**Hexaware Technical Training Program**

**Day 5 - March 17 , 2025**

**DBMS**

**Normalization:**

**Normalization** is the process of **organizing data** in a database to eliminate **redundancy** and ensure **data integrity** by structuring it efficiently.

**Why Normalize?**

* Reduces **data redundancy**
* Avoids **inconsistencies**
* Improves **data integrity**
* Ensures **efficient queries**

**Normal Forms (NF)**

Normalization is divided into **normal forms (NF)**, each with **specific rules** to improve the database structure.

**1st Normal Form (1NF) – Eliminate Duplicate Columns**

**Rule:**

* Each **column** must have **atomic (indivisible) values**.
* Each **row** must be **unique**.

🔹 **Example (Before 1NF – Repeating Groups):**

| **Student\_ID** | **Name** | **Subjects** |
| --- | --- | --- |
| 101 | Alice | Math, Science |
| 102 | Bob | English, Math |

🔹 **After 1NF (Atomic Values & Unique Rows):**

| **Student\_ID** | **Name** | **Subject** |
| --- | --- | --- |
| 101 | Alice | Math |
| 101 | Alice | Science |
| 102 | Bob | English |
| 102 | Bob | Math |

**2nd Normal Form (2NF) – Eliminate Partial Dependency**

**Rule:**

* Database must be in **1NF**.
* **No partial dependency** (a non-key attribute should depend on the entire primary key, not just part of it).

🔹 **Example (Before 2NF – Partial Dependency):**

| **Order\_ID** | **Product\_ID** | **Product\_Name** | **Customer\_Name** |
| --- | --- | --- | --- |
| 1 | P101 | Laptop | Alice |
| 2 | P102 | Phone | Bob |

🔹 **Issue:**

* Product\_Name depends only on Product\_ID, not Order\_ID.

🔹 **After 2NF (Breaking into Two Tables):**  
👉 **Orders Table**

| **Order\_ID** | **Product\_ID** | **Customer\_Name** |
| --- | --- | --- |
| 1 | P101 | Alice |
| 2 | P102 | Bob |

👉 **Products Table**

| **Product\_ID** | **Product\_Name** |
| --- | --- |
| P101 | Laptop |
| P102 | Phone |

**3rd Normal Form (3NF) – Remove Transitive Dependency**

**Rule:**

* Database must be in **2NF**.
* **No transitive dependency** (a non-key attribute must not depend on another non-key attribute).

🔹 **Example (Before 3NF – Transitive Dependency):**

| **Student\_ID** | **Name** | **Department** | **Department\_Location** |
| --- | --- | --- | --- |
| 101 | Alice | CSE | Block A |
| 102 | Bob | ECE | Block B |

🔹 **Issue:**

* Department\_Location depends on Department, not on Student\_ID.

🔹 **After 3NF (Breaking into Two Tables):**  
👉 **Student Table**

| **Student\_ID** | **Name** | **Department** |
| --- | --- | --- |
| 101 | Alice | CSE |
| 102 | Bob | ECE |

👉 **Department Table**

| **Department** | **Department\_Location** |
| --- | --- |
| CSE | Block A |
| ECE | Block B |

**Boyce-Codd Normal Form (BCNF)**

BCNF is a **stronger version of 3NF** that eliminates anomalies by ensuring every **determinant** is a **candidate key**.

**BCNF Rule**

* The database **must be in 3NF**.
* Every **determinant** (attribute that determines another attribute) must be a **candidate key**.

💡 **Key Point:** If a table has **more than one candidate key**, BCNF ensures that no partial dependencies exist.

**Example of BCNF**

🔹 **Before BCNF (Violating BCNF Rule):**

| **Student\_ID** | **Course** | **Instructor** | **Instructor\_Office** |
| --- | --- | --- | --- |
| 101 | DBMS | Prof. A | Room 101 |
| 102 | OOPS | Prof. B | Room 102 |
| 103 | DBMS | Prof. A | Room 101 |
| 104 | OOPS | Prof. B | Room 102 |

🔹 **Issue:**

* **Functional dependencies:**
  + **(Student\_ID, Course) → Instructor**
  + **Instructor → Instructor\_Office** (Instructor determines office)
* Here, **Instructor is not a candidate key**, violating BCNF.

