MySQL's Working Principles: Storage & Access Mechanisms

MySQL is a **relational database management system (RDBMS)** that stores, retrieves, and manages structured data using SQL. Here’s a deep dive into how MySQL **stores, organizes, and accesses** data internally.

---

1. MySQL Architecture Overview

MySQL follows a **client-server architecture**, meaning multiple clients can connect to the MySQL server to perform database operations.

Components of MySQL Architecture:

1. **Client Layer** (Front-end)
    - Users interact via MySQL clients, GUI tools (like MySQL Workbench), or applications.
    - Clients send SQL queries to the MySQL **server** over a network.
2. **SQL Layer** (Query Processing)
    - This layer processes SQL queries.
    - It includes:
        - **Query Parser** → Checks SQL syntax.
        - **Optimizer** → Finds the best execution plan for queries.
        - **Execution Engine** → Executes queries.
3. **Storage Engine Layer** (Back-end)
    - Data is stored in **storage engines** like InnoDB and MyISAM.
    - MySQL **manages transactions, indexing, and caching** at this level.
4. **File System & Data Storage**
    - MySQL interacts with the operating system to store data on disk.

---

2. Storage Engines (How Data is Stored)

MySQL uses **pluggable storage engines** to store and retrieve data. The two most common storage engines are:

InnoDB (Default)

- Supports **ACID transactions** (Atomicity, Consistency, Isolation, Durability).
- Uses **row-level locking** for better concurrency.
- Supports **foreign keys** (relations between tables).
- Stores data in **tablespaces**, using clustered indexes.

File Storage:

- `.ibd` → Contains table data and indexes.

- `ib_logfileX` → Stores redo logs for crash recovery.

MyISAM (Legacy)

- Faster for read-heavy applications but lacks transactions.
- Uses **table-level locking** (less concurrency than InnoDB).
- Does not support foreign keys.

File Storage:

- `.frm` → Stores table structure.
- `.MYD` → Stores table data.
- `.MYI` → Stores table indexes.

When to use?

- Use **InnoDB** for most applications (transactions, data integrity).
- Use **MyISAM** for **read-heavy workloads** (e.g., search engines, logging).

---

3. How MySQL Stores Data Internally

- **Databases** → Stored as **directories** inside MySQL's data directory.
- **Tables** → Stored as **files** within database directories.
- **Rows & Columns** → Stored as **B-Tree indexes** in tablespaces (for InnoDB).

Example:
For a database named `company`, the files are stored in:

```
swift
CopyEdit
/var/lib/mysql/company/
```

Each table has corresponding `.ibd` (InnoDB) or `.MYD/.MYI` (MyISAM) files.

---

4. Query Execution Process (How MySQL Accesses Data)

Step-by-Step Execution of a Query

Example Query:

```
sql
CopyEdit
SELECT name FROM employees WHERE salary > 50000;
```

MySQL Execution Process: **Client Request:** The MySQL client sends the SQL query to the server.

Query Parsing: The **parser** checks SQL syntax.

Query Optimization: MySQL **chooses the best way** to retrieve data using indexes and

execution plans.

Execution: The MySQL engine retrieves data from **cache** (if available) or **disk**.

Return Data: The result is sent back to the client.

---

5. Indexing (How MySQL Speeds Up Queries)

Indexes improve query performance by reducing the number of rows MySQL must scan.

Types of Indexes in MySQL

1. **Primary Index** → Automatically created on primary key columns.
2. **Secondary Index** → Created on frequently queried columns.
3. **Full-text Index** → Used for text-based searches.
4. **Spatial Index** → Used for GIS (geographic data).

How Indexes Work

- MySQL **uses B-Trees** (Balanced Trees) for indexing.
- **Without an index:** MySQL scans the entire table (`Full Table Scan`).
- **With an index:** MySQL **quickly finds data** (`Index Scan`).

Creating an Index

```
sql
CopyEdit
CREATE INDEX idx_salary ON employees(salary);
```

Checking Index Usage

```
sql
CopyEdit
EXPLAIN SELECT name FROM employees WHERE salary > 50000;
```

---

6. MySQL Caching (How MySQL Speeds Up Access)

MySQL Uses Multiple Caches:

1. **Query Cache (Deprecated in MySQL 8)**
    - Stores query results to avoid re-execution.
2. **InnoDB Buffer Pool (Most Important)**
    - Caches frequently accessed **data & indexes** in memory.
    - Improves read performance.
3. **Key Cache (For MyISAM)**
    - Speeds up index lookups in MyISAM tables.

Tuning MySQL Performance To check cache usage:

```
sql
```

```
CopyEdit
SHOW STATUS LIKE 'Innodb_buffer_pool%';
```

---

7. Transactions & Concurrency Control

A **transaction** ensures multiple SQL operations are executed together.

ACID Properties in MySQL (InnoDB)

- **Atomicity** → All or nothing execution.
- **Consistency** → Database remains in a valid state.
- **Isolation** → Multiple transactions don't interfere.
- **Durability** → Changes are saved permanently.

Example: Transaction with Rollback

```
sql
CopyEdit
START TRANSACTION;
UPDATE employees SET salary = salary + 5000 WHERE department_id = 1;
ROLLBACK;  -- Undo changes
```

To save changes:

```
sql
CopyEdit
COMMIT;
```

---

8. MySQL Locking Mechanisms

Locks prevent conflicts when multiple users modify the same data.

Locking in MySQL

- **Row-level locking (InnoDB)** → Locks only the affected rows.
- **Table-level locking (MyISAM)** → Locks the entire table.
- **Shared locks (`LOCK IN SHARE MODE`)** → Multiple reads allowed, no writes.
- **Exclusive locks (`FOR UPDATE`)** → Prevents other transactions from reading.

Example of Locking

```
sql
CopyEdit
SELECT * FROM employees WHERE id = 1 FOR UPDATE;
```

---

9. Data Backup & Recovery

Backup Methods:

1. **mysqldump (Logical Backup)**

   ```
   bash
   CopyEdit
   mysqldump -u root -p company > backup.sql
   ```

2. **Physical Backup (Copying Data Files)**

   ```
   bash
   CopyEdit
   cp -r /var/lib/mysql/company /backup/
   ```

3. **Replication (For High Availability)**
   - Master-Slave or Master-Master Replication.

Restoring from Backup

```
bash
CopyEdit
mysql -u root -p company < backup.sql
```

---

10. High Availability & Scaling

Replication (Master-Slave)

- Copies data from **Master** to **Slave servers**.
- Ensures **failover & load balancing**.

Sharding

- Splits data across multiple servers to handle large-scale applications.

Partitioning

- Divides large tables into smaller partitions for fast queries.

Conclusion

How MySQL Works Internally:

- Data is stored in **files and tablespaces** using **storage engines** (InnoDB, MyISAM).
- Queries are **parsed, optimized, executed**, and **fetched from cache or disk**.
- **Indexes and caching** speed up performance.
- **Transactions & locks** ensure data integrity.
- **Replication & partitioning** enable scalability.

Detailed Breakdown of MySQL Architecture

MySQL follows a **client-server architecture** that allows multiple clients (applications, users) to interact with the database server simultaneously. The architecture is divided into several **logical layers**, each handling different tasks like query parsing, optimization, execution, and storage.

MySQL Architecture Overview

The architecture consists of **three major layers**:

1. **Client Layer (Connection & Communication)**
2. **SQL Layer (Query Processing & Optimization)**
3. **Storage Engine Layer (Data Storage & Retrieval)**

Let's explore each in depth.

Client Layer (Connection & Communication)

This is the **entry point** where MySQL interacts with applications and users.

Components:

1. **Clients & Interfaces**
    - MySQL clients send SQL queries via:
        - MySQL Command-Line Client (`mysql -u root -p`)
        - GUI Tools (MySQL Workbench, phpMyAdmin)
        - Application APIs (JDBC, ODBC, Python Connector)
2. **Connection Handling & Authentication**
    - MySQL uses **thread-based connection handling**.
    - Each client gets a **separate thread** for communication.
    - **Authentication Plugins** handle login security (e.g., `mysql_native_password`, `caching_sha2_password`).
3. **Connection Pooling (Performance Optimization)**
    - MySQL maintains a pool of reusable connections to reduce overhead.
    - Example: `thread_pool_size` can be adjusted for handling high traffic.

Key Takeaway: The Client Layer **authenticates users**, manages **connections**, and forwards SQL queries to the **SQL Layer**.

SQL Layer (Query Processing & Optimization)

This is the **brain** of MySQL that processes queries before execution.

Components:

1. **Query Parser (Syntax & Semantic Checking)**
    - Breaks down SQL queries into **tokens**.
    - Checks for **syntax errors**.
    - Ensures column names, table names, and commands are **valid**.

