UNIT -3
Introduction to Database Normalization

Normalization is an important process in database design that **helps improve the database's efficiency, consistency, and accuracy**. It makes it easier to manage and maintain the data and ensures that the database is adaptable to changing business needs.

- Database normalization is the **process of organizing the attributes of the database to reduce or eliminate data redundancy** (having the same data but at different places).
- Data redundancy unnecessarily increases the size of the database as the same data is repeated in many places. Inconsistency problems also arise during insert, delete, and update operations.

Features of Database Normalization

- **Elimination of Data Redundancy:** One of the main features of normalization is to eliminate the data redundancy that can occur in a database. Data redundancy refers to the repetition of data in different parts of the database. Normalization helps in reducing or eliminating this redundancy, which can improve the efficiency and consistency of the database.
- **Ensuring Data Consistency:** Normalization helps in ensuring that the data in the database is consistent and accurate. By eliminating redundancy, normalization helps in preventing inconsistencies and contradictions that can arise due to different versions of the same data.
- **Simplification of Data Management:** Normalization simplifies the process of managing data in a database. By breaking down a complex data structure into simpler tables, normalization makes it easier to manage the data, update it, and retrieve it.
- **Improved Database Design:** Normalization helps in improving the overall design of the database. By organizing the data in a structured and systematic way, normalization makes it easier to design and maintain the database. It also makes the database more flexible and adaptable to changing business needs.
- **Avoiding Update Anomalies:** Normalization helps in avoiding update anomalies, which can occur when updating a single record in a table affects multiple records in other tables. Normalization ensures that each table contains only one type of data and that the relationships between the tables are clearly defined, which helps in avoiding such anomalies.
- **Standardization:** Normalization helps in standardizing the data in the database. By organizing the data into tables and defining relationships

between them, normalization helps in ensuring that the data is stored in a consistent and uniform manner.

Functional Dependency in DBMS

- A **functional dependency occurs when the value of one attribute** (or a set of attributes) uniquely **determines the value of another attribute**. This relationship is denoted as:

  X → Y

  Here, **X** is the **determinant**,
  and **Y** is the **dependent** attribute.

  This means that for each unique value of X, there is precisely one corresponding value of Y.

  Example:
  Consider a table named Students with the following attributes:
  - StudentID
  - StudentName
  - StudentAge

  If each student has a unique StudentID, and this ID determines the student's name, we can **express this functional dependency** as:
  StudentID → StudentName

  This indicates that knowing the StudentID allows us to determine the

  StudentName.

StudentID	StudentName	StudentAge
101	Rahul	23
102	Ankit	22
103	Aditya	22
104	Sahil	24

Functional Dependency

Types of Functional Dependency in DBMS

The following are some important types of FDs in DBMS:

Trivial Functional Dependency

A **Functional Dependency** is called **trivial** when the **dependent attributes (Y)** are **already a subset of** the **determinant (X)**.

$$X \rightarrow Y \text{ is trivial if } Y \subseteq X$$

Non-trivial Functional Dependency

A functional dependency is called non-trivial when **the right-hand side (Y) is not a subset of the left-hand side (X).**

$$X \rightarrow Y \text{ is non-trivial if } Y \not\subseteq X$$

Attribute Closure

The **attribute closure** of a set of attributes **X**, denoted as $\mathbf{X^+}$ **(X plus)**, is:

The **set of all attributes** that can be functionally determined from **X** using a given set of **Functional Dependencies (FDs)**.

In simple terms:

- Start with **X** (a set of attributes).
- Use the given **FDs** to find everything that depends on X.
- The final set of all such attributes = $\mathbf{X^+}$ **(closure of X)**.

Why Do We Find Attribute Closure?

We use **attribute closure** to:

- **To check if a functional dependency holds:**

- We can determine whether $\mathbf{X \rightarrow Y}$ is valid by checking if $\mathbf{Y \subseteq X^+}$.

