1. Creating a Table (`CREATE TABLE`)

```
CREATE TABLE table_name (
   column1 datatype [constraints],
   column2 datatype [constraints],
   ...
);
```

Example:

Suppose you want to create a table named `Employees` with the following columns:

- `EmployeeID`: An integer that serves as the primary key.
- `FirstName`: A variable character string up to 50 characters, cannot be null.
- `LastName`: A variable character string up to 50 characters, cannot be null.
- `Email`: A variable character string up to 100 characters, must be unique.
- `HireDate`: A date field.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE,
    HireDate DATE
);
```

2/ Dropping a Table (`DROP TABLE`)

The `DROP TABLE` statement removes a table and all of its data from the database permanently. **Use this command with caution** as it cannot be undone.

Syntax:

```
DROP TABLE table_name;
```

Example:

Dropping the `Staff` table.

```
DROP TABLE Staff;
```

Additional Options:

- **IF EXISTS**: To prevent errors if the table does not exist, you can use the `IF EXISTS` clause (supported in many DBMS).

```
DROP TABLE IF EXISTS Staff;
```

3/ INSERT

1/ Basic `INSERT` Syntax

The `INSERT INTO` statement adds a new record to a table. The simplest form involves specifying the table name and providing values for all columns in the order they appear in the table.

Syntax:

```
INSERT INTO table_name
VALUES (value1, value2, ...);
```

- **`table_name`**: The name of the table where the record will be inserted.
- **`VALUES`**: Specifies the data to insert.
- **`value1, value2, ...`**: The actual data corresponding to each column in the table.

Example:

Suppose you have a table named `Employees` with the following columns:

- `EmployeeID` (INT, Primary Key)
- `FirstName` (VARCHAR(50), NOT NULL)
- `LastName` (VARCHAR(50), NOT NULL)
- `Email` (VARCHAR(100), UNIQUE)
- `HireDate` (DATE)

To insert a new employee record:

```
INSERT INTO Employees
VALUES (1, 'John', 'Doe', 'john.doe@example.com', '2023-01-15');
```

2. Inserting Multiple Records

You can insert multiple records in a single `INSERT` statement by separating each set of values with commas.

Syntax:

```
INSERT INTO table_name

VALUES

    (value1a, value2a, ...),

    (value1b, value2b, ...),

    ...;
```

Example:

Insert two new employee records:

```
INSERT INTO Employees

VALUES

    (2, 'Jane', 'Smith', 'jane.smith@example.com', '2023-02-20'),

    (3, 'Michael', 'Johnson', 'michael.johnson@example.com', '2023-03-10');
```

3. Inserting Specific Columns

Sometimes, you may not want to insert data into all columns of a table, especially if some columns have default values or allow `NULL`. In such cases, you can specify the columns you want to insert data into.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)

VALUES (value1, value2, ...);
```

Example:

Suppose the `HireDate` has a default value of the current date, and you want to insert a new employee without specifying the `HireDate`:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Email)

VALUES (4, 'Emily', 'Davis', 'emily.davis@example.com');
```

4. Inserting Data from Another Table (`INSERT INTO ... SELECT`)

You can insert data into a table by selecting data from another table using the `SELECT` statement. This is useful for copying data or transforming data during insertion.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
SELECT columnA, columnB, ...
FROM another_table
WHERE condition;
```

Example:

Assume you have another table `FormerEmployees` and you want to archive their records into `Employees` after verifying they meet certain criteria:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Email, HireDate)
SELECT EmployeeID, FirstName, LastName, Email, HireDate
FROM FormerEmployees
WHERE TerminationDate < '2022-12-31';
```

Basic `DELETE` Syntax

The simplest form of the `DELETE` statement removes all records from a table. However, it's crucial to use the `WHERE` clause to specify which records to delete; otherwise, **all records will be removed**.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

- **`table_name`**: The name of the table from which you want to delete records.
- **`WHERE condition`**: Specifies the criteria that determine which records to delete.

Example:

Suppose you have a table named `Employees`:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE,
    HireDate DATE
);
```

To delete the record of the employee with `EmployeeID` 1:

```
DELETE FROM Employees
WHERE EmployeeID = 1;
```

Deleting Multiple Records

You can delete multiple records that meet specific criteria by using conditions that match multiple rows.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example:

Delete all employees hired before January 1, 2023:

```
DELETE FROM Employees
WHERE HireDate < '2023-01-01';
```

Key Points about the `WHERE` Clause

1. **Purpose:**
    - **Filtering Data:** Restricts the result set to only those rows that meet the specified conditions.
    - **Used With:** Commonly used in `SELECT`, `UPDATE`, `DELETE`, and `INSERT` (with conditions) statements.