    Example

```
sql
CopyEdit
SELECT * FROM employees WHERE id = 'ABC'; -- Error: id is INT, not a
string
```

2. **Query Optimizer (Execution Planning)**
    - Determines the **fastest way** to execute queries.
    - Uses **indexes, statistics, and caching**.
    - Considers different **query execution plans** and chooses the best one.

    How to Check Query Optimization?

```
sql
CopyEdit
EXPLAIN SELECT * FROM employees WHERE salary > 50000;
```

    - **Index Scan** (Fast) → Uses indexes to find matching rows.
    - **Full Table Scan** (Slow) → Scans every row.
3. **Query Execution Engine**
    - Executes SQL queries **step-by-step**.
    - Interacts with the **storage engine** to retrieve data.

Key Takeaway: The SQL Layer **parses, optimizes, and executes queries** efficiently.

Storage Engine Layer (Data Storage & Retrieval)

This is the **backend layer** responsible for **storing and retrieving data**.

Storage Engines in MySQL

MySQL uses a **pluggable storage engine architecture**, meaning different engines can be used depending on requirements.

Storage Engine	Features
InnoDB (Default)	Transactions, Foreign Keys, Row-level Locking
MyISAM	Fast Reads, Table-level Locking
Memory (HEAP)	Stores Data in RAM (Fast, but Non-Persistent)
CSV	Stores Data as Plain Text (Comma-Separated)
Archive	Optimized for Storing Large Amounts of Log Data
NDB (Clustered)	High Availability & Fault Tolerance (Used in MySQL Cluster)

How MySQL Stores Data Internally?

MySQL **stores databases as directories** and **tables as files** in its data directory:

```
swift
CopyEdit
/var/lib/mysql/{database_name}/
```

Each table consists of:

- **`table_name.frm`** → Stores table definition (pre-MySQL 8)
- **`table_name.ibd`** → Stores table data & indexes (InnoDB)
- **`table_name.MYD`** → Stores table data (MyISAM)
- **`table_name.MYI`** → Stores indexes (MyISAM)

Example: Finding Storage Files

```
bash
CopyEdit
ls -l /var/lib/mysql/employees/
```

Key Takeaway: The Storage Engine Layer **manages data storage, retrieval, and indexing**.

Additional Components in MySQL Architecture

Buffer Pool (Caching Mechanism)

- MySQL **caches frequently accessed data** in memory.
- **InnoDB Buffer Pool** is critical for performance.

Check Buffer Pool Usage

```
sql
CopyEdit
SHOW STATUS LIKE 'Innodb_buffer_pool%';
```

Transaction Management (ACID Compliance)

- MySQL **ensures data integrity** using transactions.
- **ACID Properties:**
  - **Atomicity** → All or nothing execution.
  - **Consistency** → Keeps database in a valid state.
  - **Isolation** → Prevents transactions from interfering.
  - **Durability** → Ensures committed transactions are saved permanently.

Example: Transaction with Rollback

```
sql
CopyEdit
START TRANSACTION;
UPDATE employees SET salary = salary + 5000 WHERE department_id = 1;
ROLLBACK;
```

MySQL Query Execution Process (Step-by-Step)

Let's go through the entire **query execution flow**.

Example Query

```
sql
CopyEdit
SELECT name FROM employees WHERE salary > 50000;
```

Step-by-Step Execution

Step	Component	Description
Connection	Client Layer	MySQL receives query from user/application
Parsing	Query Parser	MySQL checks syntax and semantics
Optimization	Query Optimizer	MySQL determines the best execution plan
Execution	Query Execution Engine	MySQL fetches data from the storage engine
Caching	Buffer Pool	Frequently accessed data is cached
Response	Client Layer	MySQL sends results back to client

High Availability & Scalability

MySQL provides **high availability** using **replication** and **clustering**.

Replication (Master-Slave)

- **Master Server** → Handles writes.
- **Slave Servers** → Handle reads (reducing load on master).

Check Replication Status

```
sql
CopyEdit
SHOW SLAVE STATUS\G;
```

MySQL Clustering (NDB)

- Used for **fault tolerance** and **scalability**.
- Multiple servers work together in **real-time**.

Performance Optimization Techniques

1. **Use Indexes** (`PRIMARY KEY`, `UNIQUE`, `FULLTEXT`)
2. **Optimize Queries** (`EXPLAIN`, `ANALYZE`)
3. **Use Connection Pooling** (Reduce overhead)
4. **Enable Query Caching** (For repeated queries)
5. **Partition Large Tables** (Speeds up retrieval)

Summary

Component	Purpose
Client Layer	Handles connections, authentication, and requests
SQL Layer	Parses, optimizes, and executes queries
Storage Engine Layer	Manages physical storage of data
Buffer Pool	Caches frequently accessed data
Transaction Manager	Ensures ACID compliance
Replication & Clustering	Provides scalability and high availability

---

Final Thoughts

- **MySQL is modular** → Different storage engines for different use cases.
- **Performance depends on caching, indexing, and query optimization.**
- **High availability achieved through replication and clustering.**

Here are simple, real-life examples for each ACID property:

1. **Atomicity (All or Nothing)**
    - Example: **ATM Withdrawal**
        - You withdraw ₹500 from an ATM.
        - The bank system deducts ₹500 from your account.
        - If the cash dispenser fails, the system **rolls back** and doesn't deduct money.
        - Either **both actions happen or none** (no partial transaction).
2. **Consistency (Valid State Before & After)**
    - Example: **E-commerce Order**
        - You place an order for a phone costing ₹20,000.
        - The system checks if your account has ₹20,000.
        - If yes, the money is deducted, and the order is placed.
        - If not, the transaction **fails** and the balance remains unchanged.
3. **Isolation (No Interference Between Transactions)**
    - Example: **Train Ticket Booking**
        - Two people try to book the last seat at the same time.
        - One transaction **completes first** and gets the seat.
        - The second person's transaction **sees updated availability** and fails.
        - They don't interfere with each other.
4. **Durability (Permanent Changes)**
    - Example: **Bank Deposit**
        - You deposit ₹10,000 in your account and receive a confirmation.
        - Even if the system **crashes** right after, your money stays deposited.
        - Once committed, the transaction is **permanently stored**.

First Normal Form (1NF) - Full Example with Pros & Cons

1NF Rule:

- **No repeating groups** (No multiple values in a single cell).
- **Atomic values** (Each column should have indivisible values).
- **Unique columns** (Each column should store only one type of data).
- **Identify a primary key** (A unique identifier for each row).

Example (Before 1NF - Unnormalized Table)

OrderID	CustomerName	ItemsOrdered	Quantity
1	Alice	Pizza, Burger	2, 1
2	Bob	Sandwich, Coke	1, 1
3	Charlie	Pasta	1

Problems in this Table:

1. **Repeating values in "ItemsOrdered"** (Pizza, Burger in one cell).
2. **Multiple values in "Quantity"** (2, 1 in one cell).
3. **Difficult to search, update, or delete specific items**.

Example (After 1NF - Applying Normalization)

OrderID	CustomerName	ItemsOrdered	Quantity
1	Alice	Pizza	2
1	Alice	Burger	1
2	Bob	Sandwich	1
2	Bob	Coke	1
3	Charlie	Pasta	1

Pros of 1NF:

1. **Removes Data Redundancy** – Each value is stored separately.
2. **Easy Data Access & Searching** – You can search for "Pizza" without scanning multiple values in a single cell.
3. **Data Integrity** – No duplicate or inconsistent data.
4. **Better Sorting & Filtering** – Can easily filter by item or quantity.

Cons of 1NF:

1. **Increased Data Size** – More rows are created, which can increase storage requirements.

2. **More Complex Queries** – Requires **JOINs** to get the full order details.
3. **Slower Performance** – Since data is split across more rows, retrieval can take longer.

Second Normal Form (2NF) - Full Example with Pros & Cons

2NF Rules:

1. **Must be in 1NF** (No repeating groups, atomic values).
2. **No Partial Dependency** – Every **non-key column** must depend on the **whole primary key**, not just part of it.

Example (Before 2NF - Partial Dependency Issue)

Consider an **Order Management System** where a table stores **Orders, Customers, and Items**.

OrderID	CustomerID	CustomerName	ItemID	ItemName	Quantity
1	C01	Alice	I01	Pizza	2
1	C01	Alice	I02	Burger	1
2	C02	Bob	I03	Sandwich	1
2	C02	Bob	I04	Coke	1
3	C03	Charlie	I05	Pasta	1

Problems in this Table (Partial Dependency):

- **Composite Primary Key = (OrderID, ItemID)**
- **CustomerName depends only on CustomerID**, NOT on the full primary key.
- **ItemName depends only on ItemID**, NOT on the full primary key.

