Comprehensive Guide to Database Normalization

What is Normalization?

Definition:

Normalization is the process of organizing data in a database to eliminate redundancy and ensure data integrity.

Why Normalization is Important?

- **Avoids data duplication**
- **Prevents anomalies** (insertion, update, deletion issues)
- Makes the database **efficient and clean**
- **Improves consistency** across related tables

Understanding Data Redundancy and Integrity

Data Redundancy

Definition: Data redundancy means storing the same piece of data more than once in a database.

Why it's bad:

- Wastes storage
- Causes inconsistency (e.g., you update one place but forget another)
- Makes data hard to maintain and update

Example of Data Redundancy:

Suppose we have a student table like this:

StudentID	Name	Course	Course_Fee
1	NK	Python	5000
2	Arun	Python	5000
3	Sam	Java	6000

Here, "Python" and its fee 5000 is repeated for every student who takes Python. This is redundant data.

Fix using normalization:

Create a separate Courses table:

```
CourseID | CourseName | Fee
---------|------------|-----
C1       | Python     | 5000
C2       | Java       | 6000
```

Now the Student table just stores `CourseID` – less repetition!

Data Integrity

Definition: Data integrity refers to the accuracy, consistency, and reliability of data in the database.

Types of Data Integrity:

Type	What it ensures
Entity Integrity	Each row has a unique and non-null primary key
Referential Integrity	Foreign keys must refer to valid rows
Domain Integrity	Values in a column must follow valid data types/rules

Example of Integrity Issues:

Referential Integrity Violation:
Suppose we insert a student with a `CourseID = C99` (but there's no such course in the Course table).
That would break referential integrity.

Solution: Use foreign key constraints to prevent invalid entries.

Constraints That Maintain Integrity:

- `PRIMARY KEY`: Ensures unique, non-null ID
- `FOREIGN KEY`: Links two tables and ensures relationships are valid
- `NOT NULL`: Column must always have a value
- `UNIQUE`: Prevents duplicate entries
- `CHECK`: Restricts values based on condition

Understanding Accuracy, Consistency, and Reliability

1. Accuracy

Definition: Accuracy means the data stored in the database is correct, precise, and error-free.

Example:

StudentID	Name	Age
1	NK	21

If your actual age is 21, this is accurate. If someone mistakenly enters 31, then it's inaccurate.

Accuracy ensures the real-world fact is correctly represented in the database.

2. Consistency

Definition: Consistency means that data does not conflict with itself across tables or records.

Example:
Suppose you have a student in the `Students` table with `CourseID = C1`, but `Courses` table does not have a `CourseID = C1`. This is inconsistent data!

Using foreign keys ensures that if `CourseID = C1` is used in one table, it must exist in the referenced table.

Consistency ensures logical correctness and rule compliance in relationships across tables.

3. Reliability

Definition: Reliability means the database should always return trustworthy, up-to-date, and dependable results even under stress (multiple users, transactions, etc.).

Example:

- If 3 users are updating the same row, the final data should still be valid (use of transactions).
- If the system crashes during an update, it should not leave half-updated data.

Reliability ensures that the system behaves predictably and correctly, even during errors or multi-user environments.

Understanding Database Anomalies

Definition: Anomalies are unexpected problems or errors that occur when inserting, updating, or deleting data in a poorly designed (non-normalized) database. They can lead to data inconsistency, loss, or corruption.

Types of Anomalies

There are 3 major types:

Type	Problem It Causes
Insertion Anomaly	Trouble inserting new data
Update Anomaly	Updating data in one place doesn't reflect elsewhere
Deletion Anomaly	Deleting a record removes unintended information

1. Insertion Anomaly

Problem: You cannot insert a record because some other data is missing.

Example:

```
Table: StudentCourse

StudentName | CourseName | Fee
------------|------------|-----
NK          | Python     | 5000
```

Suppose a new course Java is introduced, but no student has enrolled yet.
You can't insert `Java` into this table without a student, which is an insertion anomaly.