• **To find candidate keys:**

- By computing the closure of different attribute sets, we can identify which sets of attributes functionally determine all attributes in the relation (i.e., **candidate keys**).

• **To test for normalization:**

- It helps in analyzing dependencies when decomposing relations into higher normal forms (like 2NF, 3NF, BCNF).

• **To minimize functional dependencies:**

- Closure helps **determine redundant attributes or dependencies**.

Steps to Find Attribute Closure (X^+)

Given:

- A relation **R(A, B, C, D, …)**
- A set of FDs: **F**

To find $\mathbf{X^+}$ (closure of X):

1. **Start:**
   $X^+ = X$
2. **Apply FDs repeatedly:**
   For each FD ($A \rightarrow B$),
   if $A \subseteq X^+$, then add B to X^+.
3. **Repeat until** no new attributes can be added.

Armstrong's Axioms in Functional Dependency in DBMS

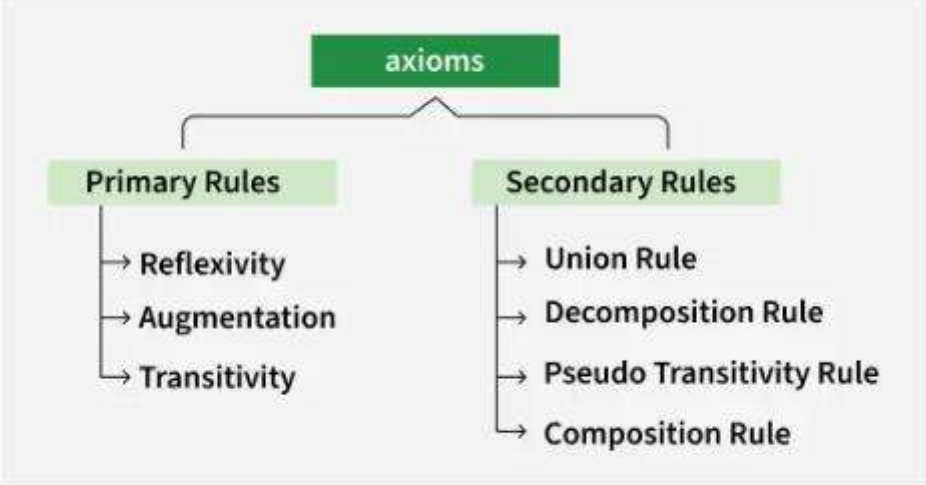
Armstrong's Axioms refer to a set of inference rules, introduced by William W. Armstrong, that are **used to test the logical implication of functional dependencies.**

Given a set of functional dependencies F, the closure of F (denoted as F+) is the set of all functional dependencies logically implied by F.

Armstrong's Axioms, when applied repeatedly, help generate the closure of functional dependencies.

These axioms are fundamental in determining functional dependencies in databases and are used to derive conclusions about the relationships between attributes.

Axioms



Axioms

- **Axiom of Reflexivity:** If A is a set of attributes and B is a subset of A, then A holds B. **If B⊆A then A→B**. This property is **trivial property**.

- **Axiom of Augmentation:** If **A→B** holds and Y is the attribute set, then **AY→BY** also holds. That is adding attributes to dependencies, does not change the basic dependencies. If **A→B**, then **AC→BC** for any C.

- **Axiom of Transitivity:** Same as the transitive rule in algebra, if **A→B** holds and **B→C** holds, then **A→C** also holds. **A→B** is called A functionally which determines B. If **X→Y** and **Y→Z**, then **X→Z.**

Example:
Let's assume the following functional dependencies:
{A} → {B}
{B} → {C}
{A, C} → {D}

1. **Reflexivity**: Since any set of attributes determines its subset, we can immediately infer the following:

- **{A}** → **{A}** (A set always determines itself).
- **{B}** → **{B}**.
- **{A, C}** → **{A}**.
- {A, C} → {C}

2. **Augmentation**: If we know that **{A}** → **{B}**, we can add the same attribute (or set of attributes) to both sides:
- From **{A}** → **{B}**, we can augment both sides with **{C}**: **{A, C}** → **{B, C}**.
- From **{B}** → **{C}**, we can augment both sides with **{A}**: **{A, B}** → **{A,C}**.