Since **CustomerName and ItemName do not depend on OrderID**, we have **partial dependencies**, which violate 2NF.

Example (After 2NF - Removing Partial Dependency)

We split the table into **three separate tables**:

Orders Table (Tracks Customer and OrderID)

OrderID	CustomerID
1	C01
2	C02
3	C03

Customers Table (Tracks Customer Info)

CustomerID	CustomerName
C01	Alice
C02	Bob
C03	Charlie

Order Details Table (Tracks Items in Each Order)

OrderID	ItemID	Quantity
1	I01	2
1	I02	1
2	I03	1
2	I04	1
3	I05	1

Items Table (Stores Item Details)

ItemID	ItemName
I01	Pizza
I02	Burger
I03	Sandwich
I04	Coke
I05	Pasta

Pros of 2NF:

1. **Eliminates Partial Dependencies** – Every **non-key column** depends **fully** on the primary key.
2. **Data Consistency** – "Alice" is stored only **once** in the **Customers Table**, avoiding duplication.
3. **Less Data Redundancy** – Instead of repeating customer and item names, we store them in separate tables.
4. **Easier Updates** – If Alice changes her name to "Alicia," we update it **once** in the Customers Table, not in multiple rows.

Cons of 2NF:

1. **More Tables, More JOINs** – Queries require **JOIN operations**, making them **complex and slower**.
2. **Increased Data Complexity** – Managing **multiple tables** is harder than one big table.
3. **More Foreign Keys** – More relationships mean **foreign keys** in tables, increasing database complexity.

Summary:

- 2NF **removes partial dependencies** by ensuring every **non-key column depends on the whole primary key**.
- This **reduces redundancy and improves consistency**, but **increases query complexity** due to multiple tables.

Third Normal Form (3NF) - Full Example with Pros & Cons

3NF Rules:

1. **Must be in 2NF** (No partial dependency).
2. **No Transitive Dependency** – A non-key column **should not depend on another non-key column**.
3. **Every non-key attribute should depend only on the primary key**.

Example (Before 3NF - Transitive Dependency Issue)

OrderID	CustomerID	CustomerName	City	ZipCode
1	C01	Alice	New York	10001
2	C02	Bob	Los Angeles	90001
3	C03	Charlie	Chicago	60601

Problems in this Table (Transitive Dependency):

- **CustomerName depends on CustomerID** (OK).
- **City depends on ZipCode, NOT on CustomerID** (Transitive Dependency).
- If the ZipCode of "New York" changes, we must update it in **multiple places** (Data inconsistency risk).

Example (After 3NF - Removing Transitive Dependency)

We split the table into **three separate tables**:

Orders Table (Tracks Order & Customer Info)

OrderID	CustomerID	ZipCode
1	C01	10001
2	C02	90001
3	C03	60601

Customers Table (Tracks Customer Info)

CustomerID	CustomerName
C01	Alice

CustomerID	CustomerName
C02	Bob
C03	Charlie

ZipCode Table (Stores City Details)

ZipCode	City
10001	New York
90001	Los Angeles
60601	Chicago

Pros of 3NF:

1. **No Data Redundancy** – "New York" appears **only once** in the ZipCode Table.
2. **Improved Data Integrity** – If a ZipCode changes, it is **updated in one place**.
3. **Efficient Storage** – Less duplicate data means **smaller database size**.
4. **Better Data Organization** – Each table has **one purpose**, making the structure cleaner.

Cons of 3NF:

1. **More Tables, More JOINs** – Queries require **JOINs** to retrieve complete information.
2. **Slightly Slower Performance** – More tables mean **more processing time** for fetching related data.
3. **More Complexity** – Maintaining multiple related tables requires **careful database design**.

Summary:

- 3NF **removes transitive dependencies** by ensuring that **every non-key column depends only on the primary key**.
- It **eliminates redundant data** and improves **database integrity**, but makes querying **more complex**.

Example (Before 4NF - Multi-Valued Dependency Issue)

Let’s consider a **Teacher-Subject-Classroom** relationship.

- A **Teacher** can teach **multiple Subjects**.
- A **Teacher** can teach in **multiple Classrooms**.
- But **Subjects and Classrooms are independent** (i.e., no direct relation between them).

TeacherID	TeacherName	Subject	Classroom
T01	Alice	Math	Room 101
T01	Alice	Math	Room 102

TeacherID	TeacherName	Subject	Classroom
T01	Alice	Science	Room 101
T01	Alice	Science	Room 102
T02	Bob	English	Room 201
T02	Bob	English	Room 202

Problems in this Table (Multi-Valued Dependency):

- The **Subjects and Classrooms are independent**, but they are **repeated unnecessarily**.
- If Alice teaches **Math and Science**, and she teaches in **Room 101 & 102**, we end up with **unnecessary duplicate rows**.
- More redundancy means **wasted storage and harder updates**.

Example (After 4NF - Removing Multi-Valued Dependency)

We split the table into **two separate tables**:

Teacher-Subject Table (Tracks subjects taught by each teacher)

TeacherID	Subject
T01	Math
T01	Science
T02	English

Teacher-Classroom Table (Tracks classrooms where each teacher teaches)

TeacherID	Classroom
T01	Room 101
T01	Room 102
T02	Room 201
T02	Room 202

Pros of 4NF:

1. **Eliminates Multi-Valued Dependencies** – No redundant combinations of subjects and classrooms.
2. **Reduces Data Redundancy** – No duplicate rows storing unrelated attributes.
3. **Improves Data Integrity** – Updates happen **only in one place**, preventing errors.
4. **More Efficient Storage** – Smaller tables save database space.

Cons of 4NF:

1. **More Tables, More JOINs** – Queries now require **JOINs** to fetch full teacher details.
2. **Increased Complexity** – More tables make the **database structure harder to understand**.

3. **Not Always Necessary** – In practical databases, **3NF or BCNF is often enough**.

Summary:

- **4NF eliminates multi-valued dependencies** by ensuring that **each independent multi-valued attribute is stored separately**.
- It **reduces redundancy** and **improves consistency**, but increases **query complexity**.

Fifth Normal Form (5NF) - Full Example with Pros & Cons

5NF Rules (Project-Join Normal Form - PJNF):

1. **Must be in 4NF** (No multi-valued dependencies).
2. **No Join Dependency** – A table should not be broken into smaller tables **that can be joined back without loss of data**.
3. **Ensures lossless decomposition** – Splitting data should not introduce **extra or missing records when re-joined**.

Example (Before 5NF - Join Dependency Issue)

Consider a **Consultant-Project-Client** relationship.

- A **Consultant** can work on **multiple Projects**.
- A **Consultant** can work with **multiple Clients**.
- A **Project can be assigned to multiple Clients**.

ConsultantID	ProjectID	ClientID
C01	P01	CL01
C01	P01	CL02
C02	P02	CL01
C02	P02	CL03

Problems in this Table (Join Dependency Issue):

- The **relationship is three-way** (**Consultant ↔ Project ↔ Client**).
- If we break it into **Consultant-Project** and **Project-Client**, we **lose the Consultant-Client relation**.
- Any **change in assignments requires multiple updates** across different projects and clients.

Example (After 5NF - Removing Join Dependency)

We break the table into **three separate tables**:

Consultant-Project Table

ConsultantID	ProjectID
C01	P01
C02	P02

Project-Client Table

ProjectID	ClientID
P01	CL01
P01	CL02
P02	CL01
P02	CL03

Consultant-Client Table (Only if needed to track direct consultant-client relationships)

ConsultantID	ClientID
C01	CL01
C01	CL02
C02	CL01
C02	CL03

Pros of 5NF:

1. **Removes Join Dependencies** – Ensures that data is stored in the most **efficient and meaningful** way.
2. **Eliminates Redundant Data** – Prevents **unnecessary duplication** across complex relationships.
3. **Improves Data Integrity** – Updates happen in **one place** instead of multiple tables.
4. **Supports Complex Many-to-Many Relationships** – Ideal for scenarios with multiple overlapping dependencies.

Cons of 5NF:

1. **More Tables, More JOINs** – Queries require **multiple joins** to reconstruct full data.
2. **Not Always Necessary** – In many real-world databases, **4NF is sufficient**.
3. **Difficult to Understand & Manage** – Requires a **deep understanding** of join dependencies.

Summary:

- **5NF removes join dependencies** and ensures that **data can be split into smaller tables without introducing redundancy or data loss**.

- It is **useful for highly complex many-to-many relationships**, but **increases database complexity**.

Sixth Normal Form (6NF) - Full Example with Pros & Cons

6NF Rules (Domain-Key Normal Form - DKNF):

1. **Must be in 5NF** (No join dependencies).
2. **Eliminates Temporal Dependencies** – The database should handle **time-variant** (historical) data efficiently.
3. **Every constraint is a function of the key, the whole key, and nothing but the key** (No non-trivial constraints).

