**Functional Dependency:**

Functional dependency is a concept in database management that describes the relationship between attributes (columns) in a relational database table. It defines how the values in one attribute are dependent on the values in another attribute. In other words, it specifies the rules that govern how the values of one attribute can be determined by the values of another attribute within the same table.

In a functional dependency, there are two types of attributes:

1. Left-hand side (LHS): This is the attribute or set of attributes on which another attribute's value depends. It is denoted as X in the notation X -> Y, where X is the LHS and Y is the right-hand side.
2. Right-hand side (RHS): This is the attribute whose value depends on the attributes on the left-hand side. It is denoted as Y in the notation X -> Y.

A functional dependency is denoted as X -> Y, which means that for each unique combination of values in X, there is a unique value in Y.

For example, in a table of employees, if the employee's Social Security Number (SSN) uniquely determines their Name, we can represent this functional dependency as SSN -> Name.

**Functional Dependency with Real Example:**

Let's take an example to illustrate functional dependency. Consider a table that stores information about students in a school, including their Student ID, Name, Age, and Class:

**Student Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student ID** | **Name** | **Age** | **Class** |
| 1 | Alice | 15 | 10A |
| 2 | Bob | 16 | 11B |
| 3 | Charlie | 15 | 10A |

In this table, you can observe functional dependencies:

1. **Student ID -> Name:** Each Student ID uniquely determines the student's Name. For example, if you know Student ID 1, you can determine that the student's Name is Alice.
2. **Student ID -> Age:** Similarly, each Student ID uniquely determines the student's Age.
3. **Student ID -> Class:** Each Student ID uniquely determines the student's Class.

**functional dependency with a real-world example:**

Consider a database table that stores information about employees in a company:

Employee Table:

| EmployeeID | EmployeeName | Department | Salary |
| --- | --- | --- | --- |
| 1 | Alice | HR | 50000 |
| 2 | Bob | IT | 60000 |
| 3 | Charlie | Sales | 55000 |
| 4 | David | HR | 52000 |

In this Employee Table, we can identify some functional dependencies:

1. EmployeeID -> EmployeeName: Each unique EmployeeID uniquely determines the name of the employee. For example, if you know EmployeeID 1, you can conclude that the EmployeeName is Alice.
2. EmployeeID -> Department: Similarly, each EmployeeID uniquely determines the department the employee works in. EmployeeID 1 corresponds to the HR department.
3. EmployeeID -> Salary: Each EmployeeID uniquely determines the salary of the employee. EmployeeID 2 corresponds to a salary of $60,000.

These functional dependencies signify that EmployeeID is the primary key of this table since it uniquely determines all other attributes. In this context:

* EmployeeName, Department, and Salary are said to be functionally dependent on EmployeeID because you can determine their values based on the value of EmployeeID.

Functional dependencies are essential in database design and help ensure data integrity by minimizing redundancy and preventing data anomalies. They provide a clear structure for organizing data and enable efficient querying of the database.

**Data Redundancy:**

Data redundancy occurs when the same piece of data is stored in multiple places within a database. This redundancy can lead to several issues, including increased storage requirements, data inconsistency, and the potential for update anomalies.

Example of Data Redundancy:

Consider a database for a library where books are stored along with the information about the authors. Without proper normalization, you might have a single table like this:

Book Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BookID | Title | Author | Genre | ISBN |
| 1 | "The Catcher..." | J.D. Salinger | Fiction | 123456789 |
| 2 | "1984" | George Orwell | Fiction | 987654321 |
| 3 | "To Kill a..." | Harper Lee | Fiction | 456789123 |
| 4 | "1984" | George Orwell | Fiction | 987654321 |

In this example, there is redundancy in the book "1984" by George Orwell, which is listed twice. This redundancy can lead to data inconsistencies if one entry is updated, but the other is not. It also consumes more storage space than necessary.

**Data Anomalies:**

Data anomalies are problems that can occur in a database due to inconsistencies, redundancy, or improper data handling. There are three main types of data anomalies:

1. **Insertion Anomalies:** These occur when you can't insert data into the database without including additional, unrelated information. In the example above, you can't add a new book without specifying an author, even if the author is not relevant to the insertion.
2. **Update Anomalies:** Update anomalies happen when you modify data in one place but not in another, leading to inconsistencies. In the example above, if you update the author name for one occurrence of "1984," you must remember to update it in the other occurrence as well.
3. **Deletion Anomalies:** Deletion anomalies occur when deleting data leads to the unintentional loss of other data. In the example above, if you delete one occurrence of "1984," you lose information about the book, even though there's another copy of it.

