Informal guidelines to ensure the database is efficient, accurate, and easy to use.

1. Making Sure the Semantics of the Attributes is Clear in the Schema

- **Objective**: Each attribute should have a clear, unambiguous meaning within the schema.
- **Explanation**: The design should ensure that users understand what each attribute represents and how it relates to others. This avoids confusion and misinterpretation.
- **Example**: In a Student table, instead of vague column names like ID, Name, or Course, use more specific names like StudentID, FullName, and CourseName. Additionally, ensure that attributes that logically belong together (e.g., StudentID and FullName in the Student table) are not spread across multiple tables unless necessary.

2. Reducing the Redundant Information in Tuples

- **Objective**: Avoid storing duplicate data in multiple locations to prevent inconsistencies and reduce storage requirements.
- **Explanation**: Redundancy can lead to anomalies (insertion, update, and deletion anomalies) and complicate data maintenance. When redundant data exists, updating it in one place but forgetting another can lead to inconsistencies.
- **Example**: If both Employee and Department tables contain DepartmentName information, changing a department's name would require updating both tables. This redundancy can be eliminated by separating DepartmentName into a Department table with a unique DepartmentID, which is then referenced in the Employee table.

3. Reducing the NULL Values in Tuples

- **Objective**: Minimize the presence of NULL values to ensure data completeness and facilitate simpler queries.
- **Explanation**: NULL values can imply missing or inapplicable data, which can complicate queries and data analysis. A schema should be designed to minimize situations where data might be missing or inapplicable.
- **Example**: Instead of having a single Contact table with attributes like HomePhone, WorkPhone, and Email, use separate tables for each type of contact information or a table that supports multiple contact types. This avoids having NULL values for unused contact types, as only relevant data is stored.

4. Disallowing the Possibility of Generating Spurious Tuples

- **Objective**: Prevent erroneous results from joins due to poor schema design.
- **Explanation**: Spurious tuples (incorrect or duplicate rows) can be generated when tables are joined improperly. Ensuring that joins between tables use appropriate foreign keys and natural relationships reduces this risk.
- **Example**: Suppose you split a CustomerOrder table into two tables, Customer and Order, without properly connecting them via a CustomerID foreign key in Order. A join operation on Customer.Name = Order.CustomerName instead of a proper key may produce spurious

tuples or duplicate rows. Ensuring each table has clear relationships and foreign keys prevents these issues.

NORMALIZATION

1. Functional Dependencies

A **functional dependency** between two attributes (or sets of attributes) in a relation exists when the value of one attribute (or a set of attributes) uniquely determines the value of another attribute.

- **Notation**: A → B means that attribute B is functionally dependent on attribute A.
- **Example**: In a table Students, if each StudentID uniquely determines a Name, then there is a functional dependency StudentID → Name.

Functional dependencies are fundamental for understanding normalization because they identify relationships between attributes and are used to eliminate redundancy.

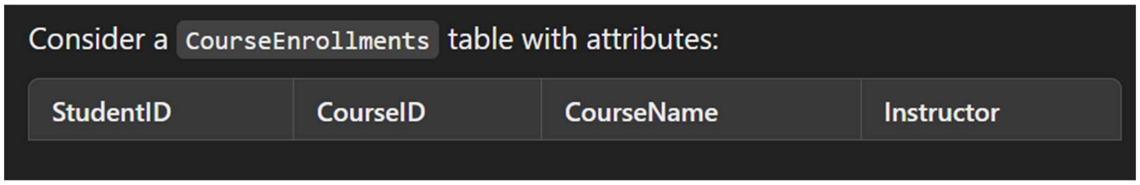
2. Second Normal Form (2NF)

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

- It is already in **First Normal Form (1NF)** (all values are atomic).
- Every **non-prime** attribute (an attribute that is not part of the primary key) is **fully functionally dependent** on the **entire primary key**.

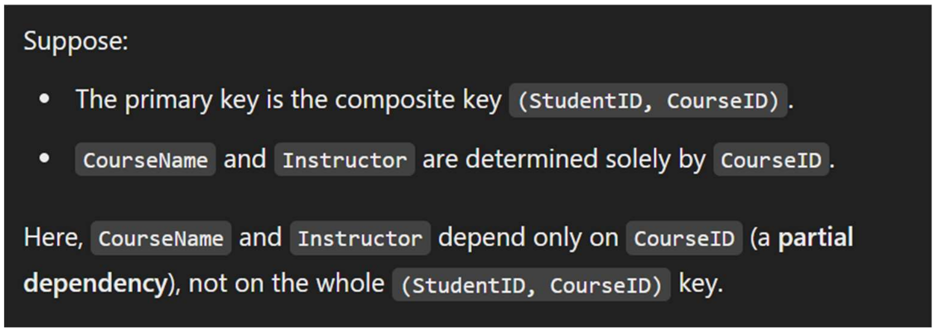
In other words, there should be no **partial dependencies**, where a non-prime attribute depends on only part of a composite primary key.