Example (Before 6NF - Temporal Dependency Issue)

Consider an **Employee-Department Assignment** where employees change departments over time.

EmployeeID	Department	StartDate	EndDate
E01	HR	2022-01-01	2023-01-01
E01	Finance	2023-01-02	NULL
E02	IT	2021-06-15	2023-06-15
E02	Marketing	2023-06-16	NULL

Problems in this Table (Temporal Dependency):

- The **EmployeeID and Department form a composite key**, but **time (StartDate & EndDate) is also important**.
- If an employee **leaves a department and later rejoins**, the system may create **duplicate or inconsistent records**.
- Updating **historical data** is difficult and can cause **inconsistencies**.

Example (After 6NF - Eliminating Temporal Dependency)

We break the table into **separate tables for each time-dependent relationship**:

Employee-Department Table (Tracks which department an employee belongs to)

EmployeeID	Department
E01	HR
E01	Finance
E02	IT
E02	Marketing

Employee-Department Timeline Table (Tracks history of department changes)

EmployeeID	Department	StartDate	EndDate
E01	HR	2022-01-01	2023-01-01
E01	Finance	2023-01-02	NULL
E02	IT	2021-06-15	2023-06-15
E02	Marketing	2023-06-16	NULL

Pros of 6NF:

1. **Handles Time-Variant Data Efficiently** – Perfect for **historical records** (e.g., tracking salary changes, job roles, project assignments).
2. **Minimizes Data Redundancy** – Each table stores **only the necessary** relationships.
3. **Prevents Data Inconsistencies** – Changes in one table do not **affect unrelated records**.
4. **Supports Auditing & Compliance** – Useful for **financial, legal, and healthcare databases** that require **accurate historical tracking**.

Cons of 6NF:

1. **Extreme Complexity** – Leads to **many small tables**, making **querying more difficult**.
2. **High Use of JOINs** – Requires multiple **JOIN operations** to retrieve complete information.
3. **Not Always Necessary** – Many practical databases **don't need 6NF**, as **5NF is usually sufficient**.

Summary:

- **6NF is mainly useful for temporal databases**, ensuring that **time-based changes are stored properly without redundancy or inconsistencies**.
- It **improves historical tracking** but makes queries **more complex**.

MySQL DDL with Sample Data

Let's assume we are managing a company's **employees** and **departments** database.

Command	Purpose
`CREATE TABLE`	Creates a new table
`ALTER TABLE`	Modifies an existing table
`DROP TABLE`	Deletes a table completely
`TRUNCATE TABLE`	Deletes all rows but keeps table structure

`CREATE INDEX`	Creates an index for faster queries
`CREATE CONSTRAINTS`	Defines rules (Primary Key, Foreign Key, Default)

---

1. Create Tables with Constraints

```
sql
CopyEdit
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL UNIQUE
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department_id INT,
    salary DECIMAL(10,2) DEFAULT 50000.00,
    email VARCHAR(100) UNIQUE,
    FOREIGN KEY (department_id) REFERENCES departments(dept_id) ON DELETE
CASCADE
);
```

What this does:

- `departments` table has a **Primary Key** (`dept_id`) and ensures **unique department names**.
- `employees` table has a **Foreign Key** linking `department_id` to `departments(dept_id)`, ensuring referential integrity.
- `email` must be **unique**.
- `salary` has a **default value** of `50000.00` if not provided.
- If a **department is deleted**, all employees in that department are also deleted (`ON DELETE CASCADE`).

---

2. Insert Sample Data

```
sql
CopyEdit
INSERT INTO departments (dept_id, dept_name) VALUES
(1, 'HR'),
(2, 'Engineering'),
(3, 'Sales');

INSERT INTO employees (emp_id, name, department_id, salary, email) VALUES
(101, 'Alice Johnson', 1, 60000.00, 'alice@example.com'),
(102, 'Bob Smith', 2, 75000.00, 'bob@example.com'),
(103, 'Charlie Brown', 3, 50000.00, 'charlie@example.com');
```

---

3. Load Data from CSV

Sample `employees.csv` File

```
graphql
CopyEdit
emp_id,name,department_id,salary,email
104,David Lee,2,72000.00,david@example.com
105,Ella Adams,1,58000.00,ella@example.com
106,Frank White,3,51000.00,frank@example.com
```

Enable CSV Import in MySQL

Ensure `local_infile` is enabled:

```
sql
CopyEdit
SET GLOBAL local_infile = 1;
```

Import CSV into MySQL

```
sql
CopyEdit
LOAD DATA INFILE '/path/to/employees.csv'
INTO TABLE employees
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

What this does:

- Loads data from `employees.csv` into the `employees` table.
- Ignores the **header row** (`IGNORE 1 ROWS`).

---

4. Load Parquet Data (Workaround)

MySQL **does not** support Parquet natively. Convert Parquet to CSV using `parquet-tools`:

Convert Parquet to CSV (Linux/Mac)

```
bash
CopyEdit
parquet-tools csv employees.parquet > employees.csv
```

Then, use `LOAD DATA INFILE` as shown above.

Index Types

Index Type	Description	Example
Regular Index	Speeds up lookups on a column	`CREATE INDEX idx_name ON employees(name);`
Unique Index	Ensures values are unique	`CREATE UNIQUE INDEX idx_email ON employees(email);`
Composite Index	Index on multiple columns	`CREATE INDEX idx_emp_dept ON employees(name, department_id);`
Full-Text Index	Optimized for searching text data	`CREATE FULLTEXT INDEX idx_fulltext_name ON employees(name);`
Primary Key Index	Automatically created for `PRIMARY KEY`	`PRIMARY KEY(emp_id);`

Indexing in MySQL

Let's assume we are managing an **employees database** with thousands of records. We'll create different types of indexes to improve performance.

---

1. Creating a Table with Sample Data

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,   -- Automatically Indexed
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,  -- Automatically Indexed
    department VARCHAR(50),
    salary DECIMAL(10,2),
    bio TEXT  -- For full-text search
);
```

Indexes automatically created for:

- `PRIMARY KEY(emp_id)`: Unique & fast lookups.
- `UNIQUE(email)`: Prevents duplicate emails.

---

2. Creating a Regular Index (Faster Search)

Problem: Searching by `name` is slow.

Solution: Create an index on `name`.

```
sql
CopyEdit
CREATE INDEX idx_name ON employees(name);
```

Speeds up queries like:

```
sql
CopyEdit
SELECT * FROM employees WHERE name = 'Alice Johnson';
```

Without an index: MySQL scans every row.
With an index: MySQL directly finds matches.

---

3. Creating a Composite Index (Multi-Column Search)

Problem: Queries filter by `name` and `department`.

Solution: Create a multi-column index.

```
sql
CopyEdit
CREATE INDEX idx_name_department ON employees(name, department);
```

Optimizes queries like:

```
sql
CopyEdit
SELECT * FROM employees WHERE name = 'Alice Johnson' AND department = 'HR';
```

Without index: MySQL checks each row.
With index: Faster lookups when filtering by **both columns**.

---

4. Creating a Unique Index (Prevent Duplicates)

```
sql
CopyEdit
CREATE UNIQUE INDEX idx_email ON employees(email);
```

Ensures **no two employees have the same email**.

```
sql
CopyEdit
INSERT INTO employees (emp_id, name, email, department, salary)
VALUES (101, 'John Doe', 'john@example.com', 'IT', 70000.00);
```

Trying to insert a duplicate email:

```
sql
CopyEdit
INSERT INTO employees (emp_id, name, email, department, salary)
VALUES (102, 'Jane Doe', 'john@example.com', 'HR', 75000.00);
```

Error: Duplicate entry 'john@example.com' for key `idx_email`

---

5. Full-Text Index for Fast Text Searching

Problem: Searching `bio` using `LIKE` is slow.

Solution: Use a Full-Text Index.

```
sql
CopyEdit
CREATE FULLTEXT INDEX idx_bio ON employees(bio);
```

Fast text search on large text fields:

```
sql
CopyEdit
SELECT * FROM employees WHERE MATCH(bio) AGAINST('Python Developer');
```

Much faster than:

```
sql
CopyEdit
SELECT * FROM employees WHERE bio LIKE '%Python Developer%';
```

---

6. Dropping an Index

Remove an index if it's no longer needed:

```
sql
CopyEdit
DROP INDEX idx_name ON employees;
```

Only removes the index, table remains unchanged.

---

7. Checking Existing Indexes

To **see all indexes** on a table:

```
sql
CopyEdit
SHOW INDEX FROM employees;
```

Performance Comparison

Query	Without Index	With Index
`SELECT * FROM employees WHERE name = 'Alice'`	Slow (Full table scan)	Fast (Direct lookup)
`SELECT * FROM employees WHERE name = 'Alice' AND department = 'HR'`	Very slow	Faster (Composite Index)
`SELECT * FROM employees WHERE MATCH(bio) AGAINST('Python Developer')`	Slow	Very fast (Full-Text Index)

When Should You Use Indexes?