**Real-world example of data anomalies:**

Let's consider a simple database for tracking customer orders and their associated order items:

Order Table:

| OrderID | CustomerName | OrderDate |
| --- | --- | --- |
| 1 | John Smith | 2023-01-15 |
| 2 | Alice Brown | 2023-01-18 |

Order Items Table:

| OrderItemID | OrderID | Product | Quantity | Price |
| --- | --- | --- | --- | --- |
| 101 | 1 | Laptop | 2 | $800 |
| 102 | 2 | Smartphone | 1 | $600 |
| 103 | 1 | Tablet | 3 | $300 |

Now, let's examine some potential data anomalies in this database:

1. Insertion Anomaly:

Suppose you want to add a new customer without them making any orders yet. In the current design, you cannot do so because the Order Table depends on having at least one order associated with a customer. This is an insertion anomaly because you can't insert a customer without also inserting an order.

1. Update Anomaly:

Imagine that John Smith decides to change his name to "John Johnson." To maintain data consistency, you need to update his name in both the Order Table and the Order Items Table. However, if you only update one of them, you'll have inconsistent data, leading to an update anomaly.

For instance, if you only update the Order Table:

Order Table:

| OrderID | CustomerName | OrderDate |
| --- | --- | --- |
| 1 | John Johnson | 2023-01-15 |
| 2 | Alice Brown | 2023-01-18 |

Order Items Table:

| OrderItemID | OrderID | Product | Quantity | Price |
| --- | --- | --- | --- | --- |
| 101 | 1 | Laptop | 2 | $800 |
| 102 | 2 | Smartphone | 1 | $600 |
| 103 | 1 | Tablet | 3 | $300 |

Now, the Order Items Table still has John Smith as the customer name, creating a data inconsistency.

1. Deletion Anomaly:

Let's say John decides to cancel his order (OrderID 1). If you delete this order from the Order Table, you also lose information about the products he ordered. This is a deletion anomaly because removing one piece of data (the order) leads to the unintentional loss of related data (the order items).

Data anomalies like these can be eliminated or minimized through proper database design techniques, including normalization, to ensure data integrity and consistency.

**Normalization in DBMS**

normalization in DBMS is a technique using which you can organize the data in the database tables so that:

* There is less repetition of data,
* A large set of data is structured into a bunch of smaller tables,
* and the tables have a proper relationship between them.

DBMS Normalization is a systematic approach to **decompose (break down) tables** to eliminate data redundancy(repetition) and undesirable characteristics like Insertion anomaly in DBMS, Update anomaly in DBMS, and Delete anomaly in DBMS.

It is a **multi-step process** that puts data into tabular form, removes duplicate data, and set up the relationship between tables

## **Why we need Normalization in DBMS?**

Normalization is required for,

* Eliminating redundant(useless) data, therefore handling **data integrity**, because if data is repeated it increases the chances of inconsistent data.
* Normalization helps in keeping **data consistent** by storing the data in one table and referencing it everywhere else.
* Storage optimization although that is not an issue these days because Database storage is cheap.
* Breaking down large tables into smaller tables with relationships, so it makes the database structure more scalable and adaptable.
* Ensuring data dependencies make sense i.e. data is logically stored.

## **Problems without Normalization in DBMS**

If a table is not properly normalized and has data redundancy(repetition) then it will not only **eat up extra memory space** but will also make it difficult for you to handle and update the data in the database, without losing data.

Insertion, Updation, and Deletion Anomalies are very frequent if the database is not normalized.

To understand these anomalies let us take an example of a **Student** table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rollno** | **name** | **branch** | **hod** | **office\_tel** |
| 401 | Akon | CSE | Mr. X | 53337 |
| 402 | Bkon | CSE | Mr. X | 53337 |
| 403 | Ckon | CSE | Mr. X | 53337 |
| 404 | Dkon | CSE | Mr. X | 53337 |

In the table above, we have data for four Computer Sci. students.

As we can see, data for the fields **branch**, **hod**(Head of Department), and **office\_tel** are repeated for the students who are in the same branch in the college, this is **Data Redundancy**.

### **1. Insertion Anomaly in DBMS**

* Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as **NULL**.
* Also, if we have to insert data for 100 students of the same branch, then the branch information will be repeated for all those 100 students.
* These scenarios are nothing but **Insertion anomalies**.
* If you have to repeat the same data in every row of data, it's better to **keep the data separately** and **reference that data** in each row.
* So in the above table, we can keep the branch information separately, and just use the **branch\_id** in the student table, where **branch\_id** can be used to get the branch information.