Example of 2NF



Consider a `CourseEnrollments` table with attributes:

StudentID	CourseID	CourseName	Instructor



Suppose:

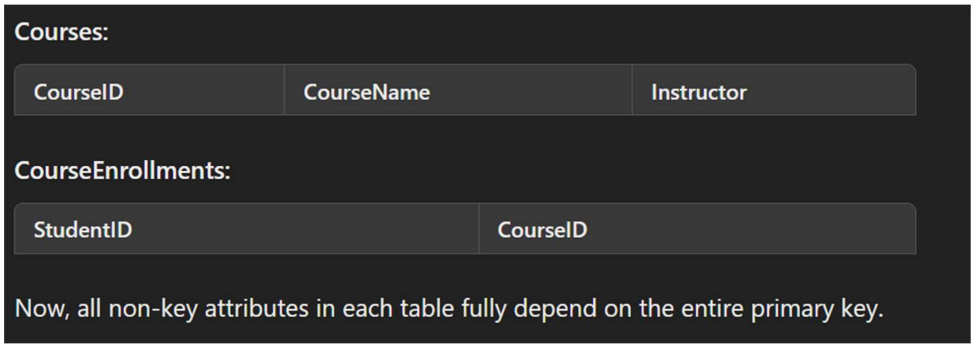
- The primary key is the composite key `(StudentID, CourseID)`.
- `CourseName` and `Instructor` are determined solely by `CourseID`.

Here, `CourseName` and `Instructor` depend only on `CourseID` (a **partial dependency**), not on the whole `(StudentID, CourseID)` key.

To achieve 2NF:

1. **Decompose** the table into two:
   - Courses: Stores (CourseID, CourseName, Instructor).

    - CourseEnrollments: Stores (StudentID, CourseID).



Courses:

CourseID	CourseName	Instructor

CourseEnrollments:

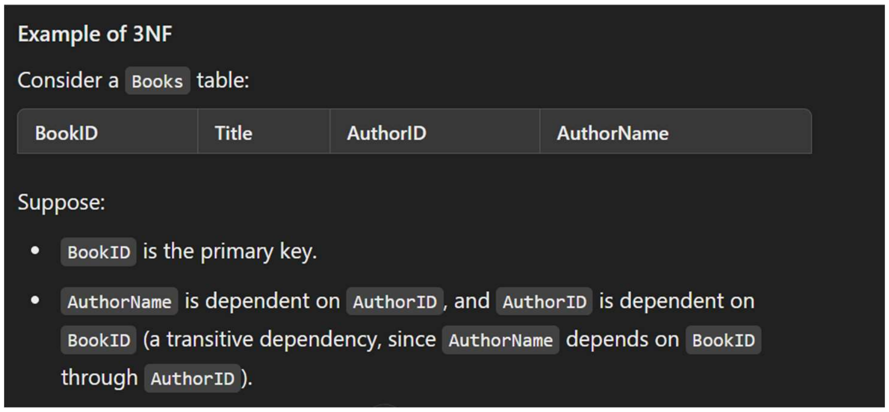
StudentID	CourseID

Now, all non-key attributes in each table fully depend on the entire primary key.

3. Third Normal Form (3NF)

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

- It is in **2NF**.
- There are **no transitive dependencies**; that is, a non-key attribute should not depend on another non-key attribute.



Example of 3NF

Consider a `Books` table:

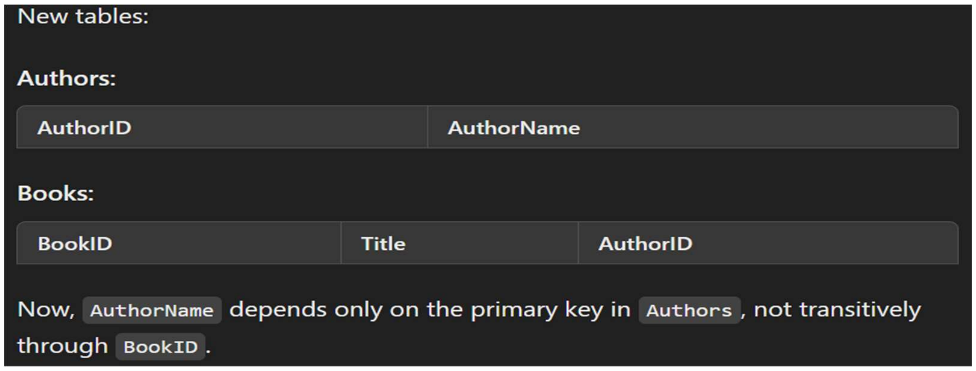
BookID	Title	AuthorID	AuthorName

Suppose:

- `BookID` is the primary key.
- `AuthorName` is dependent on `AuthorID`, and `AuthorID` is dependent on `BookID` (a transitive dependency, since `AuthorName` depends on `BookID` through `AuthorID`).

To achieve 3NF:

1. **Decompose** the table into:
    - Authors: Stores (AuthorID, AuthorName).
    - Books: Stores (BookID, Title, AuthorID).



New tables:

Authors:

AuthorID	AuthorName

Books:

BookID	Title	AuthorID

Now, `AuthorName` depends only on the primary key in `Authors`, not transitively through `BookID`.