3. **Transitivity**: If we know **{A}** → **{B}** and **{B}** → **{C}**, we can infer that:
- **{A}** → **{C}**

- Although Armstrong's axioms are sound and complete, there are additional rules for functional dependencies that are derived from them. These rules are introduced to simplify operations and make the process easier.

Secondary Rules

These rules can be derived from the above axioms.

- **Union:** If **A**→**B** holds and **A**→**C** holds, then **A**→**BC** holds.
  If **X**→**Y** and **X**→**Z** then **X**→**YZ**.

- **Composition:** If **A**→**B** and **X**→**Y** hold, then **AX**→**BY** holds.

- **Decomposition:** If **A**→**BC** holds then **A**→**B** and **A**→**C** hold.
  If **X**→**YZ** then **X**→**Y** and **X**→**Z**.

- **Pseudo Transitivity:** If **A**→**B** holds and **BC**→**D** holds,
  then **AC**→**D** holds. If **X**→**Y** and **YZ**→**W** then **XZ**→**W**.

Example:
Let’s assume we have the following functional dependencies in a relation schema:
{A} → ***{B}***
{A} → ***{C}***
{X} → ***{Y}***
{Y, Z} → ***{W}***
Now, let's apply the **Secondary Rules** to derive new functional dependencies.
1. **Union Rule**: If **A** → **B** and **A** → **C**, then by the **Union Rule**, we can infer:

- **A → BC** This means if **A** determines both **B** and **C**, it also determines their combination, **BC**.

2. **Composition Rule**: If **A → B** and **X → Y** hold, then by the **Composition Rule**, we can infer:

- **AX → BY**

3. **Decomposition Rule**: If **A → BC** holds, then by the **Decomposition Rule**, we can infer:

- **A → B** and **A → C**

4. **Pseudo Transitivity Rule**: If **A → B** and **BC → D** hold, then by the **Pseudo Transitivity Rule**, we can infer:

- **AC → D**

Benefits of Functional Dependency in DBMS

Functional dependency in a database management system offers several advantages for businesses and organizations:

- **Prevents Duplicate Data:** Functional dependency helps avoid storing the same data repeatedly in the database, reducing redundancy and saving storage space.
- **Improves Data Quality and Accuracy:** By organizing data efficiently and minimizing duplication, functional dependency ensures the data is reliable, consistent, and of high quality.
- **Reduces Errors:** Keeping data organized and concise lowers the chances of errors in records or datasets, making it easier to manage and update information.
- **Saves Time and Costs:** Properly organized data allows for quicker and easier access, improving productivity and reducing the time and cost of managing information.
- **Defines Rules and Behaviours:** Functional dependency allows setting rules and constraints that control how data is stored, accessed, and maintained, ensuring better data management.
- **Helps Identify Poor Database Design:** It highlights issues like scattered or missing data across tables, helping identify and fix design flaws to maintain consistency and integrity.

Normal Forms in DBMS

- Normal forms are a set of progressive rules (or design checkpoints) for relational schemas that reduce redundancy and prevent data anomalies.

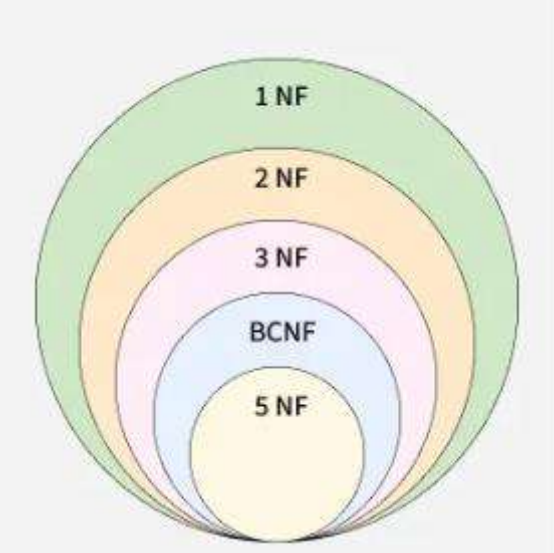
  Each normal form - 1NF, 2NF, 3NF, BCNF, 4NF, 5NF - is stricter than the previous one: meeting a higher normal form implies the lower ones are satisfied.