### **2. Updation Anomaly in DBMS**

* What if Mr. X leaves the college? or Mr. X is no longer the HOD of the computer science department? In that case, all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency.
* This is an Updation anomaly because you need to update all the records in your table just because one piece of information got changed.

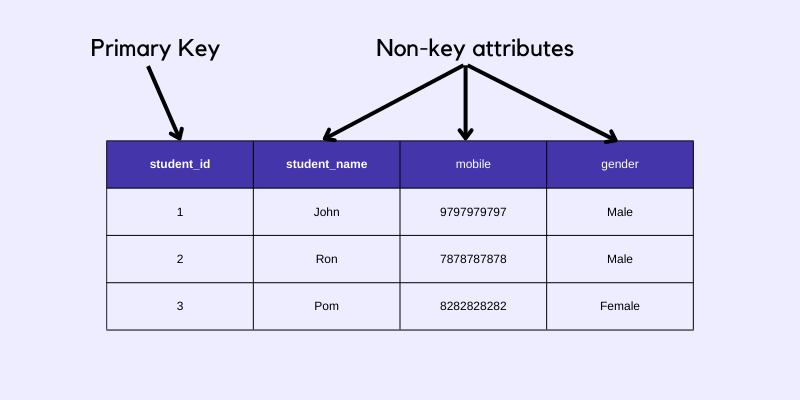
### **3. Deletion Anomaly in DBMS**

* In our **Student** table, two different pieces of information are kept together, the **Student information** and the **Branch information**.
* So if only a single student is enrolled in a branch, and that student leaves the college, or for some reason, the entry for the student is deleted, we will lose the branch information too.
* So never in DBMS, we should keep two different entities together, which in the above example is Student and branch,

The solution for all the three anomalies described above is to keep the **student** information and the **branch** information in two different tables. And use the **branch\_id** in the student table to reference the branch.

## **Primary Key and Non-key attributes**

Before we move on to learn different Normal Forms in DBMS, let's first understand what is a primary key and what are non-key attributes.



As you can see in the table above, the **student\_id** column is a **primary key** because using the **student\_id** value we can uniquely identify each row of data, hence the *remaining columns* then become the **non-key attributes**.

## **Types of DBMS Normal forms**

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

### **1. First Normal Form (1NF)**

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single(**atomic**) valued attributes/columns.
2. Values stored in a column should be of the same domain.
3. All the columns in a table should have unique names.
4. And the order in which data is stored should not matter.

Let's see an example.

If we have an **Employee** table in which we store the employee information along with the employee skillset, the table will look like this:

| **emp\_id** | **emp\_name** | **emp\_mobile** | **emp\_skills** |
| --- | --- | --- | --- |
| 1 | John Tick | 9999957773 | Python, JavaScript |
| 2 | Darth Trader | 8888853337 | HTML, CSS, JavaScript |
| 3 | Rony Shark | 7777720008 | Java, Linux, C++ |

The above table has 4 columns:

* All the columns have different names.
* All the columns hold values of the same type like **emp\_name** has all the names, **emp\_mobile** has all the contact numbers, etc.
* The order in which we save data doesn't matter
* But the **emp\_skills** column holds multiple comma-separated values, while as per the First Normal form, each column should have a single value.

**Hence the above table fails to pass the First Normal form.**

**So how do you fix the above table? There are two ways to do this:**

1. Remove the **emp\_skills** column from the **Employee** table and keep it in some other table.
2. Or add multiple rows for the employee and each row is linked with one skill.

#### **1. Create Separate tables for Employee and Employee Skills**

So the **Employee** table will look like this,

| **emp\_id** | **emp\_name** | **emp\_mobile** |
| --- | --- | --- |
| 1 | John Tick | 9999957773 |
| 2 | Darth Trader | 8888853337 |
| 3 | Rony Shark | 7777720008 |

And the new **Employee\_Skill** table:

| **emp\_id** | **emp\_skill** |
| --- | --- |
| 1 | Python |
| 1 | JavaScript |
| 2 | HTML |
| 2 | CSS |
| 2 | JavaScript |
| 3 | Java |
| 3 | Linux |
| 3 | C++ |

#### **2. Add Multiple rows for Multiple skills**

You can also simply add multiple rows to add multiple skills. This will lead to repetition of the data, but that can be handled as you further Normalize your data using the Second Normal form and the Third Normal form.

