DBMS PROJECT: RDMS NORMALIZER

ABSTRACT

This project is designed to simplify and streamline the database normalization process. By taking a database with tables and functional dependencies as input, the program organizes and restructures the tables to meet standard normalization forms, reducing data duplication and improving database reliability. The program works step-by-step to apply normalization rules, from the most basic (First Normal Form) to more advanced forms, even up to Fifth Normal Form if required. Along the way, it generates SQL queries that can recreate these refined tables in a database, making the normalized structure easy to implement. Additionally, the program can assess and indicate the highest level of normalization achieved, making it a practical tool for anyone looking to create well-organized and efficient databases.

ANUNAY REDDY KALLEM (12619987)

UDAY DOGIPARTHI (12621336)

Input –

1. Database Table (.csv) file path.
2. Functional Dependencies
3. Multi Valued Dependencies
4. Key(s)
5. Checking if user wants to find the highest normal form of Database Table.
6. Choice of the highest normal form to reach requested by the user.

Output –

1. If the input table does not meet 1NF, then the table after converting to 1NF would be written to both **Output.txt** file and to **converted1NF.csv** file.
2. **Output.txt** will include SQL queries for the tables decomposed to highest normal form specified by the user.
3. If requested, **Output.txt** will also provide information about the highest normal form achieved by the given table.

Project Stages -

The entire project can be divided into 4 stages:

Stage 1: Taking inputs from the user.

Stage 2: Finding the Highest Normal Form of the Table. (If the user asks for it)

Stage 3: Converting the table to the Highest Normal Form as per user desire.

Stage 4: Writing the output to Output.txt file.

Stage 1:

The process starts by gathering inputs from the user.

- **Database Table File Path:**
  - The user provides the file path to a **.csv** file, which will undergo normalization. This path is stored in the variable **csv_filePath.**
- **Functional Dependencies as Input:**
  - A list variable **FD** is created to store functional dependencies input by the user. These dependencies may follow formats such as A->B or A->B, C or A, B->C.
  - The user continues entering dependencies until they type **"Done"** indicating the user is done with giving FDs as input.
- **Multi-Valued Dependencies:**
  - A list variable **MVD** is set up to capture multi-valued dependencies, where dependencies follow the format A->>B.
  - Input continues until the user enters "Done" signifying the end of multi-valued dependencies.
- **Key Input:**
  - The user specifies a key, which can be a primary or composite primary key.
  - Attributes within the key should be separated by commas (",").
- **Normal Form Check:**
  - An integer (1 or 2) is entered, where 1 means the user wants to determine the highest normal form. If 1 is chosen, the **check_normal_form()** function is called, and the output is saved in **"Output.txt"**.
- **Highest Normal Form Conversion:**
  - The user specifies a normal form number to which the table should be converted.

Stage 2:

- If user gives “1” as Input for Step 5 in Stage 1.
- When “1” is received as input then **check_normal_form()** function would be executed.
- The **check_normal_form()** has 4 arguments - csv_filePath, FD, Key, MVD.
  - **csv_filePath:** input csv file path
  - **FD:** list of functional dependencies
  - **Key:** input Key
  - **MVD:** list of multi valued dependencies.

The function definition contains a nested if structure ensuring that the function progresses to higher normal form only if the previous normal form is satisfied, and it if one of the conditions is not met.

```
  # Function to identify the highest normal form of given table
> def check_normal_form(csv_filePath, FD, Key, MVD):···

  # Function used to identify if the given table is in First Normal Form (1NF)
> def check_1NF(csv_filePath):···

  # Function used to identify if the given table is in Second Normal Form (2NF)
> def check_2NF(FD,Key):···

  # Function used to identify if the given table is in Third Normal Form (3NF)
> def check_3NF(FD, Key):···

  # Function used to identify if the given table is in Boyce-Codd Normal Form (BCNF)
> def check_BCNF(FD, Key):···

  # Function used to identify if the given table is in Fourth Normal Form (4NF)
> def check_4NF(FD, Key, MVD):···

  # Function used to identify if the given table is in Fifth Normal Form (5NF)
> def check_5NF(FD, Key, MVD):···
```

The code begins by checking if the given table is in First Normal Form. If it is, the function proceeds to check the higher normal forms. Each function is explained below –

def check_1NF (csv_filePath):