🔹 **After BCNF (Splitting into Two Tables):** 👉 **Student-Course Table**

| **Student\_ID** | **Course** | **Instructor** |
| --- | --- | --- |
| 101 | DBMS | Prof. A |
| 102 | OOPS | Prof. B |
| 103 | DBMS | Prof. A |
| 104 | OOPS | Prof. B |

👉 **Instructor-Office Table**

| **Instructor** | **Instructor\_Office** |
| --- | --- |
| Prof. A | Room 101 |
| Prof. B | Room 102 |

💡 Now, **every determinant is a candidate key**, and BCNF is satisfied!

**4th Normal Form (4NF) – Eliminating Multi-Valued Dependencies**

**Rule:**

* The table **must be in BCNF**.
* **Multi-valued dependencies** (where one attribute has multiple independent values for another) must be eliminated.

**Example of 4NF Violation:**

🔹 **Before 4NF (Multi-Valued Dependency Present):**

| **Student\_ID** | **Course** | **Hobby** |
| --- | --- | --- |
| 101 | DBMS | Music |
| 101 | OOPS | Music |
| 101 | DBMS | Chess |
| 101 | OOPS | Chess |

🔹 **Issue:**

* Student **101** has two independent **multi-valued attributes**: Course & Hobby.

🔹 **After 4NF (Separate Tables for Independent Values):** 👉 **Student-Course Table**

| **Student\_ID** | **Course** |
| --- | --- |
| 101 | DBMS |
| 101 | OOPS |

👉 **Student-Hobby Table**

| **Student\_ID** | **Hobby** |
| --- | --- |
| 101 | Music |
| 101 | Chess |

💡 **Now, multi-valued dependencies are eliminated!**

**5th Normal Form (5NF) – Eliminating Join Dependencies**

**Rule:**

* The table **must be in 4NF**.
* It must not contain **join dependencies**, which means data should not be **split into multiple tables unnecessarily**.

**Example of 5NF Violation:**

🔹 **Before 5NF (Join Dependency Present):**

| **Project\_ID** | **Employee** | **Skill** |
| --- | --- | --- |
| P1 | E1 | Java |
| P1 | E2 | Python |
| P2 | E1 | C++ |
| P2 | E3 | SQL |

🔹 **Issue:**

* The table can be **divided into multiple related tables**, but doing so would cause **loss of information when joined back**.

🔹 **After 5NF (No Join Dependency):** 👉 **Project-Employee Table**

| **Project\_ID** | **Employee** |
| --- | --- |
| P1 | E1 |
| P1 | E2 |
| P2 | E1 |
| P2 | E3 |

👉 **Employee-Skill Table**

| **Employee** | **Skill** |
| --- | --- |
| E1 | Java |
| E1 | C++ |
| E2 | Python |
| E3 | SQL |

💡 **Now, data can be joined without redundancy or loss of information!**

**DDL (Data Definition Language) :**

**DDL (Data Definition Language)** is used to define and manage the structure of a database, including creating, altering, and deleting database objects such as tables, schemas, and indexes.

**🚀 DDL Commands:**

1. **CREATE** – Creates a new table, database, or other database object.
2. **ALTER** – Modifies an existing database object.
3. **DROP** – Deletes an entire database object.
4. **TRUNCATE** – Removes all records from a table but keeps the structure.
5. **RENAME** – Renames a table or column.

**CREATE Statement**

Used to create a new table, database, or index.

