Normalization

1. First Normal Form (1NF):

Rule for 1NF:
- Eliminate duplicate columns from the same table.
- Create a separate table for each set of related data.

Example 1: Non-1NF Table:
```sql
-- Non-1NF Table
CREATE TABLE Non1NFTable (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50),
    subjects VARCHAR(100)  -- This column may contain multiple subjects in a single cell
);

INSERT INTO Non1NFTable (student_id, student_name, subjects)
VALUES (1, 'John Doe', 'Math,Physics,Chemistry');
```

Example 2: 1NF Tables:
```sql
-- 1NF Tables
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50)
);

CREATE TABLE Subjects (
    subject_id INT PRIMARY KEY,
    subject_name VARCHAR(50)
);

CREATE TABLE StudentSubjects (
    student_id INT,
    subject_id INT,
    PRIMARY KEY (student_id, subject_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (subject_id) REFERENCES Subjects(subject_id)
);
```

```
INSERT INTO Students (student_id, student_name) VALUES (1, 'John Doe');
INSERT INTO Subjects (subject_id, subject_name) VALUES (1, 'Math'), (2, 'Physics'), (3, 'Chemistry');
INSERT INTO StudentSubjects (student_id, subject_id) VALUES (1, 1), (1, 2), (1, 3);
```

2. Second Normal Form (2NF):

Rule for 2NF:
- Be in 1NF.
- Have a primary key.
- No partial dependencies.

Example 1: Non-2NF Table:
```sql
-- Non-2NF Table
CREATE TABLE Non2NFTable (
    order_id INT PRIMARY KEY,
    product_id INT,
    product_name VARCHAR(50),
    quantity INT,
    PRIMARY KEY (order_id, product_id)
);

INSERT INTO Non2NFTable (order_id, product_id, product_name, quantity)
VALUES (1, 101, 'Laptop', 2);
```

Example 2: 2NF Tables:
```sql
-- 2NF Tables
CREATE TABLE Orders (
    order_id INT PRIMARY KEY
);

CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50)
);

CREATE TABLE OrderDetails (
    order_id INT,
    product_id INT,
    quantity INT,
```

```
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO Orders (order_id) VALUES (1);
INSERT INTO Products (product_id, product_name) VALUES (101, 'Laptop');
INSERT INTO OrderDetails (order_id, product_id, quantity) VALUES (1, 101, 2);
```

3. Third Normal Form (3NF):

Rule for 3NF:
- Be in 2NF.
- No transitive dependencies.

Example 1: Non-3NF Table:
```sql
-- Non-3NF Table
CREATE TABLE Non3NFTable (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    department_name VARCHAR(50),
    manager_name VARCHAR(50),
    PRIMARY KEY (employee_id)
);

INSERT INTO Non3NFTable (employee_id, employee_name, department_name,
manager_name)
VALUES (1, 'Alice', 'HR', 'Bob');
```

Example 2: 3NF Tables:
```sql
-- 3NF Tables
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES Employees(employee_id)
);

CREATE TABLE Departments (
    department_id INT PRIMARY KEY,
```

```
    department_name VARCHAR(50)
);

INSERT INTO Employees (employee_id, employee_name, manager_id) VALUES (1, 'Alice',
NULL);
INSERT INTO Employees (employee_id, employee_name, manager_id) VALUES (2, 'Bob', 1);
INSERT INTO Departments (department_id, department_name) VALUES (1, 'HR');
```

Certainly! Let's continue with examples for higher normal forms.

4. Boyce-Codd Normal Form (BCNF):

Rule for BCNF:
- Be in 3NF.
- Every determinant must be a candidate key.