- This function checks if the table satisfies the requirements for the First Normal Form (1NF), ensuring all values in the table are atomic.
- **csv_filePath**: File path for the input .csv database table.
- The file is initially opened in read mode, and its contents are processed using a CSV reader.
- Each cell value is stored in a list variable called **res [].**
- The function checks each entry in res to see if any values contain multiple elements separated by commas (",").
- If any multi-valued entry is detected, **check_normal_form ()** returns "Not in any normal form," as 1NF is the foundational normal form.
- If all values are atomic, **check_normal_form ()** then proceeds to check if the table meets the criteria for the next level, the Second Normal Form (2NF).

def check_2NF (FD, Key):

- Used to identify if the table is in Second Normal Form, i.e., should be in 1NF and no partial dependency should exist.
- FD: list of functional dependencies.
- Key: input Key
- For functional dependency A->B, if A is only a part of Key, then the partial dependency exists, and the function returns False.
- For functional dependency A, C->B, if only A is part of Key and C is not part of the Key, then the function returns True as B is uniquely identified by both A and C.
- For functional dependency A->B, if A is not in Key, then there is no partial dependency, and the function returns True.
- If the function returns False, then **check_normal_form()** returns the table is “In 1NF”.
- Else **check_normal_form()** proceeds in checking if the table is in next normal form that is Third Normal Form.

def check_3NF (FD, Key):

- Used to identify if the table is in Second Normal Form, i.e., should be in 1NF and no partial dependency should exist.
- FD: list of functional dependencies
- Key: input Key
- It initializes an empty string **c_key** to represent the candidate key.

- For each functional dependency in the outer loop has a check for each functional dependency in inner loop to check if in the left-hand side **(FD_l)** of the main functional dependency has the attributes that are in the right-hand side **(FD_rr)** of the inner loop's functional dependency.
- If both conditions are met, then the function returns False and **check_normal_form()** returns the table is **"In 2NF".**
- Else **check_normal_form()** proceeds in checking if the table is in next normal form that is Boyce-Codd Normal Form

def check_BCNF (FD, Key):

- Used to identify if the table is in **Boyce-Codd Normal Form**, i.e., should be in 3NF and for every X->A X should be a super key or A is a prime attribute.
- FD: list of functional dependencies
- Key: input Key
- Iterates through each functional dependency. Splits the functional dependency based on left hand side and right-hand side.
- Here, it checks whether the left-hand side of the functional dependency is super key or not. If super key then the relation is in BCNF else, the relation is not in BCNF.
- Here, to check whether it is super key, we check left hand side of functional dependency is a proper subset of the key or not.

def check_4NF (FD, Key, MVD):

- Multi valued dependency should exist for it to be in 4NF.
- **FD:** functional dependencies.
- **Key:** input Key
- **MVD:** list of multi valued dependencies.
- It initializes two empty lists, FDs and MVDs, to store the processed functional dependencies and multi-valued dependencies, respectively.
- FDs contains the list of elements of each functional dependency in each index.
- If A->>B and A->>C then MVDs contains an element [A, B, C]
- If there is a relationship among them then the given multi valued dependency is invalid.
- If there is no relationship among them then the multi valued dependency is valid and the function returns False resulting in **check_normal_form()** to return the table is **"In BCNF"**.

- Else **check_normal_form()** proceeds in checking if the table is in next normal form that is Fifth Normal Form.

def check_5NF (FD, Key, MVD):

- Used to identify if the table is in **Fifth Normal Form**, i.e., should be in **4NF** and no join dependency should exist.
- **FD:** list of functional dependencies.
- **Key:** input Key
- **MVD:** list of multi valued dependencies.
- **check_5NF (candidate_keys, fds, mvds):** This is the main function to check if a given set of candidate keys, FDs, and MVDs satisfy the conditions of 5NF. It works as follows:
    - **5NF Condition 1:** All tables should be in BCNF. This part checks whether each candidate key satisfies the criteria for being a super key. If any candidate key is not a super key according to the FDs, the function returns False, indicating that the relation is not in 5NF.
    - **5NF Condition 2:** Check for join dependencies using natural join. This part of the code focuses on MVDs. It ensures that the MVDs can be represented using natural join operations and FDs. It follows these steps:
    - It splits the MVD into two parts, the left-hand side (left) and the right-hand side (right) using the **" ->> "** delimiter.
    - It checks if both the left and right sides of the MVD are either super keys or candidate keys. This ensures that the MVD is based on attributes that can uniquely determine other attributes.
    - It checks if the natural join of the left and right sides of the MVD can be expressed using FDs. It does this by finding subsets of the common attributes and checking if they are super keys or candidate keys.
    - If any of these conditions fail, the function returns False. If all conditions are met, the function returns True, indicating that the relation is in 5NF.
    - If the function finds that the table satisfies a specific normal form, it returns a string indicating which normal form it's in.
    - If none of the normal forms are satisfied, the function returns **"Not in any Normal Form."**