**🔹 Example: Creating a Table**

CREATE TABLE Students (

Student\_ID INT PRIMARY KEY,

Name VARCHAR(50),

Age INT,

Course VARCHAR(50)

);

**🔹 Example: Creating a Database**

CREATE DATABASE SchoolDB;

**🔹 Example: Creating an Index**

CREATE INDEX idx\_student\_name ON Students (Name);

**ALTER Statement**

Used to modify an existing table, such as adding or deleting columns.

**🔹 Example: Adding a Column**

ALTER TABLE Students ADD Email VARCHAR(100);

**🔹 Example: Modifying a Column**

ALTER TABLE Students MODIFY Age SMALLINT;

**🔹 Example: Dropping a Column**

ALTER TABLE Students DROP COLUMN Course;

**DROP Statement**

Used to permanently delete a database object like a table or database.

**🔹 Example: Dropping a Table**

DROP TABLE Students;

**🔹 Example: Dropping a Database**

DROP DATABASE SchoolDB;

**🔹 Example: Dropping an Index**

DROP INDEX idx\_student\_name ON Students;

**TRUNCATE Statement**

Used to remove all records from a table **without affecting its structure**.

**🔹 Example: Truncating a Table**

TRUNCATE TABLE Students;

🔹 **Difference between DELETE and TRUNCATE:**

* DELETE removes rows **one by one** (can be rolled back).
* TRUNCATE removes all rows at once (faster but cannot be rolled back).

**RENAME Statement**

Used to rename a table or column.

**🔹 Example: Renaming a Table**

RENAME TABLE Students TO Student\_Info;

**🔹 Example: Renaming a Column**

ALTER TABLE Student\_Info RENAME COLUMN Name TO Full\_Name;

**🎯 Summary Table**

| **Command** | **Description** |
| --- | --- |
| CREATE | Creates a new table, database, or index |
| ALTER | Modifies an existing table (adds/drops columns) |
| DROP | Deletes a table or database permanently |
| TRUNCATE | Removes all records but retains the structure |
| RENAME | Changes the name of a table or column |

**SQL Constraints**

Constraints in SQL ensure the accuracy and integrity of data in a database by restricting the values that can be inserted, updated, or deleted.

**🔹 Types of Constraints in SQL**

1. **NOT NULL** – Ensures a column cannot have NULL values.
2. **UNIQUE** – Ensures all values in a column are unique.
3. **PRIMARY KEY** – A combination of NOT NULL and UNIQUE; uniquely identifies each row.
4. **FOREIGN KEY** – Ensures referential integrity between two tables.
5. **CHECK** – Ensures values in a column satisfy a specific condition.
6. **DEFAULT** – Provides a default value if no value is specified.
7. **INDEX** – Improves query performance (not a constraint but helps optimize queries).

**NOT NULL Constraint**

Ensures that a column cannot have NULL values.

🔹 **Example:**

CREATE TABLE Students (

Student\_ID INT NOT NULL,

Name VARCHAR(50) NOT NULL,

Age INT

);

**UNIQUE Constraint**

Ensures that all values in a column are unique.

🔹 **Example:**

CREATE TABLE Employees (

Employee\_ID INT UNIQUE,

Email VARCHAR(100) UNIQUE

);

🔹 **Adding UNIQUE to an Existing Table:**

ALTER TABLE Employees ADD CONSTRAINT unique\_email UNIQUE (Email);

**PRIMARY KEY Constraint**

A combination of NOT NULL and UNIQUE. Uniquely identifies each record in the table.

🔹 **Example:**

sql

CopyEdit

CREATE TABLE Students (

Student\_ID INT PRIMARY KEY,

Name VARCHAR(50),

Age INT

);

🔹 **Adding PRIMARY KEY to an Existing Table:**

sql

CopyEdit

ALTER TABLE Students ADD CONSTRAINT pk\_student PRIMARY KEY (Student\_ID);

**FOREIGN KEY Constraint**

Maintains referential integrity by linking a column in one table to the PRIMARY KEY of another table.