Solution: Separate `Courses` and `Students` into different tables using normalization.

2. Update Anomaly

Problem: You have to update the same data in multiple places. If you miss even one, data becomes inconsistent.

Example:

```
StudentName | CourseName | Fee
------------|------------|-----
NK          | Python     | 5000
Arun        | Python     | 5000
```

Now you want to update the Python course fee to 5500.
You must update every row with Python. If you miss one → inconsistent data.

Solution: Store course fee only once in a `Course` table.

3. Deletion Anomaly

Problem: Deleting one piece of data accidentally removes other valuable data.

Example:

```
StudentName | CourseName | Fee
------------|------------|-----
NK          | Python     | 5000
```

Now NK cancels the course → delete the row.
This also deletes info about the Python course and its fee, even though other students might be interested.

Solution: Store course info separately, so deleting a student doesn't delete the course.

How Normalization Works

It breaks a large, unstructured table into smaller related tables. Each normal form (NF) applies certain rules.

NORMAL FORMS EXPLAINED

1NF (First Normal Form)

Rule:

- Each column should have atomic (indivisible) values
- No repeating groups or arrays

Why: To ensure data is stored in its most basic form.

Example (Bad 1NF):

```
Student | Subjects
--------|---------------
NK      | Math, Physics
```

Fix (Good 1NF):

```
Student | Subject
--------|--------
NK      | Math
NK      | Physics
```

Understanding 1NF Further

A table is in 1NF if:

- Every column contains atomic (indivisible) values
- There are no repeating groups or arrays/lists in a single cell

Example (Not in 1NF):

StudentID	StudentName	Courses
101	NK	Python, Java
102	Arun	C++, JavaScript

Here, the `Courses` column has multiple values in one cell. This is not atomic.

Fix (1NF - Atomic Values):

StudentID	StudentName	Course
101	NK	Python
101	NK	Java
102	Arun	C++
102	Arun	JavaScript

Now each cell has a single value → This is in 1NF.

2NF (Second Normal Form)

Rule:

- Must be in 1NF
- No partial dependency (non-key attribute should depend on the whole primary key)

Why: To avoid duplication and ensure data depends entirely on the primary key.
Applies only if composite key is used.

Example (Bad Table - 1NF but not 2NF):

StudentID	CourseID	StudentName
101	C001	NK

Here, `StudentName` depends only on StudentID, not on the full (StudentID, CourseID) key → Partial Dependency

Fix:

- Student Table: `(StudentID, StudentName)`
- Enrollment Table: `(StudentID, CourseID)`

Understanding 2NF Further

A table is in 2NF if:

1. It is already in 1NF
2. There is no partial dependency of any non-prime attribute on a part of the primary key

Key Terms:

Term	Meaning
1NF	All values atomic, no repeating groups
Composite Key	A primary key made of two or more columns
Partial Dependency	When a non-key column depends on part of a composite key
Full Dependency	When a non-key column depends on the entire composite key

Another Example of Not in 2NF:

EmployeeProject Table:

EmpID	ProjectID	EmpName	ProjectName	Salary
E01	P01	NK	AI System	60000
E02	P01	Arun	AI System	70000

Composite Key: (EmpID, ProjectID)

Problems:

- `EmpName`, `Salary` depends only on `EmpID`
- `ProjectName` depends only on `ProjectID`

These are partial dependencies. Not in 2NF.

Fix: Break into separate tables

Employee Table:

EmpID	EmpName	Salary
E01	NK	60000
E02	Arun	70000

Project Table:

ProjectID	ProjectName
P01	AI System

Assignment Table:

EmpID	ProjectID
E01	P01
E02	P01

Now every attribute depends on a single primary key in each table — this is 2NF-compliant.

3NF (Third Normal Form)

Rule:

- Must be in 2NF
- No transitive dependency: non-key column should not depend on another non-key column

Why: Removes indirect relationships between non-key columns.