BCNF (Boyce-Codd Normal Form): Simplified Explanation

A table is in **BCNF** if:

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

Key Terms Recap

- **Determinant**: An attribute (or set of attributes) that determines another attribute. For example, if CourseID → Department, then CourseID is a determinant of Department.
- **Candidate Key**: A set of attributes that uniquely identifies a row in a table. Every table should have at least one candidate key (usually, this is the primary key, but there can be others).

Why is BCNF Needed?

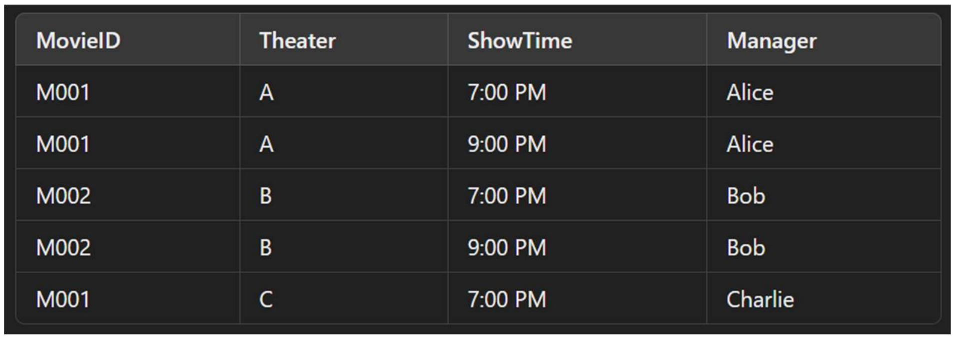
BCNF is needed when:

1. There are **dependencies** between non-primary attributes, and
2. A determinant (an attribute that determines another) is **not** a candidate key.

This situation can lead to **data anomalies** and redundancy even when the table is in 3NF.

Example: Movie Database

Consider a table named MovieShowings with the following attributes:



MovieID	Theater	ShowTime	Manager
M001	A	7:00 PM	Alice
M001	A	9:00 PM	Alice
M002	B	7:00 PM	Bob
M002	B	9:00 PM	Bob
M001	C	7:00 PM	Charlie

Key Points About This Table:

1. **Primary Key**: (MovieID, Theater, ShowTime) uniquely identifies each row.
2. **Functional Dependencies**:
   - MovieID, Theater → Manager: The manager is determined by the theater showing the movie.
   - Theater → Manager: Each theater has one specific manager.

Identifying the BCNF Violation

In this table, we can see the following:

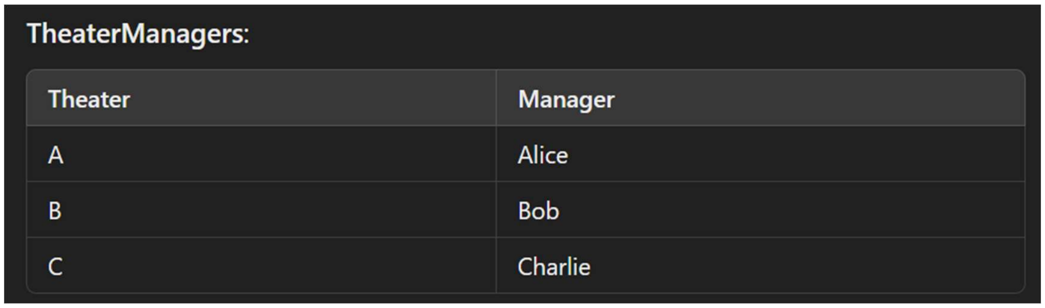
- **Determinant**: Theater determines Manager, but Theater is **not** a candidate key on its own (it's part of the primary key).

Decomposing to Achieve BCNF

To convert this table into BCNF, we need to break it down into two tables to eliminate the violation:

Step 1: Create the TheaterManagers Table

This table will store which manager is responsible for each theater.

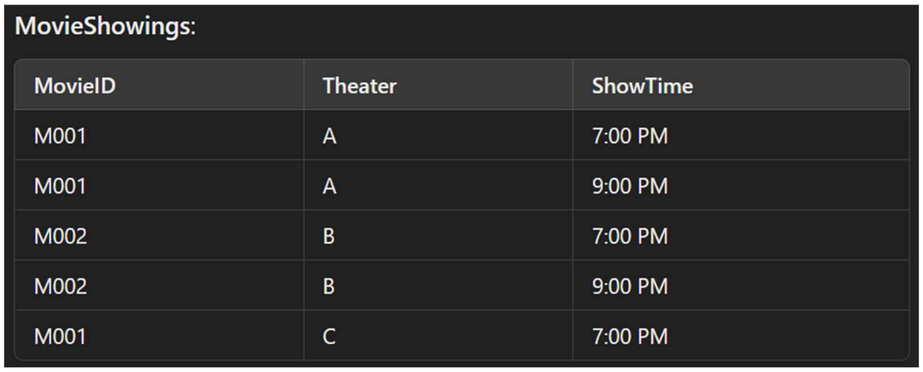


TheaterManagers:

Theater	Manager
A	Alice
B	Bob
C	Charlie

Step 2: Create the Movie Showings Table

This table will now focus on the showings without the manager information.