🔹 **Example:**

CREATE TABLE Courses (

Course\_ID INT PRIMARY KEY,

Course\_Name VARCHAR(50)

);

CREATE TABLE Students (

Student\_ID INT PRIMARY KEY,

Name VARCHAR(50),

Course\_ID INT,

FOREIGN KEY (Course\_ID) REFERENCES Courses(Course\_ID)

);

🔹 **Adding FOREIGN KEY to an Existing Table:**

ALTER TABLE Students ADD CONSTRAINT fk\_course FOREIGN KEY (Course\_ID) REFERENCES Courses(Course\_ID);

**CHECK Constraint**

Ensures that column values satisfy a specific condition.

🔹 **Example:**

sql

CopyEdit

CREATE TABLE Students (

Student\_ID INT PRIMARY KEY,

Name VARCHAR(50),

Age INT CHECK (Age >= 18) -- Ensures Age is at least 18

);

🔹 **Adding CHECK to an Existing Table:**

ALTER TABLE Students ADD CONSTRAINT check\_age CHECK (Age >= 18);

**DEFAULT Constraint**

Assigns a default value if no value is provided.

🔹 **Example:**

CREATE TABLE Employees (

Employee\_ID INT PRIMARY KEY,

Name VARCHAR(50),

Salary DECIMAL(10,2) DEFAULT 25000.00

);

🔹 **Adding DEFAULT to an Existing Table:**

ALTER TABLE Employees ALTER COLUMN Salary SET DEFAULT 25000.00;

**INDEX (Not a Constraint but Useful)**

Improves query performance by indexing a column.

🔹 **Example:**

CREATE INDEX idx\_employee\_name ON Employees (Name);

**🎯 Summary Table**

| **Constraint** | **Description** |
| --- | --- |
| NOT NULL | Prevents NULL values |
| UNIQUE | Ensures unique values in a column |
| PRIMARY KEY | Combines NOT NULL + UNIQUE |
| FOREIGN KEY | Enforces referential integrity |
| CHECK | Ensures a condition is met |
| DEFAULT | Sets a default value if none is provided |
| INDEX | Improves search performance |

**DML (Data Manipulation Language) :**

DML (Data Manipulation Language) is used to modify data in a database. It includes commands that insert, update, delete, and retrieve records.

**🔹 DML Commands:**

1. **INSERT** – Adds new records to a table.
2. **UPDATE** – Modifies existing records.
3. **DELETE** – Removes records from a table.
4. **SELECT** – Retrieves records from a table.

**INSERT Statement**

Used to add new records to a table.

🔹 **Example: Insert Single Record**

INSERT INTO Students (Student\_ID, Name, Age, Course)

VALUES (101, 'Alice', 20, 'Computer Science');

🔹 **Example: Insert Multiple Records**

INSERT INTO Students (Student\_ID, Name, Age, Course)

VALUES

(102, 'Bob', 22, 'Mechanical Engineering'),

(103, 'Charlie', 21, 'Electrical Engineering');

**UPDATE Statement**

Used to modify existing records.

🔹 **Example: Update a Single Record**

UPDATE Students

SET Age = 23

WHERE Student\_ID = 101;

🔹 **Example: Update Multiple Records**

UPDATE Students

SET Course = 'Data Science'

WHERE Course = 'Computer Science';

**DELETE Statement**

Used to remove records from a table.

🔹 **Example: Delete a Specific Record**

DELETE FROM Students

WHERE Student\_ID = 103;

🔹 **Example: Delete All Records in a Table**

DELETE FROM Students;

**⚠️ Warning:** DELETE removes data but **keeps the table structure**.

**SELECT Statement**

Used to retrieve records from a table.

