Project Documentation: Relational Database Normalizer

Sai Yashwanth Sathinapalli (12622924)

Katari Bhuvanashri (12617701)

Introduction

The **Relational Database Normalizer** is a tool developed to transform a relational database schema to comply with normal forms from **First Normal Form (1NF)** through **Domain-Key Normal Form (DKNF)**. This tool is designed to reduce redundancy, minimize anomalies, and enforce dependency constraints by normalizing the schema up to a specified normal form. The process ends by generating SQL `CREATE TABLE` statements for each of the resulting normalized tables.

Objectives

The tool provides:

1. **Interactive Input Parsing**: A guided interface for users to enter schema attributes, functional dependencies (FDs), multi-valued dependencies (MVDs), and optional data instances.
2. **Step-by-Step Normalization**: Each normal form is sequentially applied to transform the schema up to the specified level of normalization.
3. **SQL Output Generation**: Final SQL statements are generated for each table in the normalized schema, enabling easy implementation in a database.

Code Structure and Methodology

The code consists of three main classes:

1. **InputParser**: Collects and validates user inputs, preparing the data for normalization.
2. **Normalizer**: Applies normalization rules to the schema based on the specified normal form.
3. **FinalRelationGenerator**: Outputs the normalized schema in SQL format, including primary key constraints.

Each class is broken down into specific methods that serve unique roles in the normalization process. Below is a detailed breakdown of each part.

---

1. Input Parsing: `InputParser` Class

The `InputParser` class is designed to manage user inputs. It collects:

- **Schema Attributes**: Defines the attributes of the table.
- **Functional Dependencies (FDs)**: These determine which attributes are functionally dependent on others.
- **Multi-Valued Dependencies (MVDs)**: Optional, used for attributes that independently determine other attributes.
- **Data Instances**: Optional JSON data for validating dependencies and join conditions.

Methods in `InputParser`

- **`__init__`**: Initializes lists and variables for storing schema attributes, dependencies, MVDs, and data instances.
- **`get_user_input()`**:
  - Prompts the user to enter schema attributes, FDs, MVDs, and data instances.
  - Processes the input to remove extra whitespace and formats it for the normalization phase.
  - Prints feedback to guide the user on input format.

Sample Input

Given a schema with attributes, dependencies, and MVDs:

1. **Schema Attributes**:

```
Enter schema attributes (comma-separated): A, B, C, D, E
```

2. **Functional Dependencies**:

```
Enter functional dependencies (FDs) in the format 'A,B -> C' (one per
line). Type 'done' to finish:
A -> B
B -> C
C -> D
done
```

3. **Multi-Valued Dependencies**:

```
Enter multi-valued dependencies (MVDs) in the format 'A ->-> B'
(optional, type 'done' to finish):
A ->-> E
done
```

4. **Data Instances (optional)**:

```
[{"A": "1", "B": "2", "C": "3", "D": "4", "E": "5"}, {"A": "1", "B":
"2", "C": "3", "D": "4", "E": "6"}]
```

The formatted inputs are then passed to the `Normalizer` class.

---

2. Normalization: `Normalizer` Class

The `Normalizer` class is the core of the tool, handling the normalization process up to the user-defined target normal form. This class contains methods for each normal form (1NF to DKNF), with each method progressively decomposing the schema to ensure compliance with database design principles.

Methods in `Normalizer`

1. **`to_1NF()`**:
   - **Objective**: Ensure each attribute in the schema is atomic (no multi-valued attributes).
   - **Method**: Identifies and removes multi-valued attributes, creating separate tables where needed.
   - **Example Output**:

     ```
     Converted to 1NF: Removed multi-valued attributes.
     ```

2. **`to_2NF()`**:
   - **Objective**: Remove partial dependencies where non-key attributes depend only on part of a composite key.
   - **Method**: Identifies partial dependencies and decomposes tables to store these attributes in a new table.
   - **Example Output**:

     ```
     Decomposed partial dependencies to achieve 2NF.
     ```

3. **`to_3NF()`**:
   - **Objective**: Eliminate transitive dependencies, ensuring non-key attributes depend directly on the primary key.
   - **Method**: Detects transitive dependencies and creates new tables to store them independently.
   - **Example Output**:

     ```
     Decomposed transitive dependencies to achieve 3NF.
     ```

4. **`to_BCNF()`**:
   - **Objective**: Ensure that all functional dependencies have a superkey as the determinant.
   - **Method**: Identifies dependencies where the left side is not a superkey and decomposes tables to enforce BCNF.
   - **Example Output**:

```
Ensured BCNF by decomposing to make all determinants superkeys.
```

5. **`validate_mvd(left, right)`**:
   - **Objective**: Validate multi-valued dependencies using data instances.
   - **Method**: Groups data instances by `left` attributes and checks if `right` attributes are independent, creating tables if the MVD is valid.
   - **Example Output**:

     ```
     MVD (A) ->-> (E) validated successfully.
     ```

6. **`to_4NF()`**:
   - **Objective**: Eliminate non-trivial MVDs to achieve 4NF.
   - **Method**: Checks MVDs and decomposes tables as necessary to prevent redundant storage of independent attributes.
   - **Example Output**:

     ```
     Ensured 4NF by decomposing validated MVDs.
     ```

7. **`to_5NF()`**:
   - **Objective**: Decompose tables to eliminate redundancy caused by join dependencies.
   - **Method**: Validates join dependencies with data instances, creating new tables where attributes have unique value groupings.
   - **Example Output**:

     ```
     Ensured 5NF by decomposing join dependencies.
     ```

8. **`to_DKNF()`**:
   - **Objective**: Ensure every constraint is domain- or key-based, the highest level of normalization.
   - **Method**: Checks domain constraints and removes attributes that violate data type integrity.
   - **Example Output**:

     ```
     Ensured DKNF by applying domain and key constraints.
     ```

9. **`normalize(target_nf)`**:
   - **Objective**: Sequentially applies each normalization step up to the user-specified normal form.
   - **Method**: Calls each normalization method from 1NF to the target normal form and saves the results.
   - **Example Execution**:

     ```
     Enter target normal form (1 to 6): 5
     ```

---

3. SQL Generation: `FinalRelationGenerator` Class

The `FinalRelationGenerator` class creates SQL `CREATE TABLE` statements for each table in the normalized schema. These statements reflect the decompositions applied during normalization, making it easy to implement the resulting schema in a relational database.

Methods in `FinalRelationGenerator`

- **`generate_sql(output_to_file=False, filename="normalized_schema.sql")`**:
  - **Purpose**: For each table in the normalized schema, generates a SQL statement, specifying attributes and primary keys.
  - **Method**: Iterates over each table, formats `CREATE TABLE` statements, and optionally writes them to a file.
  - **Example SQL Output**:

```
CREATE TABLE Table_1 (
    A VARCHAR(255),
    B VARCHAR(255),
    PRIMARY KEY (A)
);
CREATE TABLE Table_2 (
    B VARCHAR(255),
    C VARCHAR(255),
    PRIMARY KEY (B)
);
CREATE TABLE Table_3 (
    C VARCHAR(255),
    D VARCHAR(255),
    PRIMARY KEY (C)
);
CREATE TABLE Table_4 (
    A VARCHAR(255),
    E VARCHAR(255)
);
```

The SQL output provides a normalized database schema ready for implementation, based on the user's specified normalization level.

Sample Execution and Results

Given the sample inputs:

```
plaintext
Copy code
Schema: A, B, C, D, E
Functional Dependencies: A -> B, B -> C, C -> D
Multi-Valued Dependency: A ->-> E
Data Instances:
[{"A": "1", "B": "2", "C": "3", "D": "4", "E": "5"}, {"A": "1", "B": "2",
"C": "3", "D": "4", "E": "6"}]
Target Normal Form: 5
```

Expected Output:

```
Converted to 1NF: Removed multi-valued attributes.
MVD ('A',) ->-> ('E') validated successfully.
Ensured 5NF by decomposing join dependencies.
```

SQL Output:

```
CREATE TABLE Table_1 (
    A VARCHAR(255),
    B VARCHAR(255),
    PRIMARY KEY (A)
);

CREATE TABLE Table_2 (
    B VARCHAR(255),
    C VARCHAR(255),
    PRIMARY KEY (B)
);

CREATE TABLE Table_3 (
    C VARCHAR(255),
    D VARCHAR(255),
    PRIMARY KEY (C)
);

CREATE TABLE Table_4 (
    A VARCHAR(255),
    E VARCHAR(255)
);
```

In this output:

- **Table_1** stores the dependency `A -> B` with `A` as the primary key.
- **Table_2** represents the dependency `B -> C` with `B` as the primary key.
- **Table_3** holds the dependency `C -> D` with `C` as the primary key.
- **Table_4** captures the multi-valued dependency `A ->-> E`, ensuring no redundancy in storage.

Summary of Key Design Decisions

1. **Sequential Normalization**:
    - The `Normalizer` class applies each normal form sequentially from 1NF to the user-specified target (up to DKNF).

    - This approach allows for progressive refinement of the schema, reducing redundancy and dependency anomalies at each stage.
2. **MVD Validation**:
    - For 4NF, multi-valued dependencies are validated against data instances, ensuring that tables are only decomposed for validated MVDs. This prevents unnecessary decompositions and focuses only on true independent relationships.
3. **Join Dependency Detection**:
    - For 5NF, join dependencies are analyzed using unique values within data instances, allowing for efficient decomposition of tables to eliminate redundancy due to join dependencies.
4. **Final SQL Generation**:
    - The SQL output is structured to mirror the normalized schema directly. Each dependency and decomposed table is represented with primary key constraints, enabling seamless transition from the design phase to implementation.

Conclusion

The Relational Database Normalizer provides a complete solution for database schema normalization, up to DKNF. This tool simplifies the normalization process, eliminating redundancy, minimizing anomalies, and producing SQL statements for quick implementation.

Through interactive input, structured normalization, and automated SQL generation, this tool supports best practices in relational database design and ensures that users achieve well-structured, efficient, and maintainable databases.