Basic Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;

UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;


DELETE FROM table_name
WHERE condition;
```

Common Operators:

- **Comparison Operators:**
  - `=` : Equal to
  - `<>` or `!=` : Not equal to
  - `>` : Greater than
  - `<` : Less than
  - `>=` : Greater than or equal to
  - `<=` : Less than or equal to
- **Logical Operators:**
  - `AND` : Combines multiple conditions; all must be true
  - `OR` : Combines multiple conditions; at least one must be true
  - `NOT` : Negates a condition
- **Pattern Matching:**
  - `LIKE` : Searches for a specified pattern (e.g., `%` for wildcards)
- **Set Membership:**
  - `IN` : Checks if a value matches any value in a list
- **Range Checking:**
  - `BETWEEN` : Checks if a value falls within a range

Examples:
Selecting Specific Records:

```
SELECT * FROM Employees
WHERE Department = 'Sales';
```

Using Multiple Conditions:

```
SELECT FirstName, LastName FROM Employees
```

```
WHERE Department = 'Sales' AND HireDate > '2022-01-01';
```

- *Retrieves first and last names of employees in Sales hired after January 1, 2022.*

Pattern Matching with `LIKE`:

```
SELECT * FROM Employees
WHERE Email LIKE '%@example.com';
```

- *Retrieves all employees with email addresses ending in @example.com.*

Deleting Specific Records:

```
DELETE FROM Employees
WHERE EmployeeID = 10;
```

- *Deletes the employee with EmployeeID 10.*

Best Practices:

- **Always Use Precise Conditions:** To avoid unintended data manipulation, ensure that your `WHERE` clause accurately targets the desired records.

Test with `SELECT` First: Before performing `UPDATE` or `DELETE` operations, run a `SELECT` query with the same `WHERE` conditions to verify which records will be affected.

```
SELECT * FROM Employees
WHERE Department = 'Sales' AND HireDate > '2022-01-01';
```

-

Use Parentheses for Complex Conditions: When combining multiple logical operators, use parentheses to clarify the order of evaluation.

```
SELECT * FROM Employees
```

- `WHERE (Department = 'Sales' OR Department = 'Marketing') AND HireDate > '2022-01-01';`

AGGrigate

1. `ORDER BY` Clause

Purpose:

Sorts the result set of a `SELECT` query based on one or more columns, either in ascending (`ASC`) or descending (`DESC`) order.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

Example:

```
SELECT FirstName, LastName, HireDate
FROM Employees
ORDER BY HireDate DESC;
```

Sorts employees by their hire date in descending order (newest first).

---

2. `COUNT` Function

Purpose:

Returns the number of rows that match a specified condition. Useful for counting records in a table or grouped data.

Syntax:

```
SELECT COUNT(column_name) AS AliasName
FROM table_name
WHERE condition;
```

Example:

```
SELECT COUNT(EmployeeID) AS TotalEmployees
FROM Employees;
```

Counts the total number of employees.

---

3. `MAX` Function

Purpose:

Retrieves the maximum value from a specified column.

Syntax:

```
SELECT MAX(column_name) AS AliasName
FROM table_name
WHERE condition;
```

Example:

```
SELECT MAX(Salary) AS HighestSalary
FROM Employees;
```

Finds the highest salary among employees.

---

4. `MIN` Function

Purpose:

Retrieves the minimum value from a specified column.

Syntax:

```
SELECT MIN(column_name) AS AliasName
FROM table_name
WHERE condition;
```

Example:

```
SELECT MIN(HireDate) AS EarliestHireDate
FROM Employees;
```

Finds the earliest hire date among employees.

---

5. `SUM` Function

Purpose:

Calculates the total sum of a numeric column.

Syntax:

```
SELECT SUM(column_name) AS AliasName
FROM table_name
WHERE condition;
```

Example:

```
SELECT SUM(Salary) AS TotalSalaries
FROM Employees;
```

Calculates the total sum of all employees' salaries.

---

6. `AVG` Function

Purpose:

Calculates the average value of a numeric column.

Syntax:

```
SELECT AVG(column_name) AS AliasName
FROM table_name
WHERE condition;
```

Example:

```
SELECT AVG(Salary) AS AverageSalary
FROM Employees;
```

Calculates the average salary of employees.

---

7. `DISTINCT` Keyword

Purpose:

Eliminates duplicate values in the result set, returning only unique records.

Syntax:

```
SELECT DISTINCT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
SELECT DISTINCT Department
FROM Employees;
```

Retrieves a list of unique departments from the Employees table.

---

8. MySQL Aliases

Purpose:

Provides temporary names to columns or tables within a query for better readability or convenience.

Syntax:

Column Alias:

```
SELECT column_name AS AliasName
FROM table_name;
```

Table Alias:

```
SELECT t.column1, t.column2
FROM table_name AS t;
```

Example:

```
SELECT FirstName AS Name, Salary AS EmployeeSalary
FROM Employees;
```

Renames `FirstName` to `Name` and `Salary` to `EmployeeSalary` in the result set.

---

9. `BETWEEN` Operator

Purpose:

Filters the result set to include values within a specified range (inclusive).

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example:

```
SELECT FirstName, LastName, HireDate
FROM Employees
WHERE HireDate BETWEEN '2022-01-01' AND '2023-12-31';
```

Selects employees hired between January 1, 2022, and December 31, 2023.

---

10. `AND` Operator

Purpose:

Combines multiple conditions in a `WHERE` clause. All conditions separated by `AND` must be true for a row to be included in the result set.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND ...;
```

Example:

```
SELECT FirstName, LastName, Department
FROM Employees
WHERE Department = 'Sales' AND Salary > 50000;
```

Putting It All Together: Practical Example

Scenario: Retrieve the number of unique departments, the highest and average salary per department, sorted by the total number of employees in each department.

```
SELECT
    Department AS Dept,
    COUNT(EmployeeID) AS NumEmployees,
    MAX(Salary) AS HighestSalary,
    AVG(Salary) AS AvgSalary
FROM Employees
WHERE Salary BETWEEN 30000 AND 120000
GROUP BY Department
ORDER BY NumEmployees DESC;
```

Explanation:

- **`DISTINCT` Alternative:** If you wanted to count unique departments, `COUNT(DISTINCT Department)` could be used.
- **Aliases:** `Department` is aliased as `Dept`, and aggregate results are given meaningful aliases.
- **`BETWEEN`:** Filters salaries between 30,000 and 120,000.
- **`AND`:** Not used directly here but can be added for additional conditions.
- **`GROUP BY`:** Groups results by department.

- `ORDER BY`**:** Sorts the results by the number of employees in descending order.

Summary

- `ORDER BY` **Clause:** Sorts query results based on specified columns in ascending or descending order.
- **Aggregate Functions (**`COUNT`**,** `MAX`**,** `MIN`**,** `SUM`**,** `AVG`**):** Perform calculations on a set of values to return a single value.
- `DISTINCT` **Keyword:** Ensures that only unique values are returned in the result set.
- **MySQL Aliases:** Temporary names for columns or tables to enhance readability.
- `BETWEEN` **Operator:** Filters records within a specific range.
- `AND` **Operator:** Combines multiple conditions that must all be true for records to be selected.

By mastering these SQL components, you can perform more precise and efficient data retrieval and analysis in your databases.

Feel free to ask if you need further clarification or additional examples!

What is Normalization?

Normalization is the process of organizing the attributes and tables of a relational database to minimize data redundancy and improve data integrity. It involves decomposing tables into smaller, more manageable pieces without losing information, ensuring that the database remains efficient and scalable.

Benefits of Normalization

1. **Eliminates Redundancy:** Reduces duplicate data, saving storage space.
2. **Prevents Anomalies:** Avoids insertion, deletion, and update anomalies that can lead to inconsistent data.
3. **Enhances Data Integrity:** Ensures that data dependencies make sense and are logical.
4. **Improves Query Performance:** Streamlined tables can lead to faster query processing.
5. **Facilitates Maintenance:** Easier to update and maintain the database structure.