| **emp\_id** | **emp\_name** | **emp\_mobile** | **emp\_skill** |
| --- | --- | --- | --- |
| 1 | John Tick | 9999957773 | Python |
| 1 | John Tick | 9999957773 | JavaScript |
| 2 | Darth Trader | 8888853337 | HTML |
| 2 | Darth Trader | 8888853337 | CSS |
| 2 | Darth Trader | 8888853337 | JavaScript |
| 3 | Rony Shark | 7777720008 | Java |
| 3 | Rony Shark | 7777720008 | Linux |
| 3 | Rony Shark | 7777720008 | C++ |

### **2. Second Normal Form (2NF)**

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have **Partial Dependency**.

Let's take an example to understand Partial dependency and the Second Normal Form.

#### **What is Partial Dependency?**

When a table has a primary key that is made up of two or more columns, then all the columns (not included in the primary key) in that table should depend on the entire primary key and not on a part of it. If any column (which is not in the primary key) depends on a part of the primary key then we say we have Partial dependency in the table.

Confused? Let's take an example.

If we have two tables Students and Subjects, to store student information and information related to subjects.

**Student** table:

| **student\_id** | **student\_name** | **branch** |
| --- | --- | --- |
| 1 | Akon | CSE |
| 2 | Bkon | Mechanical |

**Subject** Table:

| **subject\_id** | **subject\_name** |
| --- | --- |
| 1 | C Language |
| 2 | DSA |
| 3 | Operating System |

And we have another table **Score** to store the marks scored by students in any subject like this,

| **student\_id** | **subject\_id** | **marks** | **teacher\_name** |
| --- | --- | --- | --- |
| 1 | 1 | 70 | Miss. C |
| 1 | 2 | 82 | Mr. D |
| 2 | 1 | 65 | Mr. Op |

Now in the above table, the primary key is **student\_id + subject\_id**, because both this information are required to select any row of data.

But in the **Score** table, we have a column **teacher\_name**, which depends on the subject information or just the **subject\_id**, so we should not keep that information in the **Score** table.

The column **teacher\_name** should be in the **Subjects** table. And then the entire system will be Normalized as per the Second Normal Form.

Updated **Subject** table:

| **subject\_id** | **subject\_name** | **teacher\_name** |
| --- | --- | --- |
| 1 | C Language | Miss. C |
| 2 | DSA | Mr. D |
| 3 | Operating System | Mr. Op |

Updated **Score** table:

| **student\_id** | **subject\_id** | **marks** |
| --- | --- | --- |
| 1 | 1 | 70 |
| 1 | 2 | 82 |
| 2 | 1 | 65 |

### **3. Third Normal Form (3NF)**

A table is said to be in the Third Normal Form when,

1. It satisfies the First Normal Form and the Second Normal form.
2. And, it doesn't have Transitive Dependency.

#### **What is Transitive Dependency?**

In a table we have some column that acts as the primary key and other columns depends on this column. But what if a column that is not the primary key depends on another column that is also not a primary key or part of it? Then we have Transitive dependency in our table.

Let's take an example. We had the **Score** table in the Second Normal Form above. If we have to store some extra information in it, like,

1. **exam\_type**
2. **total\_marks**

To store the type of exam and the total marks in the exam so that we can later calculate the percentage of marks scored by each student.

The **Score** table will look like this,

| **student\_id** | **subject\_id** | **marks** | **exam\_type** | **total\_marks** |
| --- | --- | --- | --- | --- |
| 1 | 1 | 70 | Theory | 100 |
| 1 | 2 | 82 | Theory | 100 |
| 2 | 1 | 42 | Practical | 50 |

* In the table above, the column **exam\_type** depends on both **student\_id** and **subject\_id**, because,
  + a student can be in the CSE branch or the Mechanical branch,
  + and based on that they may have different exam types for different subjects.
  + The CSE students may have both Practical and Theory for Compiler Design,
  + whereas Mechanical branch students may only have Theory exams for Compiler Design.
* But the column **total\_marks** just depends on the **exam\_type** column. And the **exam\_type** column is not a part of the primary key. Because the primary key is **student\_id + subject\_id**, hence we have a Transitive dependency here.

#### **How to Transitive Dependency?**

You can create a separate table for **ExamType** and use it in the **Score** table.