MovieShowings:

MovieID	Theater	ShowTime
M001	A	7:00 PM
M001	A	9:00 PM
M002	B	7:00 PM
M002	B	9:00 PM
M001	C	7:00 PM

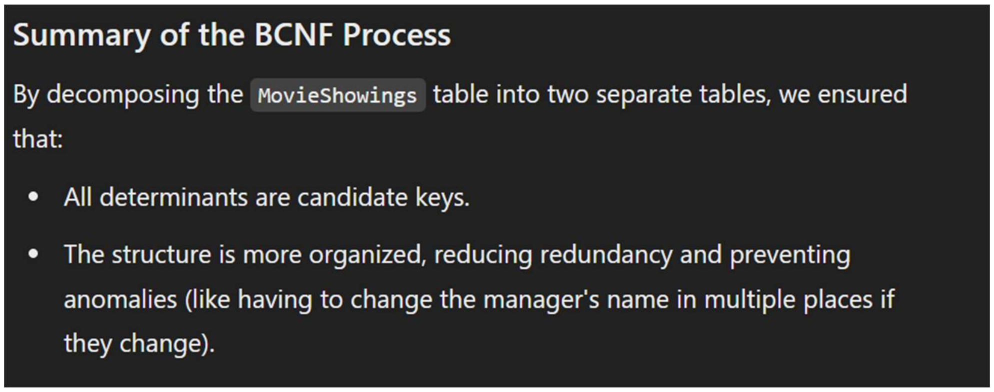
New Functional Dependencies

1. **In the TheaterManagers table**:
   - Theater → Manager: The manager is determined by the theater.
2. **In the MovieShowings table**:
   - The primary key (MovieID, Theater, ShowTime) uniquely identifies the showtime for each movie at a theater.

Conclusion

Now both tables are in BCNF:

- In **TheaterManagers**, the determinant (Theater) is a candidate key (it uniquely identifies rows).
- In **MovieShowings**, the primary key determines all other attributes without any partial or transitive dependencies.



Summary of the BCNF Process

By decomposing the `MovieShowings` table into two separate tables, we ensured that:

- All determinants are candidate keys.
- The structure is more organized, reducing redundancy and preventing anomalies (like having to change the manager's name in multiple places if they change).

Fourth Normal Form (4NF)

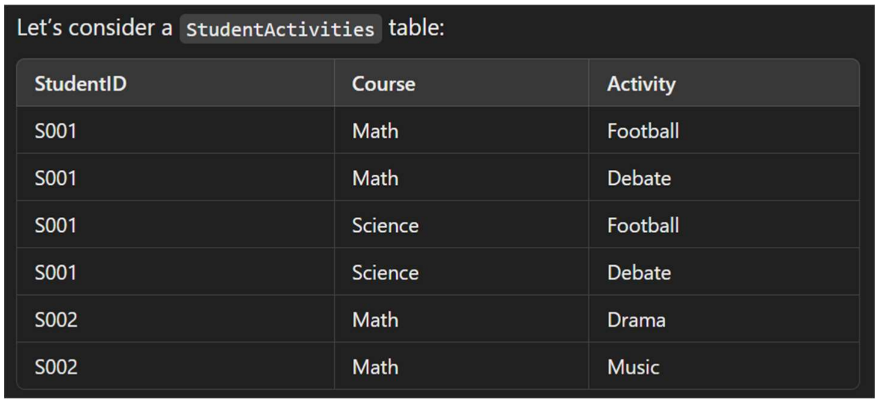
Fourth Normal Form (4NF) is a level of database normalization that aims to eliminate multi-valued dependencies in a relational database. A table is in 4NF if:

1. It is in **Boyce-Codd Normal Form (BCNF)**.
2. It has no **multi-valued dependencies**.

Key Concepts

- **Multi-Valued Dependency**: A multi-valued dependency occurs when one attribute in a table uniquely determines another attribute, but not in relation to the entire primary key. In simpler terms, if you can find a set of values for one column that corresponds to multiple values in another column, you have a multi-valued dependency.
- **Example of a Multi-Valued Dependency**: Consider a table where a student can enroll in multiple courses and also participate in multiple extracurricular activities.

Example of a Table with Multi-Valued Dependencies



Let's consider a `StudentActivities` table:

StudentID	Course	Activity
S001	Math	Football
S001	Math	Debate
S001	Science	Football
S001	Science	Debate
S002	Math	Drama
S002	Math	Music

Analyzing the Table

- **Primary Key**: (StudentID, Course, Activity).
- **Multi-Valued Dependency**:
  - The StudentID determines Course and also determines Activity.

    - However, the Course attribute is independent of the Activity attribute (i.e., a student can take multiple courses and participate in multiple activities).

