Abstract

The project assists in determining the highest level of normalization that the provided table adheres to and facilitates the process of normalizing the table to the specific level requested by the user.

Database normalization

Programming Project 01

**Input –**

Venkata Sai Sowmya Velamuri - 12607142

Navya Valivati - 12608155

Chandana Dontraboina - 12606139

1. Database Table (.csv) file path.
2. Functional Dependencies
3. Multi Valued Dependencies
4. Key
5. Checking if user wants to find the highest normal form of Input Database Table.
6. Choice of the highest normal form to reach (1: 1NF, 2: 2NF, 3: 3NF, B: BCNF, 4: 4NF, 5: 5NF).

**Output –**

1. If the input table is not in 1NF then the table after converting to 1NF would be written to **Output.txt** file and to **converted1NF.csv** file
2. Output.txt file containing the SQL queries for the tables decomposed to highest normal form taken as input.
3. Output.txt file containing the details of Highest normal form of the given table if the user asks for it.

**Pre-requisites –**

1. Database Table should be stored in a **.csv** File.

The entire project can be divided into 4 stages:

Stage 1: Taking inputs from the user.

Stage 2: Identifying the Highest Normal Form of the Table. (If user requests)

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

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

**Stage 1:**

Initially inputs are taken from the user.

1. Taking the Database Table file path.

* The .csv file path is taken as input from the user on which the normalization operations would be performed and stored in **csv\_filePath** variable.

1. The Functional Dependencies are taken as Input.

* A variable **FD** of type list is created, and all the input functional dependencies are appended to it.
* A functional dependency can be of type A->B or A->B,C or A,B->C.
* The variable takes input till user enters “Done” denoting that the list of functional dependencies is now completed.

1. The Multi Valued Dependencies are taken as Input.

* A variable **MVD** of type list is created, and all the input multi valued dependencies are appended to it.
* A functional dependency is of type A->>B.
* The variable takes input till user enters “Done” denoting that the list of multi valued dependencies is now completed.

1. The Key is taken as Input.

* Key can be primary key or composite primary key.
* Each attribute in the key is to be separated by a comma - “,”.

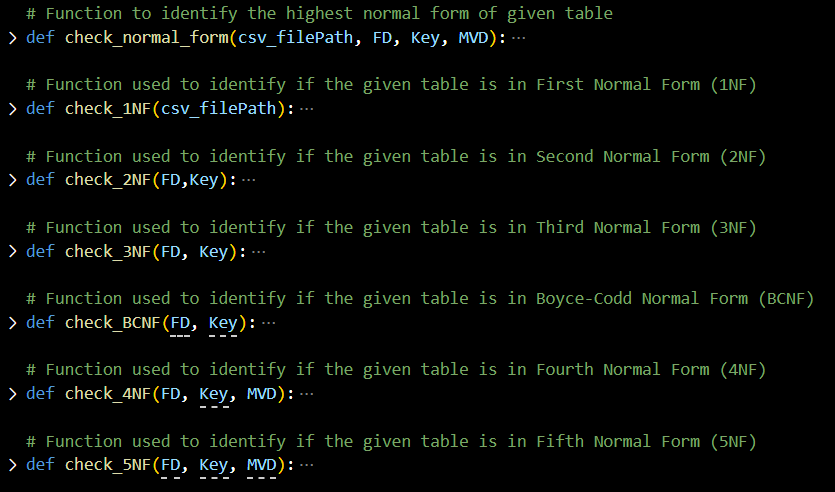
1. An integer value 1 or 2 is taken as input for yes or no respectively based on user’s choice to find the highest normal form of the input table.

If 1, then a function **check\_normal\_form()** is called as input and it’s output is written to “Output.txt” file.

1. The Highest Normal Form number is taken as input and the input table would be converted to the specified normal form.

**Stage 2:**

* If user gives “1” as Input for Step 5 in Stage 1, then this Stage would be executed to identify the Highest Normal Form of the given table.
* Upon receiving “1” as input the check\_normal\_form() function would be executed.
* The check\_normal\_form() has 4 arguments - csv\_filePath, FD, Key, MVD.
* def check\_normal\_form(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 stops as soon as one of the conditions is not met.