Stage 3:

- Converting the given table to the highest normal form specified by the user.
- Based on the user input from Stage-1: Step-6 the program identifies into which normal form the given database table is to be decomposed.
- Based on the following table each input is assigned a numeric number –

Input	user_choice
'1'	1
'2'	2
'3'	3
'b' or 'B'	3.5
'4'	4
'5'	5

- For a given k the functions from **convert_to_1NF to convert_to_(k)NF** will be executed.
- Upon executing a function and decomposing the tables, the resultant table – **res_tables {}** will be passed as the new table to the next normal form function.

```
> if(user_choice >= 1):···
  res_tables = {}
  # based on k the functions from convert_to_1NF to convert_to_(k)NF will be executed
  if(user_choice >= 2):
  |    res_tables = convert_to_2NF(FD, Key)
  if(user_choice >= 3):
  |    res_tables = convert_to_3NF(FD, Key, res_tables)
  if(user_choice >= 3.5):
  |    res_tables = convert_to_BCNF(FD, Key, res_tables)
  if(user_choice >= 4):
  |    res_tables = convert_to_4NF(FD, Key, MVD, res_tables)
  if(user_choice == 5):
  |    res_tables = convert_to_5NF(FD, Key, MVD, res_tables)
```

- **def convert_to_1NF (csv_filePath):**

This function helps in converting the given table to 1NF, i.e., all the cells of the tuples would contain only atomic values.

- We pass the csv file which contains the relational data to the function.
- **'res'** is used to store the 1NF normalized data.
- For each row, we examine whether each cell contains multiple values or not. This is done by splitting each cell using a comma and the results are stored in **'each_index'** list.
- Code calculates the number of elements or the values in each cell and records the information in **'num_elements'** list.
- Next step involves in computing the total number of combinations possible based on the number of elements in each cell. This is achieved by multiplying the number of elements for each cell together.
- For each combination, a new row is generated by selecting one value from each cell. This new row is added to the **'res'** list.

- **def convert_to_2NF (FD, Key):**

This function facilitates the conversion of a table to Second Normal Form (2NF) by eliminating partial dependencies.

a. The **tables** variable stores the resulting tables in 2NF.

b. The function iterates through each functional dependency in FD, using **fd.split('->')** to separate the left and right sides of the dependency by the arrow symbol ->.

c. Various conditions are checked to decompose the tables according to the functional dependencies.

d. Once all dependencies are processed, the function verifies if the original key is present in any of the new tables. If the key is absent, its attributes are added to a "Candidate table."

e. Duplicate attributes in the resulting tables are removed with list(set(attributes)).

f. Finally, the function returns a dictionary where the keys represent table names, and the values are lists of attributes for each table.

- **FD:** List of functional dependencies.
- **Key:** The input key.

- **def convert_to_3NF (FD, Key, tables):**

  This function converts tables from Second Normal Form (2NF) to Third Normal Form (3NF) by removing transitive dependencies.

  a. It takes a list of functional dependencies, an input key, and a dictionary of tables in 2NF as inputs.

  b. The new_tables dictionary is initialized to store the resulting tables in 3NF.

  c. Each functional dependency is processed by splitting it into its left-hand side (lhs) and right-hand side (rhs) using the -> symbol. All lhs attributes are added to **lhs_fd** to keep track of attributes that appear on the left side of the functional dependencies.

  d. The function iterates through each table in the 2NF dictionary. If a table name is not in **lhs_fd**, the function examines each functional dependency to determine if it suggests a new decomposition.

  e. If a table name is present in **lhs_fd**, it indicates the table is referenced on the left side of a functional dependency. These tables are added to **new_tables** without modifications.

  f. Duplicate attributes in the tables are removed with **list(set(attributes))**.

  g. Finally, the function returns the **new_tables** dictionary, where keys represent table names and values are lists of attributes for each table.