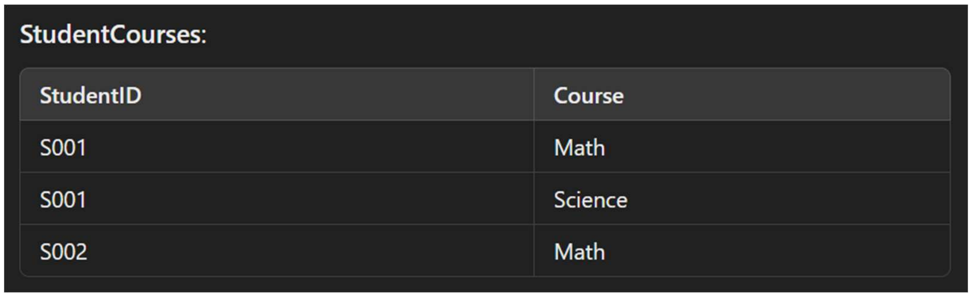
This structure creates redundancy because the same student-course combination is repeated for each activity.

Decomposing the Table to Achieve 4NF

To eliminate the multi-valued dependencies, we can decompose the StudentActivities table into two separate tables:

Step 1: Create the StudentCourses Table

This table will store which courses each student is enrolled in.

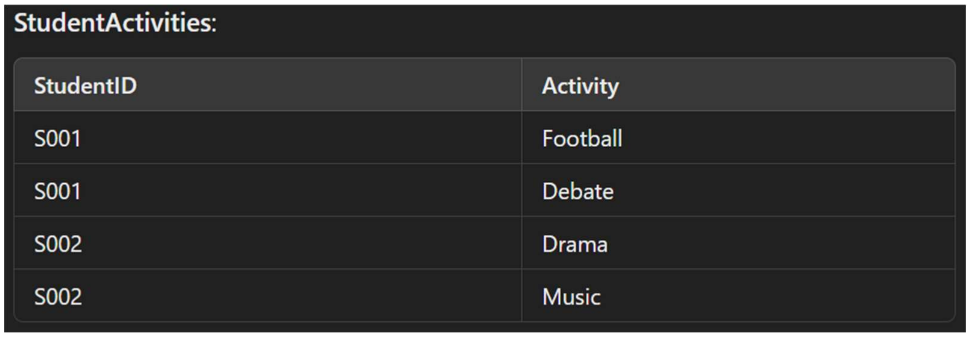


StudentCourses:

StudentID	Course
S001	Math
S001	Science
S002	Math

Step 2: Create the StudentActivities Table

This table will store the extracurricular activities for each student.



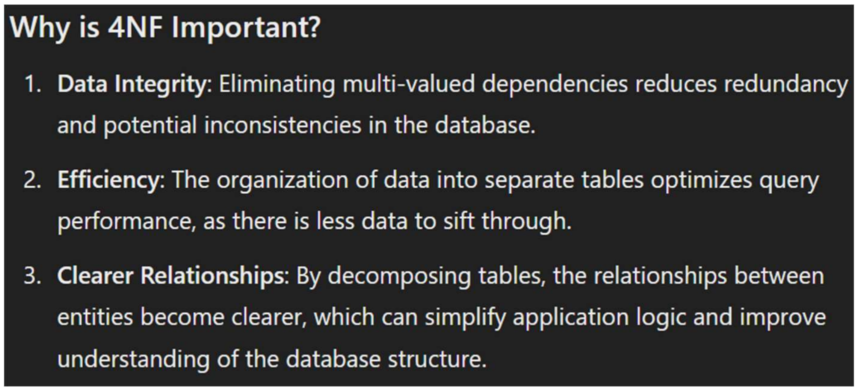
StudentActivities:

StudentID	Activity
S001	Football
S001	Debate
S002	Drama
S002	Music

Achieving 4NF

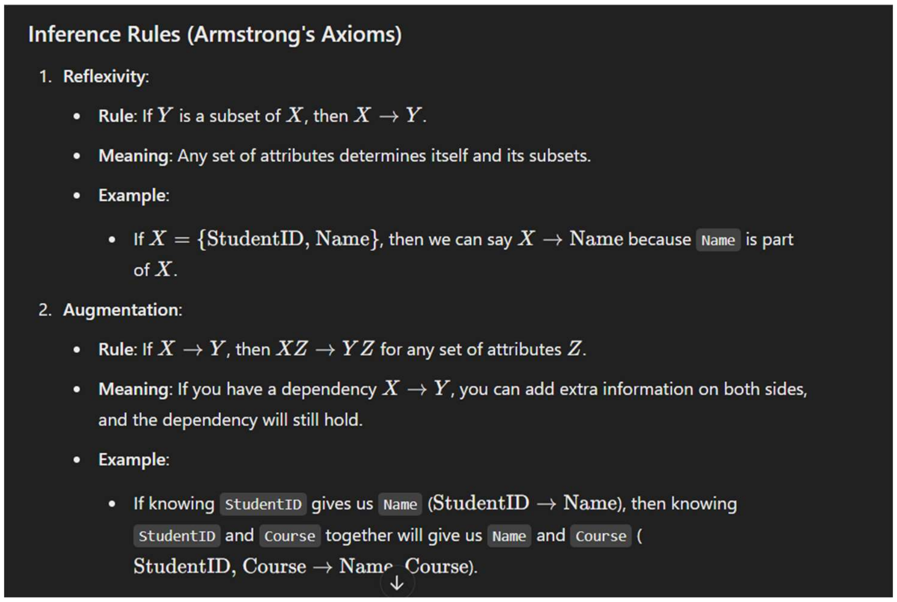
Now, both tables are in 4NF:

- **In StudentCourses**, the primary key (StudentID, Course) uniquely identifies rows without any multi-valued dependency.
- **In StudentActivities**, the primary key (StudentID, Activity) also uniquely identifies rows without any multi-valued dependency.