Use indexes when:

- Frequently searching/filtering columns (`WHERE`).
- Sorting results (`ORDER BY`).
- Joining tables (`JOIN`).

Avoid excessive indexes because:

- **Slows down `INSERT`, `UPDATE`, `DELETE` operations.**
- **Consumes extra storage.**

Testing Index Performance in MySQL

Now, let's test **how indexes improve query speed** by comparing **with and without indexing**.

---

1. Setup: Create a Large Employees Table

We'll create a **big dataset** (100,000+ rows) to measure performance.

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    department VARCHAR(50),
    salary DECIMAL(10,2),
    bio TEXT
);
```

Primary Key (`emp_id`) & Unique Index (`email`) are auto-created.

---

2. Insert 100,000 Sample Rows (Fake Data Generation)

If you're using MySQL **8+**, use `generate_series()` (or use Python script).

```
sql
CopyEdit
INSERT INTO employees (name, email, department, salary, bio)
SELECT
    CONCAT('Employee', FLOOR(RAND() * 100000)),
    CONCAT('user', FLOOR(RAND() * 100000), '@example.com'),
    ELT(FLOOR(1 + (RAND() * 3)), 'HR', 'Engineering', 'Sales'),
    ROUND(RAND() * 50000 + 50000, 2),
    'This is a sample bio for the employee'
FROM
    (SELECT * FROM information_schema.tables LIMIT 100000) AS t;
```

100,000 records added with random names, emails, and salaries.

---

3. Query Without an Index (Slow Performance)

Search employees by name (No Index)

```
sql
CopyEdit
```

```
SELECT * FROM employees WHERE name = 'Employee5000';
```

Slow Query: MySQL scans **ALL** 100,000 rows.

---

4. Add an Index on `name` and Test Again

```
sql
CopyEdit
CREATE INDEX idx_name ON employees(name);
```

Now re-run the query:

```
sql
CopyEdit
SELECT * FROM employees WHERE name = 'Employee5000';
```

Much faster! MySQL uses the index instead of scanning all rows.

---

5. Test a Multi-Column Index

Without an Index (Slow)

```
sql
CopyEdit
SELECT * FROM employees WHERE name = 'Employee5000' AND department = 'HR';
```

Slow Full Table Scan (100,000+ rows).

Add a Composite Index

```
sql
CopyEdit
CREATE INDEX idx_name_dept ON employees(name, department);
```

Now re-run the query:

```
sql
CopyEdit
SELECT * FROM employees WHERE name = 'Employee5000' AND department = 'HR';
```

Super fast! MySQL now uses the composite index.

---

6. How to Check Index Usage?

Run `EXPLAIN` before a query to see if MySQL is **using indexes**.

```
sql
CopyEdit
EXPLAIN SELECT * FROM employees WHERE name = 'Employee5000';
```

Look for:

- **Using index** → Index is working
- **Using where** → Full table scan (slow)

---

7. Remove an Index (If Needed)

```
sql
CopyEdit
DROP INDEX idx_name ON employees;
```

Deletes the index, but keeps the table and data.

Summary

Query Type	Without Index	With Index
`WHERE name = 'Employee5000'`	Slow (Full Table Scan)	Fast (Indexed Lookup)
`WHERE name = 'Employee5000' AND department = 'HR'`	Very Slow	Fast (Composite Index)
`WHERE MATCH(bio) AGAINST('developer')`	Slow	Fast (Full-Text Index)

Testing Full-Text Search Performance in MySQL

Now, let's test how **Full-Text Indexing** can dramatically improve search speed for **large text fields** (e.g., resumes, product descriptions, articles).

---

1. Create a Table with Large Text Data

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    department VARCHAR(50),
    salary DECIMAL(10,2),
    bio TEXT
);
```

The `bio` column will store **long text data** (e.g., employee descriptions).

---

2. Insert 100,000 Sample Employees (Large Text Data)

```
sql
CopyEdit
INSERT INTO employees (name, email, department, salary, bio)
SELECT
    CONCAT('Employee', FLOOR(RAND() * 100000)),
    CONCAT('user', FLOOR(RAND() * 100000), '@example.com'),
    ELT(FLOOR(1 + (RAND() * 3)), 'HR', 'Engineering', 'Sales'),
    ROUND(RAND() * 50000 + 50000, 2),
    'This employee is an experienced Python and SQL developer specializing in
database management and web applications.'
FROM
    (SELECT * FROM information_schema.tables LIMIT 100000) AS t;
```

100,000 rows added with random text in `bio`.

---

3. Searching Without an Index (Slow Performance)

Querying Using `LIKE` (Inefficient)

```
sql
CopyEdit
SELECT * FROM employees WHERE bio LIKE '%Python developer%';
```

Problem: MySQL does a **full table scan** (slow for 100,000+ rows).

---

4. Create a Full-Text Index on `bio`

```
sql
CopyEdit
CREATE FULLTEXT INDEX idx_bio ON employees(bio);
```

Now MySQL can search text efficiently.

---

5. Fast Full-Text Search Query

```
sql
CopyEdit
SELECT * FROM employees WHERE MATCH(bio) AGAINST('Python developer');
```

Super Fast! MySQL uses the full-text index for searching.

---

6. Advanced Full-Text Search (Ranking & Relevance)

Search for multiple keywords

```
sql
```

```
CopyEdit
SELECT * FROM employees
WHERE MATCH(bio) AGAINST('Python SQL' IN NATURAL LANGUAGE MODE);
```

Finds employees skilled in Python OR SQL.

Prioritize Exact Matches

```
sql
CopyEdit
SELECT * FROM employees
WHERE MATCH(bio) AGAINST('+Python +SQL' IN BOOLEAN MODE);
```

Finds employees skilled in BOTH Python AND SQL.

Exclude Keywords

```
sql
CopyEdit
SELECT * FROM employees
WHERE MATCH(bio) AGAINST('+Python -SQL' IN BOOLEAN MODE);
```

Finds employees skilled in Python but NOT SQL.

---

7. Checking Index Usage with `EXPLAIN`

```
sql
CopyEdit
EXPLAIN SELECT * FROM employees WHERE MATCH(bio) AGAINST('Python developer');
```

Look for:

- **Using index** → Index is working
- **Full table scan** → Index not being used

---

8. Removing a Full-Text Index (If Needed)

```
sql
CopyEdit
DROP INDEX idx_bio ON employees;
```

Removes the full-text index but **keeps the table and data**.

Performance Comparison

Query	Without Full-Text Index	With Full-Text Index
`WHERE bio LIKE '%Python%'`	Slow (Full Table Scan)	Fast (Optimized Search)
`WHERE MATCH(bio) AGAINST('Python')`	Not Supported	Works Efficiently

Query	Without Full-Text Index	With Full-Text Index
`WHERE MATCH(bio) AGAINST('Python SQL' IN BOOLEAN MODE)`	Not Supported	Ranked Results

Real-World Full-Text Search Performance Testing

Now, let's **test full-text search performance** on a **large dataset** and compare it with traditional `LIKE` queries.

---

1. Create a Table with Large Text Data

We'll store **employee profiles** with skills and experiences.

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    department VARCHAR(50),
    salary DECIMAL(10,2),
    bio TEXT -- Large text field for skills & experience
);
```

The `bio` column will store resumes or skill descriptions.

---

2. Insert 100,000 Sample Employees with Text Data

We'll add **realistic job profiles** for performance testing.