    - **FD:** List of functional dependencies.
    - **Key:** The input key.

- **def convert_to_BCNF (FD, Key, tables):**

  This function is used to convert the decomposed 3NF table to BCNF.

  a. This function takes functional dependencies, candidate keys, and a dictionary of tables as inputs.

  b. The lists **lhs_fd** and **rhs_fd** are created to store the left and right-side attributes of each functional dependency.

c. The function populates **lhs_fd** and **rhs_fd** with the individual attributes from the left and right sides of each dependency.

d. If the left-hand side attributes contain multiple elements and match the candidate keys in Key, the right-hand side attributes are added to a "Candidate" table in the **new_tables** dictionary.

e. If the left-hand side attributes contain multiple elements and do not match the candidate keys, the left-hand attributes are added to the candidate table, and a new table is created that includes both the left and right-hand side attributes.

f. Duplicate attributes are removed from **new_table** to ensure each table contains only unique attributes.

g. Finally, the function returns the **new_tables** dictionary, which contains the decomposed tables in Boyce-Codd Normal Form (BCNF).

- **def convert_to_4NF (FD, Key, MVD, tables):**

  This function is used to convert BCNF table to 4NF.

  a. This function accepts functional dependencies (FD), candidate keys (Key), multi-valued dependencies (MVD), and a table dictionary as input parameters.

  b. Initially, it checks if each multi-valuued dependency can be derived from the functional dependencies. For example, if there is a dependency **StudentID->>Course** and StudentID is not a superkey, the relation should be decomposed into two tables: one containing StudentID and Course, and another with StudentID and all attributes except Course.

  c. These new relations are then stored in a new table dictionary. The function ultimately returns this dictionary containing the tables normalized to Fourth Normal Form (4NF).

- **def convert_to_5NF (FD, Key, MVD, tables):**

  This function is used to convert 4NF to 5NF.

  a. This function takes functional dependencies (FD), candidate keys (Key), multi-valued dependencies (MVD), and a tables dictionary as input parameters.

  b. For each table in the dictionary, it examines each attribute to check if it exists in the original tables and creates a new table with the identified attribute values.

c. The code then performs a natural join on these newly created tables, combining attribute pairs by matching values to form a new table.

d. Finally, the function returns a new tables dictionary that represents the database schema in Fifth Normal Form (5NF).

Stage 4:

1. Once the table is converted to the Highest Normal Form two functions are called:

   a. check_datatypes()

   - This function is used to identify the datatype of each attribute in the csv file.
   - This function takes **csv_filePath** as an argument.
   - The three datatypes this function would help in identifying is if an attribute is in INT, VARCHAR, DATE.
   - To identify the datatype of an attribute this function internally calls 4 other functions. Each function takes the cell value of the attribute as a parameter.

```
  # Function to check if the column is of INT Datatype
> def is_integer(attr): ···

  # Function to check if the column is of VARCHAR Datatype
> def is_alphanumeric(attr): ···

  # Function to check if the column is of Date Datatype
> def is_date(attr): ···

  # Function to check if the column is of Email - VARCHAR Datatype
> def is_email(attr): ···
```

   - Based on the Boolean results from each function the datatype of each attribute.
   - The attribute and datatype of that attribute is stored in the **data_types** dictionary as a Key and Value pair.

   b. generate_sql_queries()

   - This function is used to create an SQL query for every table contained within the **res_tables** dictionary, which is the output from Stage 3.
   - The function takes 4 arguments def generate_sql_queries(FD, Key, res_tables, data_types):
     - **FD:** list of functional dependencies

- **Key:** Input Key
- **res_tables:** The resultant dictionary of tables after converting to the highest normal form as per user request.
- **data_types:** The resultant dictionary from above step identifying the datatype of each attribute.

- The function returns a list of queries for each table in the **res_table** dictionary which would be stored in a **SQL_queries** list.

2. A new **"Output.txt"** File would be created and opened in write mode using csv.
3. If the table is initially not in 1NF and user wants to convert to a normal form greater than or equal to 1 then the table converted to a first normal form would be written to the Output.txt file.
4. The generated SQL queries from Step – 1(b) is written to the Output.txt file.
5. If the user wants to find the highest NF of given table, then the result from Stage - 2 would be written to **Output.txt**.