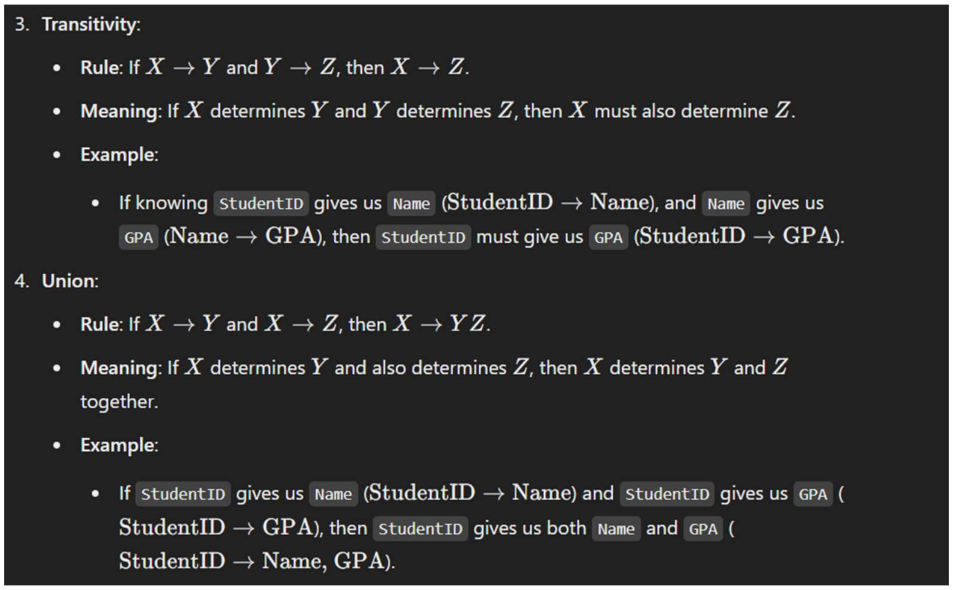

Why is 4NF Important?

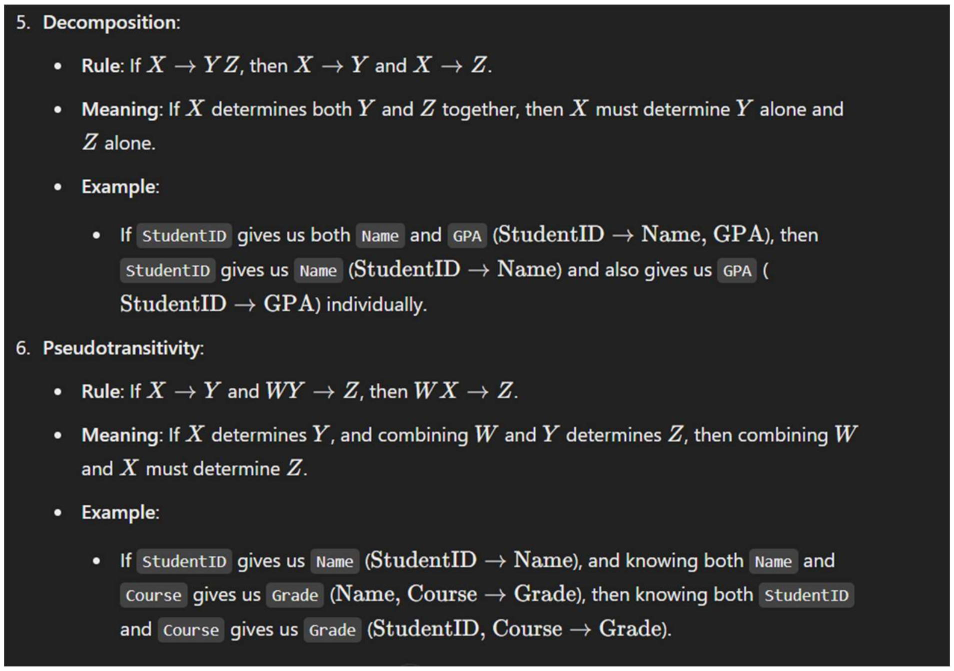
1. **Data Integrity**: Eliminating multi-valued dependencies reduces redundancy and potential inconsistencies in the database.
2. **Efficiency**: The organization of data into separate tables optimizes query performance, as there is less data to sift through.
3. **Clearer Relationships**: By decomposing tables, the relationships between entities become clearer, which can simplify application logic and improve understanding of the database structure.

