Database Normalization

Yehor KOROTENKO

29th Apr, 2025

1 Database Normalization Explained

Database normalization is a process of organizing data in a database to reduce redundancy and improve data integrity. It involves dividing databases into two or more tables and defining relationships between the tables. The primary goal is to isolate data so that updates, insertions, and deletions to the database can be made efficiently and without creating inconsistencies.

Think of it like organizing your closet. Instead of piling all your clothes in one big heap, you separate them into categories (shirts, pants, socks) and organize within each category (color, size, season). This makes it easier to find what you need and keeps your closet tidy.

Benefits of Normalization:

- **Reduced data redundancy:** Saves storage space and improves efficiency.
- **Improved data integrity:** Ensures that data is consistent and accurate.
- **Simplified data modifications:** Easier to update, insert, and delete data without affecting other parts of the database.
- **Better query performance:** Normalized data is typically easier to query and analyze.
- **Enhanced data security:** Can help enforce security constraints more effectively.

Normalization Forms

Normalization is achieved through a series of "normal forms" (1NF, 2NF, 3NF, BCNF, 4NF, 5NF). Each normal form builds upon the previous one. While higher normal forms exist, 3NF is generally considered sufficient for most practical database designs. BCNF (Boyce-Codd Normal Form) is a stricter version of 3NF and is used in specific cases.

Let's look at the most common normal forms with examples:

1. First Normal Form (1NF)

- **Rule:** Eliminate repeating groups of data within a table. Each column should contain only atomic (indivisible) values.

Example (Unnormalized Table):

`Orders` Table (Violates 1NF)

OrderID	CustomerID	ProductName	Quantity	Price
1	101	Laptop, Mouse	1, 2	1200, 25
2	102	Keyboard	1	75
3	101	Monitor, Speakers	1, 2	300, 50

Problem: The `ProductName`, `Quantity`, and `Price` columns contain multiple values for a single order. This makes it difficult to query for specific products and their associated details.

Solution (1NF Table):

`Orders` Table (1NF)

OrderID	CustomerID	ProductName	Quantity	Price
1	101	Laptop	1	1200
1	101	Mouse	2	25
2	102	Keyboard	1	75
3	101	Monitor	1	300
3	101	Speakers	2	50

, each row represents a single product within an order, and the columns contain atomic values.

2. Second Normal Form (2NF)

- **Rule:** Must be in 1NF. Eliminate redundant data that depends only on part of the primary key (partial dependency). 2NF applies only when the primary key is a composite key (made up of two or more columns).

Example (1NF but Violates 2NF):

`OrderDetails` Table (Violates 2NF - Assuming OrderID, ProductID are Primary Key)

OrderID	ProductID	ProductName	Quantity	UnitPrice
1	1	Laptop	1	1200
1	2	Mouse	2	25
2	1	Keyboard	1	75
3	3	Monitor	1	300
3	4	Speakers	2	50

Problem: The primary key is (OrderID, ProductID). The `ProductName` depends *only* on `ProductID`, not on the entire primary key (OrderID, ProductID). This means the same `ProductName` will be repeated every time the same `ProductID` is used in a different order, causing redundancy.

Solution (2NF Tables):

`OrderDetails` Table (2NF)

OrderID	ProductID	Quantity	UnitPrice
1	1	1	1200
1	2	2	25
2	1	1	75
3	3	1	300
3	4	2	50

`Products` Table (2NF)

ProductID	ProductName
1	Laptop
2	Mouse
3	Monitor
4	Speakers

've created a separate `Products` table to store product information. The `OrderDetails` table now references the `Products` table using the `ProductID` as a foreign key. The `ProductName` is now only stored once for each product.

3. Third Normal Form (3NF)

- **Rule:** Must be in 2NF. Eliminate redundant data that depends on non-key attributes (transitive dependency). A transitive dependency occurs when a non-key attribute determines another non-key attribute.