  Think of them as layers of cleanliness for your tables: the deeper you go, the fewer redundancy and integrity problems you'll have.

  Benefits of using Normal Forms:
  - **Reduce duplicate data** and wasted storage.
  - Prevent insert, update, and delete anomalies.
  - Improve **data consistency** and **integrity**.
  - Make the **schema easier to maintain** and **evolve**.

  The Diagram below shows the hierarchy of database normal forms.
  Each **level builds upon the previous one to ensure a cleaner and more efficient database design.**

  Normal Form



Let's break down the various normal forms step-by-step to understand the conditions that need to be satisfied at each level:

1. **First Normal Form (1NF)**: **Eliminating Duplicate Records**

A table is in 1NF if it satisfies the following conditions:

- All columns contain atomic values (i.e., indivisible values).
- Each **row is unique** (i.e., no duplicate rows).
- Each **column has a unique name**.
- The order in which data is stored does not matter.

Example of 1NF Violation: If a table has a column "**Phone Numbers**" that **stores multiple phone numbers in a single cell, it violates 1NF**. To bring it into 1NF, you need to separate phone numbers into individual rows.

2. **Second Normal Form (2NF)**: **Eliminating Partial Dependency**

A relation is in 2NF if it satisfies the conditions of 1NF and additionally.

No partial dependency exists, meaning every non-prime attribute (non-key attribute) must depend on the entire primary key, not just a part of it.

Example: For a composite key (StudentID, CourseID), if the "StudentName" depends only on "StudentID" and not on the entire key, it violates 2NF.

To normalize, move StudentName into a separate table where it depends only on "StudentID".

3. **Third Normal Form (3NF)**: **Eliminating Transitive Dependency**

A relation is in 3NF if it satisfies 2NF and additionally, there are no transitive dependencies. In simpler terms, non-prime attributes should not depend on other non-prime attributes.

Example: Consider a table with (StudentID, CourseID, Instructor). If Instructor depends on "CourseID", and "CourseID" depends on "StudentID", then Instructor indirectly depends on "StudentID", which violates 3NF.

To resolve this, place Instructor in a separate table linked by "CourseID".

4. **Boyce-Codd Normal Form (BCNF)**: The Strongest Form of 3NF
BCNF is a stricter version of 3NF where for every non-trivial functional dependency (X → Y), X must be a superkey (a unique identifier for a record in the table).

Example: If a table has a dependency (StudentID, CourseID) → Instructor, but neither "StudentID" nor "CourseID" is a superkey, then it violates BCNF. To bring it into BCNF, decompose the table so that each determinant is a candidate key.

Normal Forms in DBMS

Normal Forms	Description of Normal Forms
First Normal Form (1NF)	A relation is in first normal form if **every attribute** in that relation is **single-valued** attribute.
Second Normal Form (2NF)	A relation that is in First Normal Form and every **non-primary-key attribute is fully functionally dependent on the primary key**, then the relation is in Second Normal Form (2NF).
Third Normal Form (3NF)	A relation is in the third normal form, if there is **no transitive dependency** for non-prime attributes as well as it is in the second normal form. A relation is in **3NF if at least one of the following conditions holds in every non-trivial function dependency** X –> Y. • **X is a super key.** • **Y is a prime attribute** (each element of Y is part of some candidate key).
Boyce-Codd Normal Form (BCNF)	For BCNF the relation should satisfy the below conditions • The relation should be in the 3rd Normal Form. • **X should be a super-key** for every functional dependency (FD) X–>Y in a given relation.