```
sql
CopyEdit
INSERT INTO employees (name, email, department, salary, bio)
SELECT
    CONCAT('Employee', FLOOR(RAND() * 100000)),
    CONCAT('user', FLOOR(RAND() * 100000), '@example.com'),
    ELT(FLOOR(1 + (RAND() * 3)), 'HR', 'Engineering', 'Sales'),
    ROUND(RAND() * 50000 + 50000, 2),
    CASE
        WHEN RAND() < 0.33 THEN 'Experienced in Python, MySQL, and Data
Analysis.'
        WHEN RAND() < 0.66 THEN 'Expert in Java, Spring Boot, and microservices
architecture.'
        ELSE 'Skilled in JavaScript, React, and front-end development.'
    END
FROM
    (SELECT * FROM information_schema.tables LIMIT 100000) AS t;
```

100,000 employees added with realistic job descriptions.

---

3. Test Search Without Index (Slow Performance)

Query Using `LIKE` (Slow)

```
sql
CopyEdit
SELECT * FROM employees WHERE bio LIKE '%Python%';
```

> **Problem:** MySQL **scans all 100,000 rows** → Very slow!

---

4. Add a Full-Text Index on `bio`

```
sql
CopyEdit
CREATE FULLTEXT INDEX idx_bio ON employees(bio);
```

> **Now MySQL can search text much faster.**

---

5. Test Full-Text Search Performance

Search for “Python”

```
sql
CopyEdit
SELECT * FROM employees WHERE MATCH(bio) AGAINST('Python');
```

> **Super Fast!** MySQL **uses the full-text index** instead of scanning all rows.

---

6. Advanced Full-Text Search Queries

Search for Multiple Keywords

```
sql
CopyEdit
SELECT * FROM employees
WHERE MATCH(bio) AGAINST('Python SQL' IN NATURAL LANGUAGE MODE);
```

> **Finds employees with Python OR SQL skills.**

Prioritize Exact Matches

```
sql
CopyEdit
SELECT * FROM employees
WHERE MATCH(bio) AGAINST('+Python +SQL' IN BOOLEAN MODE);
```

> **Finds employees with both Python AND SQL.**

Exclude Keywords

```
sql
CopyEdit
SELECT * FROM employees
WHERE MATCH(bio) AGAINST('+Python -Java' IN BOOLEAN MODE);
```

Finds employees with Python but NOT Java experience.

---

7. Compare Performance: `LIKE` vs. `FULLTEXT`

Query	Without Index (`LIKE` Query)	With Full-Text Index (`MATCH() AGAINST()`)
`WHERE bio LIKE '%Python%'`	Slow (Full Table Scan)	Fast (Index Lookup)
`WHERE MATCH(bio) AGAINST('Python')`	Not Supported	Instant Results
`WHERE MATCH(bio) AGAINST('Python SQL' IN BOOLEAN MODE)`	Not Supported	Ranked Results

---

8. Verify Index Usage with `EXPLAIN`

```
sql
CopyEdit
EXPLAIN SELECT * FROM employees WHERE MATCH(bio) AGAINST('Python');
```

Look for:

- **Using index** → Index is working
- **Full table scan** → Index not being used

---

9. Remove a Full-Text Index (If Needed)

```
sql
CopyEdit
DROP INDEX idx_bio ON employees;
```

Removes the index but keeps the table data.

Key Takeaways

- **`LIKE '%word%'` is slow** → MySQL scans all rows
- **`FULLTEXT` index makes text searches much faster**
- **Advanced search options** (Boolean mode, Natural language mode).

Foreign Keys in MySQL: What They Are and How to Use Them

A **foreign key** is a column (or a set of columns) in one table that establishes a link to the primary key in another table. It helps maintain data integrity by ensuring that the values in the foreign key column match values in the referenced primary key or unique key column of another table.

---

1. Basics of Foreign Key Constraints

A **foreign key** ensures that a record in one table corresponds to a valid record in another table. This is typically used to create a **relationship** between two tables, like:

- **One-to-many** (e.g., one department can have many employees)
- **Many-to-many** (e.g., students can enroll in many courses, and each course can have many students)

Foreign Key Example

Suppose we have two tables: `employees` and `departments`.

```
sql
CopyEdit
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

- **`employees`** has a **foreign key (`dept_id`)** that links to **`departments(dept_id)`**.
- This ensures that every **employee** must be assigned to an existing **department**.

---

2. Different Foreign Key Actions (Not Just for Cascade Delete)

Foreign keys are not only used for **cascade delete** but also to define how actions (insert, update, delete) in the **parent table** affect the **child table**. Here are the main types of actions you can set on foreign keys:

---

a. `ON DELETE CASCADE`

- When a record in the **parent table** is deleted, the **related records in the child table** are **automatically deleted**.
- **Use case**: Deleting an employee should also delete the employee's associated records (e.g., payroll, task assignments).

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE CASCADE
);
```

- **Benefit**: Automatically maintains referential integrity (no orphaned records in the child table).

---

b. `ON DELETE SET NULL`

- When a record in the **parent table** is deleted, the **foreign key column in the child table** is set to **NULL**.
- **Use case**: If a department is deleted, employees should be moved to a `NULL` state (i.e., without a department).

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE SET NULL
);
```

- **Benefit**: Useful when you don't want to delete the child records but want to indicate that they no longer belong to the parent.

---

c. `ON DELETE RESTRICT`

- Prevents deletion of a record in the **parent table** if there are related records in the **child table**.
- **Use case**: A department can't be deleted if there are still employees assigned to it.

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE RESTRICT
);
```

- **Benefit**: Ensures that data integrity is not violated by accidental deletions.

---

d. `ON DELETE NO ACTION`

- **Similar to `RESTRICT`**, but MySQL checks for referential integrity only when the statement is executed (not during insert or delete).
- **Use case**: When you want to enforce data integrity only after certain operations.

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE NO ACTION
);
```

- **Benefit**: Provides more flexible control over how changes to the parent table affect the child table.

---

e. `ON UPDATE CASCADE`

- When the value of the **primary key** in the **parent table** is updated, the corresponding **foreign key values** in the **child table** are **automatically updated**.
- **Use case**: If a department's `dept_id` is updated (e.g., a renumbering of departments), the change should automatically propagate to all employees in that department.

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON UPDATE CASCADE
);
```

- **Benefit**: Ensures consistency across related records when the parent record's key changes.

---

f. `ON UPDATE SET NULL`

- When the value of the **primary key** in the **parent table** is updated, the **foreign key column** in the **child table** is set to `NULL`.
- **Use case**: If a department's `dept_id` is changed, employees who were previously assigned to the old `dept_id` should be moved to a `NULL` state (i.e., without a department).

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON UPDATE SET NULL
);
```

---

3. When Should You Use Foreign Keys?

- **Data Integrity**: Foreign keys help maintain the relationship between related tables and ensure that data is consistent and valid.
- **Prevent Orphaned Records**: They prevent **orphaned child records** (i.e., child records that don't have a corresponding parent record).
- **Enforce Relationships**: Foreign keys enforce **one-to-many** and **many-to-many** relationships between tables.

4. Benefits of Foreign Keys:

- **Enforce Referential Integrity**: Ensures data consistency and integrity by making sure the relationships between tables are respected.

- **Automatic Handling of Dependencies**: With actions like `CASCADE` and `SET NULL`, foreign keys handle related data when records are modified or deleted.
- **Data Cleanup**: In situations like `ON DELETE CASCADE`, it allows for automatic data removal, keeping your database clean without needing extra cleanup logic.

---

5. Caveats of Using Foreign Keys

- **Performance Impact**: Foreign keys can impact the performance of `INSERT`, `UPDATE`, and `DELETE` operations because MySQL must check the constraints.
- **Requires Storage Engines**: Foreign keys only work with certain storage engines, like `InnoDB` (not `MyISAM`).
- **Complex Relationships**: They can make database design more complex, especially in cases with circular references (tables that reference each other).

Example with Multiple Foreign Key Actions:

Let's say we have a **projects** table that references the **employees** and **departments** tables.

```
sql
CopyEdit
CREATE TABLE projects (
    project_id INT PRIMARY KEY,
    project_name VARCHAR(100),
    emp_id INT,
    dept_id INT,
    FOREIGN KEY (emp_id)
        REFERENCES employees(emp_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE RESTRICT
);
```

In this case:

- **If an employee is deleted**, their `emp_id` in the `projects` table will be set to `NULL` (to keep the project record intact).
- **If a department is deleted**, the associated projects can't be deleted because of the `RESTRICT` action.

Summary of Foreign Key Actions:

Action	Description
CASCADE	Automatically delete or update child records.
SET NULL	Set the foreign key to `NULL` when the parent is deleted/updated.
RESTRICT	Prevent deletion or update if child records exist.

Action	Description
NO ACTION	Similar to `RESTRICT`, but checks after the operation.
CASCADE (UPDATE)	Automatically update child records if the parent is updated.

When to Use Foreign Keys?

- Use them when you need to ensure **data integrity**.
- Use actions like **CASCADE** or **SET NULL** when you need to handle related data automatically.
- Use **RESTRICT** when you want to prevent changes to the parent table that would leave orphaned child records.

Let's walk through a **real-world example** where we set up **foreign key relationships** with different actions like `CASCADE`, `SET NULL`, and `RESTRICT`.

Scenario: Company Database

We have a **Company** database with the following tables:

- **Departments**: Contains department details.
- **Employees**: Contains employee details, with each employee linked to a department.
- **Projects**: Contains project details, with each project linked to both an employee and a department.

We'll define the **foreign key relationships** with different actions and show how they work.

---

1. Create the `departments` Table

```
sql
CopyEdit
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL
);
```

Insert Sample Departments

```
sql
CopyEdit
INSERT INTO departments (dept_id, dept_name)
VALUES
(1, 'HR'),
(2, 'Engineering'),
(3, 'Sales');
```

---

2. Create the `employees` Table with a Foreign Key to `departments`

This table will have a **foreign key constraint** linking each employee to a department.