🔹 **Example: Select All Columns**

SELECT \* FROM Students;

🔹 **Example: Select Specific Columns**

SELECT Name, Age FROM Students;

🔹 **Example: Filtering with WHERE**

SELECT \* FROM Students WHERE Age > 20;

🔹 **Example: Sorting Results**

SELECT \* FROM Students ORDER BY Age DESC;

**🎯 Summary Table**

| **Command** | **Description** |
| --- | --- |
| INSERT | Adds new records |
| UPDATE | Modifies existing records |
| DELETE | Removes records from a table |
| SELECT | Retrieves records |

**Different Ways to Insert Data in SQL (INSERT Statement)**

**Inserting a Single Row (Basic INSERT)**

This is the most common way to insert a single row into a table.

🔹 **Example:**

INSERT INTO Students (Student\_ID, Name, Age, Course)

VALUES (101, 'Alice', 20, 'Computer Science');

✔ *You must specify column names and corresponding values.*

**Inserting Multiple Rows at Once**

You can insert multiple rows in a single INSERT statement.

🔹 **Example:**

INSERT INTO Students (Student\_ID, Name, Age, Course)

VALUES

(102, 'Bob', 22, 'Mechanical Engineering'),

(103, 'Charlie', 21, 'Electrical Engineering'),

(104, 'David', 23, 'Civil Engineering');

✔ *Efficient when inserting multiple records at once.*

**Inserting Data Without Specifying Column Names**

If you're inserting values for **all columns**, you can skip the column names, but the values must be in the correct order.

🔹 **Example:**

INSERT INTO Students

VALUES (105, 'Emma', 19, 'Biotechnology');

⚠ *Not recommended if table structure changes in the future.*

**Inserting Data from Another Table (INSERT INTO SELECT)**

This method is useful when you want to copy data from one table to another.

🔹 **Example:**

INSERT INTO Graduate\_Students (Student\_ID, Name, Course)

SELECT Student\_ID, Name, Course FROM Students WHERE Age > 21;

✔ *Efficient for migrating data between tables.*

**Inserting Data Using DEFAULT Values**

If a column has a **DEFAULT** constraint, you can use DEFAULT keyword to insert default values.

🔹 **Example:**

INSERT INTO Students (Student\_ID, Name, Age, Course)

VALUES (106, 'Frank', DEFAULT, 'Mathematics');

✔ *Useful when certain fields have default values.*

**Inserting Data Using Subqueries**

You can use a SELECT query inside an INSERT statement.

🔹 **Example:**

INSERT INTO Alumni (Student\_ID, Name)

SELECT Student\_ID, Name FROM Students WHERE Age > 22;

✔ *Helps in dynamic data insertion based on conditions.*

**Inserting NULL Values**

If a column allows NULL, you can insert NULL explicitly.

🔹 **Example:**

INSERT INTO Students (Student\_ID, Name, Age, Course)

VALUES (107, 'Grace', NULL, 'Physics');

✔ *Ensures missing data is handled properly.*

**🎯 Summary Table**

| **Insertion Method** | **Example** |
| --- | --- |
| **Single Row** | INSERT INTO Students VALUES (101, 'Alice', 20, 'CS'); |
| **Multiple Rows** | INSERT INTO Students VALUES (102, 'Bob', 22, 'ME'), (103, 'Charlie', 21, 'EE'); |
| **Without Column Names** | INSERT INTO Students VALUES (104, 'David', 23, 'CE'); |
| **Using INSERT INTO SELECT** | INSERT INTO Alumni SELECT Student\_ID, Name FROM Students WHERE Age > 22; |
| **Using DEFAULT Values** | INSERT INTO Students VALUES (105, 'Emma', DEFAULT, 'Bio'); |
| **Using Subquery** | INSERT INTO Alumni (Student\_ID, Name) SELECT Student\_ID, Name FROM Students WHERE Age > 22; |
| **Inserting NULL Values** | INSERT INTO Students VALUES (106, 'Frank', NULL, 'Math'); |

**Different Ways to Use UPDATE Statement**

The UPDATE statement modifies existing records in a table.

**🔹 Basic Update (Updating a Single Column)**

UPDATE Students

SET Age = 22

WHERE Student\_ID = 101;

✔ Updates only the Age column for the student with Student\_ID = 101.