New **ExamType** table,

| **exam\_type\_id** | **exam\_type** | **total\_marks** | **duration** |
| --- | --- | --- | --- |
| 1 | Practical | 50 | 45 |
| 2 | Theory | 100 | 180 |
| 3 | Workshop | 150 | 300 |

We have created a new table **ExamType** and we have added more related information in it like **duration** (duration of exam in mins.), and now we can use the **exam\_type\_id** in the **Score** table.

### **4. Boyce-Codd Normal Form (BCNF)**

* **Boyce and Codd Normal Form** is a higher version of the Third Normal Form.
* This form deals with a certain type of anomaly that is not handled by 3NF.
* A 3NF table that does not have **multiple overlapping candidate keys** is said to be in BCNF.
* For a table to be in BCNF, the following conditions must be satisfied:
  + R must be in the 3rd Normal Form
  + and, for each functional dependency ( X → Y ), X should be a Super Key.

### **5. Fourth Normal Form (4NF)**

A table is said to be in the Fourth Normal Form when,

1. It is in the Boyce-Codd Normal Form.
2. And, it doesn't have Multi-Valued Dependency.

### **5. Fifth Normal Form (5NF)**

* The fifth normal form is also called the **PJNF** - **Project-Join Normal Form**
* It is the most advanced level of Database Normalization.
* Using Fifth Normal Form you can fix **Join dependency** and reduce data redundancy.
* It also helps in fixing **Update anomalies** in DBMS design.

**ACID**

is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties are essential in ensuring the reliability and integrity of database transactions:

1. Atomicity: Atomicity guarantees that a transaction is treated as a single, indivisible unit of work. It ensures that all the operations within a transaction are either completed entirely or have no effect at all. If any part of a transaction fails, the entire transaction is rolled back, and the database returns to its previous state.
2. Consistency: Consistency ensures that a transaction brings the database from one consistent state to another. It means that the data must meet all integrity constraints defined in the database schema after a transaction is executed. If a transaction violates any constraint, it is rolled back, and the database remains unchanged.
3. Isolation: Isolation ensures that concurrent transactions do not interfere with each other. Each transaction appears to be executed in isolation from other transactions, even if they are executed concurrently. This prevents issues like dirty reads, non-repeatable reads, and phantom reads.
4. Durability: Durability guarantees that once a transaction is successfully completed and committed, its changes are permanent and will survive any subsequent system failures. This is usually achieved by writing transaction changes to non-volatile storage like disk.

**The ACID properties and the concept of a database instance and schema with examples:**

**ACID Properties:**

Let's consider a banking system where customers can transfer money between their accounts. We'll use the ACID properties to ensure data integrity:

1. **Atomicity:**

Imagine a customer initiates a fund transfer from their savings account to their checking account. The transaction should either complete entirely or fail entirely. If it fails, no changes should occur. Atomicity ensures this.

BEGIN TRANSACTION;

UPDATE SavingsAccount SET Balance = Balance - 100 WHERE AccountNumber = '123';

UPDATE CheckingAccount SET Balance = Balance + 100 WHERE AccountNumber = '456';

COMMIT;

1. If any of these SQL statements fails, the entire transaction is rolled back, ensuring atomicity.
2. **Consistency:**

Let's say there's a constraint that ensures the sum of balances in all accounts is always greater than zero. If a transaction violates this constraint, it is rolled back to maintain consistency.

1. **Isolation:**

Consider two customers transferring money simultaneously. The isolation property ensures that one customer's transaction doesn't interfere with the other's.

* + **Customer 1's transaction:**

BEGIN TRANSACTION;

UPDATE SavingsAccount SET Balance = Balance - 100 WHERE AccountNumber = '123';

UPDATE CheckingAccount SET Balance = Balance + 100 WHERE AccountNumber = '456';

COMMIT;

**Customer 2's transaction (concurrently):**

BEGIN TRANSACTION;

UPDATE SavingsAccount SET Balance = Balance - 50 WHERE AccountNumber = '789';

UPDATE CheckingAccount SET Balance = Balance + 50 WHERE AccountNumber = '101';

COMMIT;

Isolation ensures that these two transactions don't interfere with each other, even if they run concurrently.

1. **Durability:**

Once a transaction is committed, its changes must be permanent, surviving system failures. The RDBMS ensures that the changes are written to durable storage like disk.