Example (Bad Table):

EmpID	EmpName	DeptID	DeptName
1	NK	D01	HR

Here, `DeptName` depends on `DeptID`, which is not a primary key.

Fix:

- Employee Table: `(EmpID, EmpName, DeptID)`
- Department Table: `(DeptID, DeptName)`

BCNF (Boyce-Codd Normal Form)

Rule:

- A stronger version of 3NF
- For every functional dependency (X → Y), X should be a super key

Why: Fixes anomalies not handled by 3NF when there are overlapping candidate keys.

Example Problem in 3NF:

StudentID	Course	Instructor
101	DBMS	Prof. A
101	OS	Prof. B
102	DBMS	Prof. A

Functional dependencies:

- `(StudentID, Course)` → `Instructor` (Composite key)
- `Course` → `Instructor` (Course determines Instructor, but Course is not a super key)

Hence, Course → Instructor violates BCNF, though it's fine in 3NF.

Fix:
Decompose into two tables:

Course Table:

Course	Instructor
DBMS	Prof. A
OS	Prof. B

StudentCourse Table:

StudentID	Course
101	DBMS
101	OS
102	DBMS

4NF (Fourth Normal Form)

Rule:

- Should be in BCNF
- No multi-valued dependencies

Why: Avoids cases where a row has two or more independent multivalued facts.

Example of Multi-Valued Dependency:

Student	Language	Hobby
NK	English	Reading
NK	English	Drawing
NK	French	Reading
NK	French	Drawing

Here:

- NK speaks 2 languages → English, French
- NK has 2 hobbies → Reading, Drawing
- They are independent, but repeated → MVD present

Fix:

Split into two independent tables:

StudentLanguages:

Student	Language
NK	English
NK	French

StudentHobbies:

Student	Hobby
NK	Reading
NK	Drawing

This eliminates unnecessary duplication and satisfies 4NF.

5NF (Fifth Normal Form)

Rule:

- Should be in 4NF
- No join dependency

Why: Breaks down tables to avoid lossless joins with complex relationships.

Example Scenario:

Suppose you store which:

- Model is made by which Manufacturer
- Model is sold in which Country

Manufacturer	Model	Country
Samsung	S21	USA
Samsung	S21	India
Apple	iPhone	USA
Apple	iPhone	India

You might be joining data from 3 different tables, and this data can be recombined via joins, so storing all three together is redundant.

Fix:
Decompose into 3 separate tables:

ManufacturerModel:

Manufacturer	Model
Samsung	S21
Apple	iPhone

ModelCountry:

Model	Country
S21	USA
S21	India
iPhone	USA
iPhone	India

ManufacturerCountry:

Manufacturer	Country
Samsung	USA
Samsung	India
Apple	USA
Apple	India

Now, you don't need the full 3-column table anymore — joins can reconstruct it without redundancy.

Interview Key Notes

- **1NF** = Atomic columns, no arrays or multivalued fields
- **2NF** = No partial dependencies on a composite key
- **3NF** = No transitive dependencies between non-key attributes
- **BCNF** = Every determinant is a candidate key
- Normalize till **3NF or BCNF** in most practical applications
- **De-normalization** may be used for performance in read-heavy apps
- **Use foreign keys** to connect normalized tables

Summary Table

Normal Form	Key Rule	Fixes
1NF	Atomic values, no repeating groups	Redundancy
2NF	No partial dependency (for composite keys)	Duplication
3NF	No transitive dependency	Indirect linkage
BCNF	Every determinant is a candidate key	Advanced anomaly
4NF	No multivalued dependency	Multi-fact rows
5NF	No join dependency	Complex joins

Real-Life Analogy

Imagine a grocery bill with repeated item details:

- Instead of writing item names, prices, and categories repeatedly for each bill →
- We create separate tables for items, categories, and just reference them using IDs in the bill table.