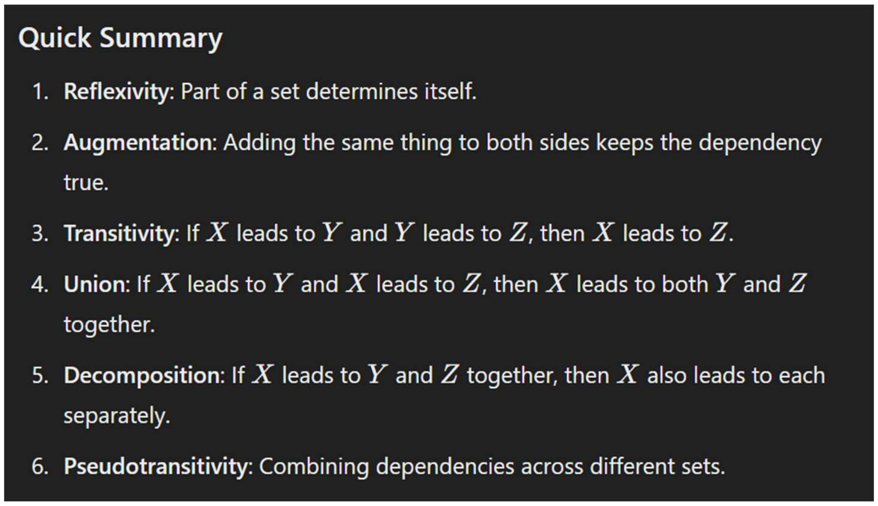
INFERENCE RULES


Inference Rules (Armstrong's Axioms)

1. **Reflexivity**:
   - **Rule**: If Y is a subset of X, then $X \to Y$.
   - **Meaning**: Any set of attributes determines itself and its subsets.
   - **Example**:
     - If $X = \{\text{StudentID, Name}\}$, then we can say $X \to \text{Name}$ because `Name` is part of X.
2. **Augmentation**:
   - **Rule**: If $X \to Y$, then $XZ \to YZ$ for any set of attributes Z.
   - **Meaning**: If you have a dependency $X \to Y$, you can add extra information on both sides, and the dependency will still hold.
   - **Example**:
     - If knowing `StudentID` gives us `Name` ($\text{StudentID} \to \text{Name}$), then knowing `StudentID` and `Course` together will give us `Name` and `Course` ($\text{StudentID, Course} \to \text{Name, Course}$).


3. **Transitivity**:
   - **Rule**: If $X \to Y$ and $Y \to Z$, then $X \to Z$.
   - **Meaning**: If X determines Y and Y determines Z, then X must also determine Z.
   - **Example**:
     - If knowing `StudentID` gives us `Name` ($\text{StudentID} \to \text{Name}$), and `Name` gives us `GPA` ($\text{Name} \to \text{GPA}$), then `StudentID` must give us `GPA` ($\text{StudentID} \to \text{GPA}$).
4. **Union**:
   - **Rule**: If $X \to Y$ and $X \to Z$, then $X \to YZ$.
   - **Meaning**: If X determines Y and also determines Z, then X determines Y and Z together.
   - **Example**:
     - If `StudentID` gives us `Name` ($\text{StudentID} \to \text{Name}$) and `StudentID` gives us `GPA` ($\text{StudentID} \to \text{GPA}$), then `StudentID` gives us both `Name` and `GPA` ($\text{StudentID} \to \text{Name, GPA}$).


5. **Decomposition**:
   - **Rule**: If $X \to YZ$, then $X \to Y$ and $X \to Z$.
   - **Meaning**: If X determines both Y and Z together, then X must determine Y alone and Z alone.
   - **Example**:
     - If `StudentID` gives us both `Name` and `GPA` ($\text{StudentID} \to \text{Name, GPA}$), then `StudentID` gives us `Name` ($\text{StudentID} \to \text{Name}$) and also gives us `GPA` ($\text{StudentID} \to \text{GPA}$) individually.
6. **Pseudotransitivity**:
   - **Rule**: If $X \to Y$ and $WY \to Z$, then $WX \to Z$.
   - **Meaning**: If X determines Y, and combining W and Y determines Z, then combining W and X must determine Z.
   - **Example**:
     - If `StudentID` gives us `Name` ($\text{StudentID} \to \text{Name}$), and knowing both `Name` and `Course` gives us `Grade` ($\text{Name, Course} \to \text{Grade}$), then knowing both `StudentID` and `Course` gives us `Grade` ($\text{StudentID, Course} \to \text{Grade}$).


Quick Summary

1. **Reflexivity**: Part of a set determines itself.
2. **Augmentation**: Adding the same thing to both sides keeps the dependency true.
3. **Transitivity**: If X leads to Y and Y leads to Z, then X leads to Z.
4. **Union**: If X leads to Y and X leads to Z, then X leads to both Y and Z together.
5. **Decomposition**: If X leads to Y and Z together, then X also leads to each separately.
6. **Pseudotransitivity**: Combining dependencies across different sets.