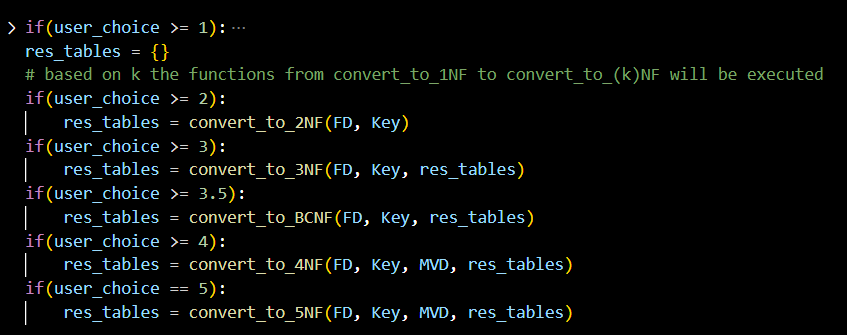
* The code begins by checking if the given table is in First Normal Form. If it is, the function proceeds to check the higher normal forms in a cascading manner. Each function is explained below -
* **def check\_1NF(csv\_filePath):**
* Used to identify if the table is in First Normal Form, i.e., all the values must be atomic in the table.
* csv\_filePath: input database table .csv filepath
* Initially opens the file in read mode and reads the file using a csv reader object.
* Each cell value is then stored into a **res[]** list variable.
* For each index in res it is checked if there are more than one elements in it separated by a comma “,”.
* If it is identified that there exists more than 1 element, then the check\_normal\_form() returns “Not in any normal form” as 1NF is the basic normal form.
* If NO, then the check\_normal\_form() proceeds on checking if the table is in next normal form that is Second Normal Form.
* **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 2NF and no transitive dependency should exist.
  + FD: list of functional dependencies.
  + Key: input Key
  + It initializes an empty string c\_key to represent the candidate key in a concatenated form.
  + 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.
  + Additionally, it checks if the left-hand side of the inner functional dependency (FD\_ll) is part of the concatenated candidate key (c\_key).
  + 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):**
  + Used to identify if the table is in Fourth Normal Form, i.e., should be in BCNF and no multi valued dependency should exist.
  + FD: list of 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]
  + Later for each MVD in MVDs the function checks if there is a relation between them by traversing through each functional dependency.
  + 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 superkey. If any candidate key is not a superkey 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 superkeys 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 superkeys or candidate keys.
    - If any of these conditions fail, the function returns False, indicating that the given set of FDs, candidate keys, and MVDs do not satisfy 5NF. 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:**

* The heart of this project – 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 k | * 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.



* Below is the function declaration of each function.
* In these functions we define the conversion of the table to the highest normal form and decompose the tables.
* The decomposed table of each function is passed as an input to next function to convert them to higher normal form.

A computer screen with text on it

Description automatically generated

* **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.

* 1. We pass the csv file which contains the relational data to the function.
  2. ‘res’ is used to store the 1NF normalized data.
  3. For each row, we examine whether the 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.
  4. Code calculates the number of elements or the values in each cell and records the information in ‘num\_elements’ list.
  5. 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.
  6. For each combination, a new row is generated by selecting one value from each cell. This new row is added to the ‘res’ list.
  7. In the end, the function returns the ‘res’ list, which holds the data in 1NF, ensuring that each row contains non-complex and individual values.
     1. csv\_filepath : the path where the data is stored in csv format.
* **def convert\_to\_2NF(FD, Key):**

This function helps in converting the given table to 2NF, i.e., removing the partial dependencies.

1. ‘tables’ stores the resulting tables in 2NF
2. Code iterates through each functional dependency in ‘FD’. ‘lhs, rhs = fd.split(‘->’)’ separates the left hand side and right hand side of the functional dependency using the arrow ‘->’.
3. We check different set of conditions to decompose the tables based on functional dependencies.
4. After processing all functional dependencies, the code checks whether the original key is present in any of the tables. If it is not found in any of the resulting tables, it adds the key attributes to ‘Candidate table’.
5. We remove duplicate attributes in the resulting table using ‘list(set(attributes))’.
6. In the end, the function returns the tables where keys are the table names and values are list of attributes for each table.
   * 1. FD: list of functional dependencies.
     2. Key: input Key

* **def convert\_to\_3NF(FD, Key, tables):**

This function is used to convert the decomposed 2NF table to 3NF. i.e., removing transitive dependencies.