Example (2NF but Violates 3NF):

`Employees` Table (Violates 3NF - Assuming EmployeeID is Primary Key)

EmployeeID	EmployeeName	DepartmentID	DepartmentName	DepartmentLocation
1	John Smith	101	Sales	New York
2	Jane Doe	102	Marketing	Los Angeles
3	Peter Jones	101	Sales	New York

Problem: The primary key is `EmployeeID`. The `DepartmentName` and `DepartmentLocation` depend on `DepartmentID`, which is a non-key attribute. If the Sales department moves to Chicago, you'd have to update multiple rows in the `Employees` table.

Solution (3NF Tables):

`Employees` Table (3NF)

EmployeeID	EmployeeName	DepartmentID
1	John Smith	101
2	Jane Doe	102
3	Peter Jones	101

`Departments` Table (3NF)

DepartmentID	DepartmentName	DepartmentLocation
101	Sales	New York
102	Marketing	Los Angeles

, the department information is stored only once in the `Departments` table, and the `Employees` table references the `Departments` table using `DepartmentID` as a foreign key. If the Sales department moves, you only need to update one row in the `Departments` table.

Example Summary:

Imagine you're designing a database for a library. Here's how normalization might apply:

- **Unnormalized:** A single table with book information, author information, and borrower information all mixed together, leading to significant redundancy and potential inconsistencies.
- **1NF:** Breaking the table down to have atomic values in each column. Removing multiple authors from a single "Author" field.
- **2NF:** Separating author information into a separate `Authors` table, related to the `Books` table by `AuthorID`.
- **3NF:** If you also stored publisher information (publisher name, publisher address) in the `Books` table, you'd create a `Publishers` table and link it to `Books` by `PublisherID`.

Beyond 3NF (Briefly):

- **Boyce-Codd Normal Form (BCNF):** A stricter version of 3NF. Addresses rare cases where multiple overlapping candidate keys can cause redundancy. If every determinant (attribute or set of attributes that determines another attribute) is a candidate key, the relation is in BCNF.
- **4NF:** Deals with multi-valued dependencies (a column having multiple independent values for one instance of a different column).
- **5NF:** Deals with join dependencies.

Important Considerations:

- **Denormalization:** While normalization is generally good, sometimes denormalization (intentionally introducing redundancy) is necessary to improve query performance, especially for reporting or analytical workloads. This should be done carefully and only when the benefits outweigh the drawbacks.
- **Practicality:** Don't blindly normalize to the highest possible form. Consider the specific requirements of your application and the trade-offs between normalization and performance. 3NF is often a good balance.
- **Database Management System (DBMS):** The DBMS provides features like foreign key constraints to enforce referential integrity and maintain relationships between tables.

In conclusion, database normalization is a fundamental concept in database design that helps ensure data integrity, reduce redundancy, and improve database performance. Understanding the different normal forms and applying them appropriately is crucial for building robust and scalable database applications. Remember to weigh the benefits of normalization against potential performance considerations and choose the level of normalization that best suits your specific needs.

Okay, let's dive deeper into Boyce-Codd Normal Form (BCNF). As I mentioned before, BCNF is a stricter version of 3NF. While 3NF eliminates transitive dependencies, BCNF addresses a specific type of anomaly that can still exist in 3NF tables when there are multiple overlapping *candidate keys*.

Key Concepts Refresher:

- **Candidate Key:** A minimal set of attributes that uniquely identifies a tuple (row) in a table. A table can have multiple candidate keys.
- **Primary Key:** The candidate key chosen to be the main identifier for the table.
- **Non-Key Attribute:** An attribute that is not part of any candidate key.
- **Determinant:** An attribute (or a set of attributes) that functionally determines another attribute. In other words, if you know the value of the determinant, you can uniquely determine the value of the dependent attribute. We write this as: `A ->B` (A determines B).
- **Functional Dependency:** The relationship between a determinant and the attribute it determines (as shown above).

BCNF Rule:

A table is in BCNF if and only if every determinant is a candidate key. In simpler terms, for every functional dependency `A ->B`, A must be a candidate key.

Why BCNF Matters:

The situations where BCNF differs from 3NF are relatively rare, but when they occur, they can lead to update anomalies that 3NF doesn't prevent. These anomalies arise because BCNF aims to eliminate all redundancy based on *any* key, not just the primary key.