Normal Forms Overview

Normalization is achieved through a series of rules known as **Normal Forms**. Each normal form addresses specific types of issues:

1. **First Normal Form (1NF):** Ensures that the table has a primary key and that all columns contain atomic (indivisible) values.
2. **Second Normal Form (2NF):** Achieves 1NF and ensures that all non-key attributes are fully functionally dependent on the primary key.
3. **Third Normal Form (3NF):** Achieves 2NF and ensures that all non-key attributes are not only fully dependent on the primary key but also **transitively independent** (i.e., no dependency on other non-key attributes).
4. **Boyce-Codd Normal Form (BCNF):** A stricter version of 3NF where every determinant is a candidate key.

Let's delve into each normal form with definitions and examples.

First Normal Form (1NF)

Definition

A table is in **First Normal Form (1NF)** if:

1. **Atomicity:** Each column contains only atomic (indivisible) values; there are no repeating groups or arrays.
2. **Uniqueness:** Each row is unique, typically enforced by a primary key.

Example

Unnormalized Table:

OrderID	CustomerName	Products
1	John Doe	Laptop, Mouse
2	Jane Smith	Smartphone
3	Bob Johnson	Tablet, Keyboard, Charger

Issues:

- The `Products` column contains multiple values, violating atomicity.

Converting to 1NF

To convert to 1NF, ensure that each field contains only one value. This often involves creating separate rows for each product in an order.

1NF Compliant Table:

OrderID	CustomerName	Product
1	John Doe	Laptop
1	John Doe	Mouse
2	Jane Smith	Smartphone
3	Bob Johnson	Tablet
3	Bob Johnson	Keyboard
3	Bob Johnson	Charger

Explanation:

- The `Products` column has been split into individual `Product` entries, ensuring atomicity.
- `OrderID` combined with `Product` can serve as a composite primary key to ensure row uniqueness.

Second Normal Form (2NF)

Definition

A table is in **Second Normal Form (2NF)** if:

1. **It is in 1NF.**
2. **Full Functional Dependency:** All non-key attributes are fully functionally dependent on the entire primary key (i.e., no partial dependency on a part of the composite primary key).

Key Concepts

- **Functional Dependency:** A relationship where one attribute uniquely determines another.
- **Partial Dependency:** A non-key attribute depends only on part of a composite primary key.

Example

1NF Compliant Table:

OrderID	Product	CustomerName	CustomerAddress
1	Laptop	John Doe	123 Elm Street
1	Mouse	John Doe	123 Elm Street

2	Smartphone	Jane Smith	456 Oak Avenue
3	Tablet	Bob Johnson	789 Pine Road
3	Keyboard	Bob Johnson	789 Pine Road
3	Charger	Bob Johnson	789 Pine Road

Issues:

- The composite primary key is (`OrderID`, `Product`).
- `CustomerName` and `CustomerAddress` depend only on `OrderID`, not on `Product`.
- This creates partial dependencies.

Converting to 2NF

To eliminate partial dependencies, decompose the table into two:

1. **Orders Table:** Contains information about orders.
2. **OrderDetails Table:** Contains information about products in each order.

Orders Table:

OrderID	CustomerName	CustomerAddress
1	John Doe	123 Elm Street
2	Jane Smith	456 Oak Avenue
3	Bob Johnson	789 Pine Road

OrderDetails Table:

OrderID	Product
1	Laptop
1	Mouse
2	Smartphone
3	Tablet
3	Keyboard
3	Charger

Explanation:

- **Orders Table:** `OrderID` is the primary key. `CustomerName` and `CustomerAddress` depend fully on `OrderID`.
- **OrderDetails Table:** The composite primary key is (`OrderID`, `Product`). No partial dependencies exist as all attributes depend on the entire primary key.

Third Normal Form (3NF)

Definition

A table is in **Third Normal Form (3NF)** if:

1. **It is in 2NF.**
2. **No Transitive Dependencies:** No non-key attribute depends on another non-key attribute.

Key Concepts

- **Transitive Dependency:** A non-key attribute depends on another non-key attribute, which in turn depends on the primary key.