1. This function accepts a list of functional dependencies, an input key and a dictionary of tables that are in 2NF.
2. ‘new\_tables’ is a dictionary that stores resulting tables in 3NF.
3. Code processes the functional dependencies by splitting each functional dependency to its left hand side(lhs) and right hand side (rhs) using ‘->’. We add all LHS attributes to ‘lhs\_fd’ list to keep track of the attributes that appear on the left hand side of the functional dependencies.
4. Code iterates through each existing table. If the table name is not in ‘lhs\_fd’, the code checks each FD to see if it implies a new decomposition.
5. If the table name is present in ‘lhs\_fd’, it means that it is referenced in the left hand side of some functional dependency. For such tables, the code adds the table to the ‘new\_tables’ dictionary without any changes.
6. Duplicate attributes are removed using ‘list(set(attributes)).
7. In the end, the code returns the new\_tables dictionary, where the keys are the table names and values are the list of attributes for each table.
   * 1. FD: list of functional dependencies.
     2. Key: input Key

* **def convert\_to\_BCNF(FD, Key, tables):**

This function is used to convert the decomposed 3NF table to BCNF. i.e., eliminating other anomalies.

1. This function takes functional dependencies, candidate keys and dictionary of tables as input.
2. ‘lhs\_fd’ and ‘rhs\_fd’ are the lists used to store left hand side and right hand side attributes of each functional dependency.
3. This code then populates the ‘lhs\_fd’ and ‘rhd\_fd’ lists with the individual attributes from the left hand side and right hand side of functional dependency.
4. If left hand side attributes are multiple and they match the candidate keys in ‘Key’ then the attributes on right hand side are added to “Candidate” table in the new\_tables dictionary.
5. If the left hand side attributes are multiple and doesn’t match if candidate keys then left hand side elements are added to candidate table and new table is created which contains both the left and right hand side attributes.
6. Duplicates are removed from new\_table to ensure that attributes areunique in each table.
7. In the end, this function returns the new\_tables dictionary which contains the decomposed tables in BCNF form.

* **def convert\_to\_4NF(FD, Key, MVD, tables):**

This function is used to convert the decomposed BCNF table to 4NF. i.e., eliminates multi valued dependencies.

1. This function takes functional dependencies FD, candidate keys Key, multi valued dependencies MVD , a table dictionary as input parameters.
2. At First, we check whether the each multivalued dependency can be derived from the functional dependency or not.   
   For example,  
   StudentID->>Course, and assume that StudentId is not a superkey. Then we need to decompose this relation into two relations. One with StudentID and Course and Second one with StudentID and attributes other than Course.
3. Store these new relations into a new table dictionary. In the end, this function returns the new tables dictionary which contains normalized 4NF tables.

* **def convert\_to\_5NF(FD, Key, MVD, tables):**

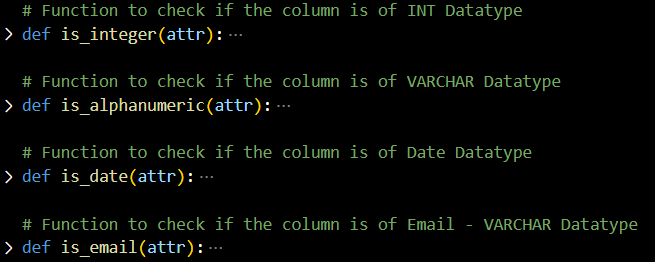
This function is used to convert decomposed 4NF to 5NF. i.e., removing join dependency and making sure that joining is lossless.

* 1. This function takes functional dependencies FD, Key, Multi valued functional dependencies MVD, tables dictionary as input parameters.
  2. For each table in the table, we consider each attribute and check if it exists in the original tables and create a table with the attributes value.
  3. Code performs a natural join on the above newly created tables. It combines pair of attributes by checking whether there are any matching values and new table is created.
  4. This function finally returns a new tables dictionary which contains database schema which is in 5NF.

**Stage 4:**

1. Once the table is converted to the Highest Normal Form two functions are called:
2. 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.
* def check\_datatypes(csv\_filePath):
* 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.



* 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.

1. 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 as shown below -
* 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.

1. A new “Output.txt” File would be created and opened in write mode using csv.
2. If the table is initially not in 1NF and user inputs 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.
3. The generated SQL queries from Step – 1(b) is written to the Output.txt file.
4. If user inputs to find the highest normal form of given table, then the result from Stage - 2 would be written to Output.txt file.