Example: A Classic BCNF Violation

Let's consider a table to store information about courses, instructors, and textbooks:

`CourseInstructorTextbook` Table:

Course	Instructor	Textbook
Math101	Professor Smith	Calculus for Dummies
Math101	Professor Jones	Calculus for Dummies
CS101	Professor Brown	Intro to Programming
CS101	Professor Green	Intro to Java
Physics201	Professor Davis	Classical Mechanics

Assumptions:

1. A course can have multiple instructors.
2. A course can have multiple textbooks.
3. For a given course, each instructor uses *only* one specific textbook. (This is the crucial assumption that causes the problem.)
4. The same textbook might be used by different instructors for the same course.

Identifying the Keys and Functional Dependencies:

- **Candidate Keys:** {Course, Instructor} and {Course, Textbook} (Knowing the course and the instructor uniquely identifies the textbook being used *for that instructor* in that course, and knowing the course and the textbook uniquely identifies which instructor uses *that textbook* in that course.)
- **Functional Dependencies:**
  - `Course, Instructor ->Textbook` (The Course and Instructor determine the Textbook)
  - `Course, Textbook ->Instructor` (The Course and Textbook determine the Instructor)
  - `Textbook ->Instructor` (This comes from the assumption 3, the Textbook determines the Instructor. Given textbook, for a given course, the instrcutor is the same, assuming there is only one instructor).

Violation of BCNF:

The `CourseInstructorTextbook` table is in 3NF. Why? Because there are no transitive dependencies based on the primary key you might choose (say, {Course, Instructor}). But, it's *not* in BCNF.

- Consider the functional dependency `Textbook ->Instructor`. `Textbook` is a determinant (it determines `Instructor`).
- But `Textbook` is *not* a candidate key.

The Resulting Anomaly:

Suppose Professor Smith stops teaching Math101. To remove Professor Smith from the database, you would have to delete the row containing (Math101, Professor Smith, Calculus for Dummies). However, this *also* removes the information that `Calculus for Dummies` is used for Math101, even though Professor Jones might still be using it. This is an update anomaly.

Decomposition to BCNF:

To decompose the table into BCNF, we create two tables:

`CourseTextbook` Table:

Course	Textbook
Math101	Calculus for Dummies
CS101	Intro to Programming
CS101	Intro to Java
Physics201	Classical Mechanics

`InstructorTextbook` Table:

Textbook	Instructor
Calculus for Dummies	Professor Smith
Calculus for Dummies	Professor Jones
Intro to Programming	Professor Brown
Intro to Java	Professor Green
Classical Mechanics	Professor Davis

Explanation:

1. **CourseTextbook:** Stores the relationship between courses and the textbooks used. The primary key is {Course, Textbook}.
2. **InstructorTextbook:** Stores the relationship between textbooks and instructors using them. The primary key is {Textbook, Instructor}.

Now, if Professor Smith stops teaching Math101, you can remove the row (Calculus for Dummies, Professor Smith) from the `InstructorTextbook` table without losing the information that `Calculus for Dummies` is used in Math101 (which is stored in the `CourseTextbook` table). The anomaly is resolved.

Summary of the BCNF Process:

1. Identify all candidate keys.
2. Identify all functional dependencies.
3. If any determinant is *not* a candidate key, decompose the table. The decomposition usually involves creating a new table with the determinant as the key, and moving all attributes determined by that determinant into the new table. The original table then contains only the minimal set of attributes to maintain the relationships.

When to Worry About BCNF:

You primarily need to consider BCNF when:

- You have overlapping candidate keys.
- There are functional dependencies where a non-candidate key attribute determines another attribute.
- You want to absolutely minimize redundancy and eliminate update anomalies.

In practice, achieving BCNF can sometimes lead to more tables and more complex queries. As always, you need to weigh the benefits of BCNF against the potential performance impact and complexity. 3NF is often sufficient, but understanding BCNF allows you to recognize and address more subtle data integrity issues when they arise.

Example and explanation of the algorithm for the db normalisation is provided in this file