_query += f"  {column} {dtype},\n"
    create_query = create_query.rstrip(',\n') + "\n);"
    print(create_query)  # Output the generated SQL query

#function to generate output queries after 2NF including foreign keys
def create_tables_for_normalized_relations(relations, fds):
    for rel_name, relation in relations.items():
        # Assume `rel_name` contains the primary key(s) and convert to tuple if it's a single string
        pks = rel_name
        pks = (pks,) if isinstance(pks, str) else pks

        # Construct table name from primary keys
```

```
    table_name = "_".join(pks)

    # Begin the CREATE TABLE statement for the relation
    create_query = f"CREATE TABLE {table_name} (\n"

    # Loop through columns in the relation and add each to the CREATE TABLE statement
    for column in relation.columns:
      # Pass the actual column to determine_sql_datatype
      create_query += f"  {column} {determine_sql_datatype(relation[column])}"
      if column in pks:  # Add NOT NULL and PRIMARY KEY constraint if column is a primary key
        create_query += " NOT NULL PRIMARY KEY"
      create_query += ",\n"

    # Track foreign keys already added to avoid duplicates
    foreign_keys_added = set()

    # Loop through functional dependencies to create foreign key constraints where applicable
    for (determinants, dependents) in fds.items():
      for dep in dependents:
        # Only add foreign key if column is part of current relation and not a primary key
        if dep in relation.columns and dep not in pks:
          # Search for the source table containing the foreign key column
          for source_table, source_columns in relations.items():
            if dep in source_columns.columns and source_table != rel_name:
              # Add the foreign key if it has not already been added
              if (dep, source_table) not in foreign_keys_added:
                # Format source table name if its a tuple
                source_table_name = "_".join(source_table) if isinstance(source_table, tuple) else
source_table
                create_query += f"  FOREIGN KEY ({dep}) REFERENCES
{source_table_name}({dep}),\n"
                foreign_keys_added.add((dep, source_table))

    # Remove trailing comma, newline, and finalize the CREATE TABLE statement
    create_query = create_query.rstrip(',\n') + "\n);"

    # Output the SQL query for the generated table (useful for debugging or review)
    print(create_query)
```

Explanation: The determine_sql_datatype function checks the data types in the DataFrame and assigns appropriate SQL types. The generate_1nf_table_sql function generates SQL statements for creating tables, focusing on primary keys and their constraints for 1NF. This function is used to generate output queries after 2NF create_tables_for_normalized_relations(relations, fds) it takes fds and generates pk and foreign keys .

5. Normalization and Table Creation for Each Normal Form:

```
table = inputparser(table)
# Normalize to 1NF
if step >= 1:
  normalized_table_1, onenfcheck = normalizer.transform_to_1NF(table, pk)
  if onenfcheck:
    print("Given input table is already in 1NF.\n")
    highest_normal_form = max(highest_normal_form, 1)
  if step == 1:
    print("Generating output queries for 1NF-->\n")
    generate_1nf_table_sql(pk, normalized_table_1)

# Normalize to 2NF
if step >= 2:
  normalized_table_2, twonfcheck = normalizer.transform_to_2NF(normalized_table_1, pk, fds)
  if twonfcheck:
    print("Given input table is already in 2NF.\n")
    highest_normal_form = max(highest_normal_form, 2)
  if step == 2:
    print("Generating output queries for 2NF-->\n")
    create_tables_for_normalized_relations(normalized_table_2, fds)

# ContinueS this process for 3NF, BCNF, 4NF, and 5NF
```

Explanation: The normalization functions (transform_to_1NF, transform_to_2NF, etc.) are called to check and process the table for each normal form.
SQL queries are generated for each step if requested by the user.

6. Final Outputs and Normalization Completion:

```
# Output the highest normal form achieved if the user requested it
if current == 1:
  if highest_normal_form == 0:
    print("The input table is still not normalized into any specific normal form.")
  else:
    normal_forms = {
      1: "1NF",
      2: "2NF",
      3: "3NF",
      4: "BCNF",
      5: "4NF",
      6: "5NF"
    }
    print(f"The highest normal form achieved is: {normal_forms[highest_normal_form]}")
print("Normalization process completed.")
```

Explanation: This final section checks if the user wants to know the highest normal form achieved and displays it. It concludes the normalization process, signaling that all operations are complete.