Example 1: Non-BCNF Table:
```sql
-- Non-BCNF Table
CREATE TABLE NonBCNFTable (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    project_id INT,
    project_name VARCHAR(50),
    PRIMARY KEY (employee_id, project_id)
);

INSERT INTO NonBCNFTable (employee_id, employee_name, project_id, project_name)
VALUES (1, 'Alice', 101, 'ProjectA');
```

Example 2: BCNF Tables:
```sql
-- BCNF Tables
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50)
);

CREATE TABLE Projects (
    project_id INT PRIMARY KEY,
    project_name VARCHAR(50)
);
```

```sql
CREATE TABLE EmployeeProjects (
    employee_id INT,
    project_id INT,
    PRIMARY KEY (employee_id, project_id),
    FOREIGN KEY (employee_id) REFERENCES Employees(employee_id),
    FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);

INSERT INTO Employees (employee_id, employee_name) VALUES (1, 'Alice');
INSERT INTO Projects (project_id, project_name) VALUES (101, 'ProjectA');
INSERT INTO EmployeeProjects (employee_id, project_id) VALUES (1, 101);
```

5. Fourth Normal Form (4NF):

Rule for 4NF:
- Be in BCNF.
- No multi-valued dependencies.

Example 1: Non-4NF Table:
```sql
-- Non-4NF Table
CREATE TABLE Non4NFTable (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50),
    course_id INT,
    course_name VARCHAR(50),
    instructor_name VARCHAR(50),
    PRIMARY KEY (student_id, course_id)
);

INSERT INTO Non4NFTable (student_id, student_name, course_id, course_name,
instructor_name)
VALUES (1, 'John Doe', 101, 'Math', 'Professor Smith');
```

Example 2: 4NF Tables:
```sql
-- 4NF Tables
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50)
);
```

```
CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50),
    instructor_name VARCHAR(50)
);

CREATE TABLE StudentCourses (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);

INSERT INTO Students (student_id, student_name) VALUES (1, 'John Doe');
INSERT INTO Courses (course_id, course_name, instructor_name) VALUES (101, 'Math',
'Professor Smith');
INSERT INTO StudentCourses (student_id, course_id) VALUES (1, 101);
```

These examples demonstrate the transformation of tables from non-BCNF and non-4NF forms to BCNF and 4NF, respectively. The goal is to eliminate anomalies and ensure the database structure is robust and well-organized.

Certainly! Let's dive deeper into the examples, this time focusing on the `COMMIT` and `ROLLBACK` operations.

5. COMMIT Operation:

The `COMMIT` operation finalizes a transaction, making all the changes made during the transaction permanent.

Example 1: Using SQL

```sql
-- Example of COMMIT operation in SQL
BEGIN TRANSACTION;

UPDATE Employee SET Salary = Salary + 500 WHERE DepartmentID = 101;
INSERT INTO SalaryLog (EmployeeID, Amount) VALUES (123, 500);

COMMIT;
```

Example 2: Using Python and SQLAlchemy

```python
# Assuming the same setup as the previous Python examples

# Example of COMMIT operation in Python using SQLAlchemy
try:
    session.begin()

    employee = session.query(Employee).filter_by(department_id=101).first()
    employee.salary += 500

    salary_log = SalaryLog(employee_id=123, amount=500)
    session.add(salary_log)

    session.commit()

except Exception as e:
    print(f"Error: {e}")
    session.rollback()
finally:
    session.close()
```

6. ROLLBACK Operation:

The `ROLLBACK` operation undoes the changes made during a transaction, reverting the database to its state before the transaction started.

Example 1: Using SQL

```sql
-- Example of ROLLBACK operation in SQL
BEGIN TRANSACTION;

UPDATE Product SET Stock = Stock - 10 WHERE ProductID = 456;
INSERT INTO OrderLog (ProductID, Quantity) VALUES (456, 10);

-- Simulate an error to trigger a rollback
ROLLBACK;
```

Example 2: Using Python and SQLAlchemy

```python
# Assuming the same setup as the previous Python examples

# Example of ROLLBACK operation in Python using SQLAlchemy
try:
    session.begin()

    product = session.query(Product).filter_by(id=456).first()
    product.stock -= 10

    order_log = OrderLog(product_id=456, quantity=10)
    session.add(order_log)

    # Simulate an error to trigger a rollback
    raise Exception("Simulated error")

    session.commit()

except Exception as e:
    print(f"Error: {e}")
    session.rollback()
finally:
    session.close()
```

In these examples, the `COMMIT` operation is used to make changes permanent, while the `ROLLBACK` operation is used to undo changes and maintain the atomicity of the transaction. Note that the error in the second `ROLLBACK` example is simulated to trigger the `ROLLBACK` operation, ensuring that the changes are not committed.