**Concept of Database Instance and Schema:**

1. **Database Instance:** A database instance refers to a running, operational database system. It includes the database software, memory structures, and processes necessary to manage and serve data. In a multi-user environment, multiple database sessions can connect to the same database instance simultaneously. Each database instance typically corresponds to a specific RDBMS (Relational Database Management System) like MySQL, PostgreSQL, Oracle, or SQL Server.
   * **Key Characteristics of a Database Instance:**
     + It manages and maintains the actual data stored in the database.
     + It controls access to the data and enforces security and integrity rules.
     + It handles concurrent user requests and manages resources like CPU and memory.
     + It can host one or more databases, each with its schema and data.
2. **Schema:** A schema is a logical container within a database that organizes database objects, such as tables, views, indexes, and stored procedures. It provides a way to group related objects together and define the structure of the database. Each database can have one or more schemas, and each schema can contain multiple database objects.
   * **Key Points about Schemas:**
     + Schemas help in organizing and managing database objects, especially in large databases.
     + They provide a way to segregate and control access to objects based on user roles and privileges.
     + Objects within different schemas can have the same name without conflicts.
     + Schemas are a part of the database's metadata and do not directly store data; they define the structure and organization of data.

* **Database Instance:**
  + Running instance of our RDBMS (e.g., MySQL Server).
  + Manages and serves data for multiple databases.
* **Schema:**
  + Within "BankDB," we have multiple schemas, one for each banking service.
  + The "Accounts" schema may contain tables like SavingsAccount and CheckingAccount.

CREATE DATABASE BankDB;

USE BankDB;

-- Create a schema for accounts

CREATE SCHEMA Accounts;

-- Create tables within the schema

CREATE TABLE Accounts.SavingsAccount (

AccountNumber INT PRIMARY KEY,

Balance DECIMAL(10, 2)

);

CREATE TABLE Accounts.CheckingAccount (

AccountNumber INT PRIMARY KEY,

Balance DECIMAL(10, 2)

);

These schemas help us organize our data, manage access, and prevent naming conflicts.

In this example, the database instance manages multiple databases, and each database can have its own set of schemas. Schemas allow us to structure and organize database objects within a database.

**MySQL Storage Engines (InnoDB, MyISAM and others)**

MySQL offers different storage engines, also known as table types, to manage how data is stored, accessed, and manipulated within a database. Each storage engine has its own characteristics, performance characteristics, and features. Two of the most commonly used storage engines in MySQL are InnoDB and MyISAM, but there are others as well. Let's explore some of the popular MySQL storage engines:

1. **InnoDB:**
   * **ACID Compliant:** InnoDB is ACID-compliant, which means it supports transactions with properties like atomicity, consistency, isolation, and durability.
   * **Referential Integrity:** It supports foreign keys and enforces referential integrity constraints, making it suitable for applications that require complex relationships between tables.
   * **Row-Level Locking:** InnoDB uses row-level locking, allowing multiple transactions to work on different rows simultaneously without blocking each other.
   * **Crash Recovery:** It has crash recovery mechanisms, ensuring data durability even in the event of a system failure.
   * **Default Storage Engine:** InnoDB is the default storage engine for MySQL since version 5.5.
2. **MyISAM:**
   * **Non-Transactional:** MyISAM lacks transaction support, making it less suitable for applications where data consistency and integrity are critical.
   * **Table-Level Locking:** MyISAM uses table-level locking, which can lead to contention when multiple transactions want to access the same table simultaneously.
   * **Fast Reads:** It performs well for read-intensive workloads, such as search or logging systems, due to its simple structure and indexing.
   * **Full-Text Search:** MyISAM includes built-in support for full-text search capabilities.
   * **No Foreign Key Constraints:** It does not support foreign key constraints, which can lead to data integrity issues if not carefully managed.
3. **MEMORY (HEAP):**
   * **In-Memory Storage:** The MEMORY engine stores data in memory rather than on disk, providing extremely fast read and write operations.
   * **Non-Persistent:** Data is not persistent; it's lost when the MySQL server is restarted.
   * **Used for Caching:** Often used for caching temporary data, session management, or temporary working tables.
4. **NDB (MySQL Cluster):**
   * **Distributed Database:** NDB is designed for high-availability and high-performance, supporting distributed databases and data partitioning.
   * **ACID Compliant:** It provides ACID compliance and supports real-time applications requiring low-latency access to data.
   * **Data Replication:** Supports synchronous replication for data redundancy and high availability.
5. **CSV:**
   * **CSV Format:** CSV stores data in comma-separated values format. It's useful for data interchange between MySQL and other applications.
6. **JSON:**
   * **JSON Storage:** The JSON engine is used for storing and querying JSON data efficiently. It's available in MySQL 5.7 and later.