Example

2NF Compliant Tables:

1. **Orders Table:**

OrderID	CustomerID	OrderDate
1	C001	2023-01-15
2	C002	2023-02-20
3	C003	2023-03-10

2. **Customers Table:**

CustomerID	CustomerName	CustomerAddress
C001	John Doe	123 Elm Street
C002	Jane Smith	456 Oak Avenue
C003	Bob Johnson	789 Pine Road

3. **OrderDetails Table:**

OrderID	Product
1	Laptop
1	Mouse
2	Smartphone
3	Tablet
3	Keyboard
3	Charger

Issues (Before 3NF):

Assume an initial 2NF table that includes:

OrderID	CustomerID	CustomerName	CustomerAddress	OrderDate	Product
1	C001	John Doe	123 Elm Street	2023-01-15	Laptop
1	C001	John Doe	123 Elm Street	2023-01-15	Mouse
2	C002	Jane Smith	456 Oak Avenue	2023-02-20	Smartphone
3	C003	Bob Johnson	789 Pine Road	2023-03-10	Tablet
3	C003	Bob Johnson	789 Pine Road	2023-03-10	Keyboard
3	C003	Bob Johnson	789 Pine Road	2023-03-10	Charger

Here, `CustomerName` and `CustomerAddress` depend on `CustomerID`, not directly on `OrderID` or the composite key (`OrderID`, `Product`), creating transitive dependencies.

Converting to 3NF

To eliminate transitive dependencies, further decompose the tables:

1. **Create a separate `Customers` table** where customer details depend solely on `CustomerID`.
2. **Modify the `Orders` table** to reference `CustomerID` instead of including customer details.

3NF Compliant Tables:

1. **Orders Table:**

OrderID	CustomerID	OrderDate

1	C001	2023-01-15
2	C002	2023-02-20
3	C003	2023-03-10

2. **Customers Table:**

CustomerID	**CustomerName**	**CustomerAddress**
C001	John Doe	123 Elm Street
C002	Jane Smith	456 Oak Avenue
C003	Bob Johnson	789 Pine Road

3. **OrderDetails Table:**

OrderID	**Product**
1	Laptop
1	Mouse
2	Smartphone
3	Tablet
3	Keyboard
3	Charger

Explanation:

- **Orders Table:** Now only contains `OrderID`, `CustomerID`, and `OrderDate`. There's no transitive dependency because customer details are moved to the `Customers` table.
- **Customers Table:** All customer-related information depends solely on `CustomerID`.
- **OrderDetails Table:** Remains unchanged, focusing on the relationship between orders and products.

Boyce-Codd Normal Form (BCNF)

Definition

Boyce-Codd Normal Form (BCNF) is a stricter version of the Third Normal Form. A table is in **BCNF** if:

1. **It is in 3NF.**
2. **Every determinant is a candidate key.**

Key Concepts

- **Determinant:** An attribute or a set of attributes that uniquely determines another attribute.
- **Candidate Key:** A minimal set of attributes necessary to uniquely identify a row.

BCNF Addresses Situations Where:

Even if a table is in 3NF, it might still have anomalies if there are dependencies where non-candidate keys determine other attributes. BCNF ensures that all functional dependencies are based on candidate keys.

Example

Consider a University Database:

Table: InstructorCourses

InstructorID	CourseID	Room
I001	C101	R1
I001	C102	R2
I002	C101	R1
I003	C103	R3

Assumptions:

1. **Functional Dependencies:**
    - `InstructorID`, `CourseID` → `Room` (Composite primary key: `InstructorID`, `CourseID`)
    - `CourseID` → `Room` (Each course is assigned to a specific room)
2. **Candidate Keys:**
    - (`InstructorID`, `CourseID`)
    - (`CourseID`, `Room`) is not a candidate key as `CourseID` uniquely determines `Room`.

Issues:

- The dependency `CourseID` → `Room` violates BCNF because `CourseID` is not a candidate key for the table.

Converting to BCNF

To achieve BCNF, decompose the table to ensure that every determinant is a candidate key.

Decomposed Tables:

1. **CourseRooms Table:**

CourseID	**Room**
C101	R1
C102	R2
C103	R3

2. **InstructorCourses Table:**

InstructorID	**CourseID**
I001	C101
I001	C102
I002	C101
I003	C103

Explanation:

- **CourseRooms Table:** Now `CourseID` is a candidate key, and `CourseID` → `Room` satisfies BCNF.
- **InstructorCourses Table:** The composite primary key (`InstructorID`, `CourseID`) ensures that all dependencies are based on candidate keys.