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Explanation:

- **ON DELETE SET NULL**: If a department is deleted, the `dept_id` in the `employees` table will be set to `NULL` (meaning they no longer belong to a department).
- **ON UPDATE CASCADE**: If the department's `dept_id` is updated, the corresponding `dept_id` in the `employees` table will also be updated.

---

3. Insert Sample Employees

```
sql
CopyEdit
INSERT INTO employees (emp_id, emp_name, dept_id)
VALUES
(1, 'Alice', 1),
(2, 'Bob', 2),
(3, 'Charlie', 3);
```

---

4. Create the `projects` Table with Foreign Keys to `employees` and `departments`

This table will contain project assignments, with foreign keys pointing to both the `employees` and `departments` tables.

```
sql
CopyEdit
CREATE TABLE projects (
    project_id INT PRIMARY KEY,
    project_name VARCHAR(100),
    emp_id INT,
    dept_id INT,
    FOREIGN KEY (emp_id)
        REFERENCES employees(emp_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
    FOREIGN KEY (dept_id)
```

```
        REFERENCES departments(dept_id)
        ON DELETE RESTRICT
);
```

Explanation:

- **Foreign Key 1 (`emp_id`)**:
  - **ON DELETE SET NULL**: If an employee is deleted, the project assignment for that employee is set to `NULL` (the project still exists but without an assigned employee).
  - **ON UPDATE CASCADE**: If an employee's `emp_id` is updated, the corresponding `emp_id` in the `projects` table will also update.
- **Foreign Key 2 (`dept_id`)**:
  - **ON DELETE RESTRICT**: If a department is deleted, the deletion is **restricted** if there are any projects linked to that department. This ensures that departments with active projects can't be deleted without first handling the projects.

---

5. Insert Sample Projects

```
sql
CopyEdit
INSERT INTO projects (project_id, project_name, emp_id, dept_id)
VALUES
(1, 'Project Alpha', 1, 2),  -- Alice working in Engineering
(2, 'Project Beta', 2, 2),   -- Bob working in Engineering
(3, 'Project Gamma', 3, 3);  -- Charlie working in Sales
```

---

6. Demonstrating the Foreign Key Actions

Scenario 1: Deleting a Department with `SET NULL` on Employees

Let's delete a department (e.g., `HR`) and see how the employees linked to that department are affected.

```
sql
CopyEdit
DELETE FROM departments WHERE dept_id = 1;  -- Deleting HR department
```

Result:

- **Employee Alice** (who was in the `HR` department) will now have her `dept_id` set to `NULL` in the `employees` table because of the `ON DELETE SET NULL` action.

Scenario 2: Updating a Department's `dept_id`

Let’s update the `dept_id` of the `Engineering` department.

```
sql
CopyEdit
UPDATE departments SET dept_id = 10 WHERE dept_id = 2;  -- Renumbering
Engineering to 10
```

Result:

- **Employees Bob and Alice** will have their `dept_id` updated to `10` in the `employees` table because of the `ON UPDATE CASCADE` action.

Scenario 3: Deleting a Department with `RESTRICT` on Projects

Let's try deleting a department (`Engineering`) that still has associated projects.

```
sql
CopyEdit
DELETE FROM departments WHERE dept_id = 2;  -- Trying to delete Engineering
```

Result:

- This query will **fail** because there are **projects** that depend on the `Engineering` department. The `ON DELETE RESTRICT` action prevents the deletion of a department if there are still related projects in the `projects` table.

---

7. Foreign Key Error Scenarios

Let’s look at the types of errors you might encounter if foreign key constraints are violated.

Scenario 1: Trying to Insert an Employee Without a Valid Department

```
sql
CopyEdit
INSERT INTO employees (emp_id, emp_name, dept_id)
VALUES (4, 'David', 99);  -- Dept_id 99 does not exist
```

Result:

- **Error**: MySQL will **reject** the insert because `dept_id = 99` does not exist in the `departments` table (foreign key violation).

Scenario 2: Trying to Delete an Employee Who Has Projects Assigned

```
sql
CopyEdit
DELETE FROM employees WHERE emp_id = 1;  -- Alice has a project assigned
```

Result:

- **Error**: If we didn't have `ON DELETE SET NULL` or `CASCADE`, deleting Alice would fail because there are **projects** that still reference her. It’s **protected** by the foreign key.

---

8. Verifying the Foreign Key Constraints

You can check the foreign key constraints using `SHOW CREATE TABLE` or by querying the `information_schema`.

```
sql
CopyEdit
SHOW CREATE TABLE employees;
SHOW CREATE TABLE projects;
```

This will give you details of the **foreign key actions** defined on those tables.

---

9. Removing Foreign Key Constraints

If you ever need to **remove a foreign key constraint**, you can use `ALTER TABLE` to drop the index.

```
sql
CopyEdit
ALTER TABLE projects DROP FOREIGN KEY fk_emp_id;
```

Replace `fk_emp_id` with the actual foreign key name, which you can find using `SHOW CREATE TABLE`.

Summary of Foreign Key Actions:

Action	Effect
ON DELETE CASCADE	Automatically delete related child records.
ON DELETE SET NULL	Set the foreign key to NULL when the parent is deleted.
ON DELETE RESTRICT	Prevent deletion of the parent if there are dependent child records.
ON UPDATE CASCADE	Automatically update child foreign key when the parent is updated.

Data Manipulation Language (DML)

DML consists of SQL commands used to manipulate data in a database. The main DML statements are **INSERT, UPDATE, DELETE, and SELECT** (though SELECT is often categorized under DQL—Data Query Language).

---

1. Adding Data with the INSERT Statement

The `INSERT` statement is used to add new rows to a table. Basic syntax:

```
sql
CopyEdit
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

---

2. INSERT Statement with Columns, NULL, DEFAULT, Identity, and Constraints

- **Specifying Columns:** If you specify column names, only those columns need values.
- **NULL Values:** If a column allows NULL values, you can insert NULL explicitly:

```
sql
CopyEdit
INSERT INTO employees (id, name, salary) VALUES (1, NULL, 50000);
```

- **DEFAULT Values:** Some columns may have a `DEFAULT` value.

```
sql
CopyEdit
INSERT INTO employees (id, name) VALUES (2, 'John');  -- Salary will take
the default value
```

- **Identity Columns:** If a column is an **IDENTITY** column (auto-incrementing), omit it from `INSERT`:

```
sql
CopyEdit
INSERT INTO employees (name, salary) VALUES ('Alice', 60000);
```

- **Constraints:** Constraints like `NOT NULL`, `UNIQUE`, `CHECK`, and `FOREIGN KEY` ensure data integrity.

---

3. Modifying Data Using the UPDATE Statement

The `UPDATE` statement modifies existing records in a table:

```
sql
CopyEdit
UPDATE employees
SET salary = 70000
WHERE id = 3;
```

Always use a **WHERE** clause to prevent unintended updates.

---

4. Modifying Data Using UPDATE with Constraints and JOINs

- **Using Constraints:** The update will fail if it violates a constraint (e.g., unique key, foreign key).
- **Using JOINs:** Update data based on another table.

```
sql
CopyEdit
UPDATE e
SET e.salary = d.average_salary
FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE e.salary < d.average_salary;
```

---

5. Removing Data Using DELETE, JOINs, and Cascade Deletion

- **Basic DELETE:**

```
sql
CopyEdit
```

```
DELETE FROM employees WHERE id = 4;
```

- **Using JOINs to DELETE:**

```
sql
CopyEdit
DELETE e
FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE d.name = 'HR';
```

- **Cascade Deletion:** If a **foreign key** is set with `ON DELETE CASCADE`, deleting a parent row will delete child rows automatically.

---

6. Clearing Tables: TRUNCATE vs DELETE

Feature	DELETE	TRUNCATE
Removes Data	Row by row	All rows at once
WHERE Clause	Allowed	Not allowed
Performance	Slower (logged for rollback)	Faster (minimal logging)
Resets Identity Column	No	Yes

- **DELETE**:

```
sql
CopyEdit
DELETE FROM employees WHERE department_id = 3;
```

- **TRUNCATE**:

```
sql
CopyEdit
TRUNCATE TABLE employees;
```

1. Adding Data with the INSERT Statement

Example: Basic `INSERT`

```
sql
CopyEdit
INSERT INTO employees (id, name, department, salary)
VALUES (1, 'Alice', 'HR', 50000);
```

---

2. INSERT with NULL, DEFAULT, Identity, and Constraints

Example: Inserting NULL