**🔹 Updating Multiple Columns**

UPDATE Students

SET Age = 23, Course = 'Data Science'

WHERE Student\_ID = 102;

✔ Updates Age and Course for the student with Student\_ID = 102.

**🔹 Updating Multiple Rows Based on a Condition**

UPDATE Students

SET Course = 'Artificial Intelligence'

WHERE Course = 'Computer Science';

✔ Changes the course name for all students in "Computer Science" to "Artificial Intelligence."

**🔹 Using Subquery in UPDATE**

UPDATE Students

SET Age = (SELECT AVG(Age) FROM Students)

WHERE Course = 'Data Science';

✔ Sets the Age to the average age of all students who are in the "Data Science" course.

**🔹 Updating Without WHERE (Updates All Records)**

UPDATE Students

SET Age = Age + 1;

✔ Increments the Age of **all** students in the table by 1. ⚠ *Be cautious while using this!*

**🎯 Summary Table**

| **Operation** | **Example** | **Description** |
| --- | --- | --- |
| **Basic Update** | UPDATE Students SET Age = 22 WHERE Student\_ID = 101; | Updates a single column for one row |
| **Update Multiple Columns** | UPDATE Students SET Age = 23, Course = 'Data Science' WHERE Student\_ID = 102; | Updates multiple columns |
| **Update Multiple Rows** | UPDATE Students SET Course = 'AI' WHERE Course = 'CS'; | Updates multiple rows based on condition |
| **Update with Subquery** | UPDATE Students SET Age = (SELECT AVG(Age) FROM Students) WHERE Course = 'Data Science'; | Updates using a subquery |
| **Update All Rows** | UPDATE Students SET Age = Age + 1; | Updates all rows in the table |

**Different Ways to Use DELETE Statement**

The DELETE statement removes records from a table.

**🔹 Basic Delete (Deleting a Specific Record)**

DELETE FROM Students

WHERE Student\_ID = 103;

✔ Deletes the student with Student\_ID = 103.

**🔹 Deleting Multiple Rows**

DELETE FROM Students

WHERE Age > 25;

✔ Deletes all students older than 25.

**🔹 Using DELETE with LIMIT (Deletes a Limited Number of Rows)**

DELETE FROM Students

WHERE Course = 'Mechanical Engineering'

LIMIT 2;

✔ Deletes **only 2** students from "Mechanical Engineering."  
(Note: LIMIT is supported in MySQL but not in some databases like SQL Server.)

**🔹 Using Subquery in DELETE**

DELETE FROM Students

WHERE Student\_ID IN (SELECT Student\_ID FROM Alumni);

✔ Deletes all students whose IDs are present in the Alumni table.

**🔹 Deleting Without WHERE (Deletes All Rows 😱)**

DELETE FROM Students;

✔ Deletes **all records** in the Students table but **keeps the table structure**. ⚠ *Use with caution!*

**🔹 Using TRUNCATE Instead of DELETE**

TRUNCATE TABLE Students;

✔ Removes **all rows** from the table **faster** than DELETE.  
⚠ *Unlike DELETE, TRUNCATE does not allow a WHERE condition.*

**🎯 Summary Table**

| **Operation** | **Example** | **Description** |
| --- | --- | --- |
| **Basic Delete** | DELETE FROM Students WHERE Student\_ID = 103; | Deletes a single row |
| **Delete Multiple Rows** | DELETE FROM Students WHERE Age > 25; | Deletes multiple rows |
| **Delete with Limit** | DELETE FROM Students WHERE Course = 'ME' LIMIT 2; | Deletes a limited number of rows |
| **Delete with Subquery** | DELETE FROM Students WHERE Student\_ID IN (SELECT Student\_ID FROM Alumni); | Deletes rows using a subquery |
| **Delete All Rows** | DELETE FROM Students; | Deletes all rows (table structure remains) |
| **Truncate Table** | TRUNCATE TABLE Students; | Deletes all rows (faster than DELETE) |