Result:

Both tables are now in BCNF, eliminating the anomaly caused by the original table.

Normalization Process: An Example

Let's walk through a complete normalization process from an unnormalized table to BCNF.

Scenario

Business Context:

A small bookstore maintains information about book sales, including details about sales transactions, books, authors, and publishers.

Unnormalized Table:

TransactionID	BookTitle	AuthorName	PublisherName	PublisherAddress	Quantity	SaleDate
T001	SQL Basics	John Smith	TechBooks	100 Tech Ave	3	2023-01-15
T002	Advanced SQL	Jane Doe	TechBooks	100 Tech Ave	2	2023-01-16
T003	Cooking 101	Mary Johnson	HomePress	200 Home St	5	2023-01-17
T004	Gardening Tips	Mary Johnson	HomePress	200 Home St	1	2023-01-18
T005	SQL Basics	John Smith	TechBooks	100 Tech Ave	4	2023-01-19

Issues:

- **Redundancy:** `PublisherName` and `PublisherAddress` are repeated for each book.
- **Non-Atomicity:** All data appears atomic, so 1NF is satisfied.
- **Functional Dependencies:**
  - `TransactionID` → `BookTitle`, `AuthorName`, `PublisherName`, `PublisherAddress`, `Quantity`, `SaleDate`
  - `BookTitle` → `AuthorName`, `PublisherName`, `PublisherAddress`
  - `PublisherName` → `PublisherAddress`

Step 1: Ensure 1NF

The table already satisfies **1NF** as all columns contain atomic values and each row is unique (assuming `TransactionID` is unique).

Step 2: Convert to 2NF

Identify partial dependencies. The primary key is `TransactionID`. However, all non-key attributes depend fully on `TransactionID`, so technically, it's already in 2NF. However, given the dependencies, further normalization can help reduce redundancy.

Alternatively, if considering a composite key (e.g., `BookTitle`, `SaleDate`), then dependencies might create partial dependencies. To avoid confusion, we'll proceed to decompose based on functional dependencies.

Step 3: Convert to 3NF

Eliminate transitive dependencies.

Identify Transitive Dependencies:

- `BookTitle` → `AuthorName`, `PublisherName`, `PublisherAddress`
- `PublisherName` → `PublisherAddress`

Decompose into 3NF Compliant Tables:

1. **Transactions Table:**

TransactionID	BookTitle	Quantity	SaleDate
T001	SQL Basics	3	2023-01-15
T002	Advanced SQL	2	2023-01-16
T003	Cooking 101	5	2023-01-17
T004	Gardening Tips	1	2023-01-18
T005	SQL Basics	4	2023-01-19

2. **Books Table:**

BookTitle	AuthorName	PublisherName
SQL Basics	John Smith	TechBooks
Advanced SQL	Jane Doe	TechBooks
Cooking 101	Mary Johnson	HomePress
Gardening Tips	Mary Johnson	HomePress

3. **Publishers Table:**

PublisherName	PublisherAddress
TechBooks	100 Tech Ave
HomePress	200 Home St

Explanation:

- **Transactions Table:** Focuses solely on transactions, linking to `BookTitle`.

- **Books Table:** Contains book-specific information, linking each book to its author and publisher.
- **Publishers Table:** Contains publisher-specific information, ensuring that `PublisherAddress` is determined by `PublisherName` alone.

Step 4: Convert to BCNF

Ensure that every determinant is a candidate key.

Check Each Table:

1. **Transactions Table:**
    - **Primary Key:** `TransactionID`
    - **Functional Dependencies:** `TransactionID` → `BookTitle`, `Quantity`, `SaleDate`
    - **All determinants are candidate keys.**
2. **Status:** BCNF compliant.
3. **Books Table:**
    - **Primary Key:** `BookTitle`
    - **Functional Dependencies:**
        - `BookTitle` → `AuthorName`, `PublisherName`
    - **No determinants other than candidate keys.**
4. **Status:** BCNF compliant.
5. **Publishers Table:**
    - **Primary Key:** `PublisherName`
    - **Functional Dependencies:**
        - `PublisherName` → `PublisherAddress`
    - **No determinants other than candidate keys.**
6. **Status:** BCNF compliant.

Result:

All tables are now in **BCNF**, eliminating any remaining anomalies and ensuring optimal structure.