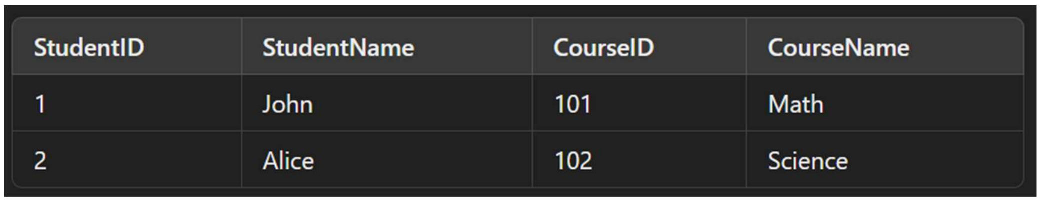
Different types of anamolies with example

In database design, anomalies are issues that can arise in a poorly normalized database. They can lead to redundancy, inconsistency, and inefficiency when inserting, updating, or deleting data. There are three main types of anomalies:

1. Insertion Anomaly

An insertion anomaly occurs when we can't add data to the database because certain other data is missing. This is common in unnormalized or partially normalized tables.

Example:
Consider a table Student_Course that includes StudentID, StudentName, CourseID, and CourseName.

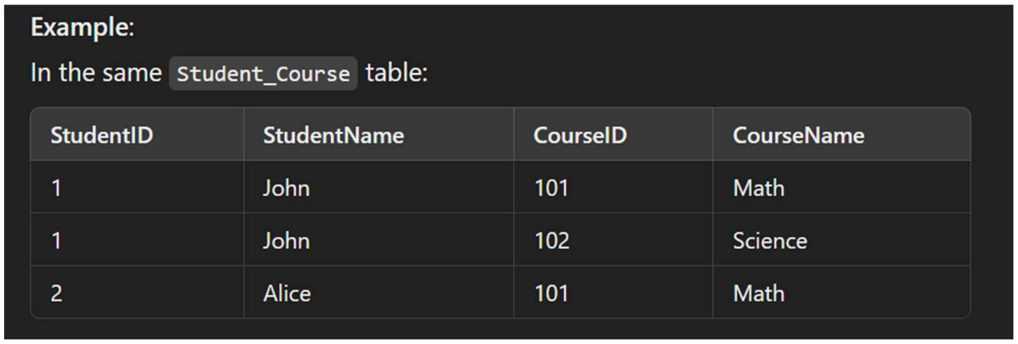


StudentID	StudentName	CourseID	CourseName
1	John	101	Math
2	Alice	102	Science

Problem: If we want to insert information for a new course, like CourseID = 103, CourseName = History, we cannot do so without having a student enrolled in that course. We are forced to enter a dummy student or leave StudentID and StudentName fields as NULL, which creates redundancy or incomplete data.

2. Update Anomaly

An update anomaly occurs when we have to update multiple rows of data due to redundancy, leading to data inconsistency if some rows are updated while others are missed.



Example:

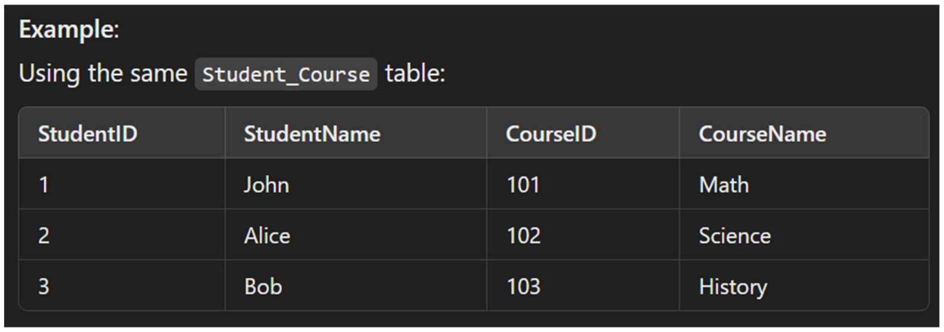
In the same `Student_Course` table:

StudentID	StudentName	CourseID	CourseName
1	John	101	Math
1	John	102	Science
2	Alice	101	Math

Problem: If John's name changes, say, to "John Smith," we have to update it in multiple rows (all rows where StudentID = 1). If we miss updating any row, we end up with inconsistent data (some rows have "John" and others "John Smith").

3. Deletion Anomaly

A deletion anomaly occurs when deleting data inadvertently results in the loss of other valuable information.



Example:

Using the same `Student_Course` table:

StudentID	StudentName	CourseID	CourseName
1	John	101	Math
2	Alice	102	Science
3	Bob	103	History

Problem: If Bob drops out and we delete his row (since there are no other students enrolled in CourseID = 103), we lose the information about the course History. Now, the database no longer has any record of the course History, which might still be useful information.

Solution: Normalization

To reduce these anomalies, we apply **normalization**—organizing the database into multiple tables so each piece of data is stored only once and redundancy is minimized. For example, splitting the Student_Course table into two separate tables, Student and Course, with a Student_Course_Enrollments table, can help reduce anomalies:

- **Student Table**: stores unique student information.
- **Course Table**: stores unique course information.
- **Student_Course_Enrollments Table**: links students with the courses they are enrolled in, helping prevent data redundancy and ensuring consistency.