```
sql
CopyEdit
INSERT INTO employees (id, name, department, salary)
VALUES (2, 'Bob', 'Finance', NULL);
```

Example: Using DEFAULT (Assuming `DEFAULT salary = 40000`)

```
sql
CopyEdit
INSERT INTO employees (id, name, department)
VALUES (3, 'Charlie', 'IT');
```

Example: Identity Column (Auto-increment)

```
sql
CopyEdit
INSERT INTO employees (name, department, salary)
VALUES ('David', 'HR', 55000);
```

---

3. Modifying Data Using the UPDATE Statement

Example: Updating a Single Record

```
sql
CopyEdit
UPDATE employees
SET salary = 60000
WHERE id = 1;
```

---

4. Modifying Data Using UPDATE with Constraints and JOINs

Example: Updating with a JOIN

```
sql
CopyEdit
UPDATE e
SET e.salary = d.average_salary
FROM employees e
JOIN departments d ON e.department = d.name
WHERE e.salary < d.average_salary;
```

---

5. Removing Data Using DELETE, JOINs, and Cascade Deletion

Example: Basic DELETE

```
sql
CopyEdit
DELETE FROM employees WHERE id = 2;
```

Example: DELETE Using JOIN

```
sql
CopyEdit
DELETE e
FROM employees e
JOIN departments d ON e.department = d.name
WHERE d.name = 'HR';
```

Example: Cascade DELETE (assuming `ON DELETE CASCADE` is set)

```
sql
CopyEdit
DELETE FROM departments WHERE name = 'Finance';
```

---

6. Clearing Tables: TRUNCATE vs DELETE

Example: Using DELETE (Removes rows but keeps structure)

```
sql
CopyEdit
DELETE FROM employees;
```

Example: Using TRUNCATE (Faster, resets auto-increment)

```
sql
CopyEdit
TRUNCATE TABLE employees;
```

Creating the `employees` Table

```
sql
CopyEdit
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    department_id INT,
    salary DECIMAL(10,2) DEFAULT 40000,
    hire_date DATE NOT NULL,
    FOREIGN KEY (department_id) REFERENCES departments(id) ON DELETE CASCADE
);
```

Explanation:

- `id INT AUTO_INCREMENT PRIMARY KEY` → Unique identifier, auto-increments for each new employee.
- `name VARCHAR(50) NOT NULL` → Stores employee names, cannot be NULL.
- `department_id INT` → Foreign key referencing the `departments` table.
- `salary DECIMAL(10,2) DEFAULT 40000` → Stores salary with 2 decimal places, defaulting to **40,000** if not specified.
- `hire_date DATE NOT NULL` → Stores the hiring date of employees.
- `FOREIGN KEY (department_id) REFERENCES departments(id) ON DELETE CASCADE` → Ensures that deleting a department removes all related employees.

---

2. Creating the `departments` Table

```
sql
CopyEdit
CREATE TABLE departments (
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    name VARCHAR(50) UNIQUE NOT NULL,
    budget DECIMAL(12,2) CHECK (budget >= 10000)
);
```

Explanation:

- `id INT AUTO_INCREMENT PRIMARY KEY` → Unique department identifier.
- `name VARCHAR(50) UNIQUE NOT NULL` → Ensures each department name is unique and cannot be NULL.
- `budget DECIMAL(12,2) CHECK (budget >= 10000)` → Enforces a minimum budget of **10,000**.

---

3. Creating the `projects` Table

```
sql
CopyEdit
CREATE TABLE projects (
    id INT AUTO_INCREMENT PRIMARY KEY,
    project_name VARCHAR(100) NOT NULL,
    department_id INT,
    start_date DATE,
    end_date DATE,
    FOREIGN KEY (department_id) REFERENCES departments(id) ON DELETE SET NULL
);
```

Explanation:

- `id INT AUTO_INCREMENT PRIMARY KEY` → Unique project identifier.
- `project_name VARCHAR(100) NOT NULL` → Stores the project name.
- `department_id INT` → Links to `departments(id)`.
- `start_date DATE, end_date DATE` → Stores project timelines.
- `ON DELETE SET NULL` → If a department is deleted, `department_id` becomes `NULL` instead of deleting the project.

---

4. Creating the `employee_projects` Table (Many-to-Many Relationship)

```
sql
CopyEdit
CREATE TABLE employee_projects (
    employee_id INT,
    project_id INT,
    role VARCHAR(50),
    PRIMARY KEY (employee_id, project_id),
    FOREIGN KEY (employee_id) REFERENCES employees(id) ON DELETE CASCADE,
    FOREIGN KEY (project_id) REFERENCES projects(id) ON DELETE CASCADE
);
```

Explanation:

- This table represents a **many-to-many** relationship between employees and projects.
- `PRIMARY KEY (employee_id, project_id)` → Ensures an employee can only be assigned **once** per project.
- `FOREIGN KEY (employee_id) REFERENCES employees(id) ON DELETE CASCADE` → If an employee is deleted, their project assignments are also deleted.
- `FOREIGN KEY (project_id) REFERENCES projects(id) ON DELETE CASCADE` → If a project is deleted, all associated employees are removed from it.

---

5. Sample Data Insertion

Insert Departments

```
sql
CopyEdit
INSERT INTO departments (name, budget) VALUES
('HR', 50000),
('IT', 120000),
('Finance', 80000);
```

Insert Employees

```
sql
CopyEdit
INSERT INTO employees (name, department_id, salary, hire_date) VALUES
('Alice', 1, 55000, '2023-01-10'),
('Bob', 2, 75000, '2022-05-15'),
('Charlie', 3, 60000, '2021-09-20');
```

Insert Projects

```
sql
CopyEdit
INSERT INTO projects (project_name, department_id, start_date, end_date) VALUES
('Website Redesign', 2, '2024-01-01', '2024-06-30'),
('Recruitment Drive', 1, '2024-02-01', '2024-04-30');
```

Assign Employees to Projects

```
sql
CopyEdit
INSERT INTO employee_projects (employee_id, project_id, role) VALUES
(1, 2, 'HR Manager'),
(2, 1, 'Lead Developer');
```

---

6. Querying Data

Get All Employees with Their Department Names

```
sql
```

```
CopyEdit
SELECT e.id, e.name, d.name AS department, e.salary
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

Find Employees Assigned to a Project

```
sql
CopyEdit
SELECT e.name, p.project_name, ep.role
FROM employee_projects ep
JOIN employees e ON ep.employee_id = e.id
JOIN projects p ON ep.project_id = p.id;
```

Retrieving Data with SQL - A Deep Dive

SQL provides powerful ways to **retrieve** and **manipulate** data using the `SELECT` statement and **JOINs** to combine data from multiple tables. Let's go step by step.

SELECT Statement and FROM Clause

The `SELECT` statement is the **most basic SQL command** used to **retrieve data** from one or more tables.

Basic Syntax

```
sql
CopyEdit
SELECT column1, column2, ... FROM table_name;
```

- `SELECT` → Specifies the columns to retrieve.
- `FROM` → Specifies the table from which to fetch the data.

Example Table: `employees`

id	name	department_id	salary
1	Alice	10	50000
2	Bob	20	60000
3	Charlie	10	70000
4	David	30	80000

Example Queries

Retrieve all columns

```
sql
CopyEdit
SELECT * FROM employees;
```

Retrieve specific columns

```
sql
```

```
CopyEdit
SELECT name, salary FROM employees;
```

Filtering results using `WHERE`

```
sql
CopyEdit
SELECT name, salary FROM employees WHERE salary > 60000;
```

Result:

name	salary
Charlie	70000
David	80000

Understanding SQL Joins

A `JOIN` is used to **combine rows** from multiple tables based on a **related column**.

Types of Joins

Join Type	Description
INNER JOIN	Returns matching rows from both tables
LEFT JOIN	Returns all rows from the left table, and matching rows from the right table
RIGHT JOIN	Returns all rows from the right table, and matching rows from the left table
FULL JOIN	Returns all rows from both tables
SELF JOIN	Joins a table to itself

Example Tables: `employees` & `departments`

`employees` Table

id	name	department_id	salary
1	Alice	10	50000
2	Bob	20	60000
3	Charlie	10	70000
4	David	30	80000

`departments` Table

department_id	department_name
10	HR
20	IT
30	Finance
40	Marketing

INNER JOIN

Returns **only matching rows** where `employees.department_id = departments.department_id`.

Query

```
sql
CopyEdit
SELECT employees.name, employees.salary, departments.department_name
FROM employees
INNER JOIN departments ON employees.department_id = departments.department_id;
```

Result

name	salary	department_name
Alice	50000	HR
Bob	60000	IT
Charlie	70000	HR
David	80000	Finance

Explanation:

- The `INNER JOIN` returns **only rows** where there is a **match** in both tables.
- Since **Marketing (department_id = 40)** has no employees, it is **not included**.

LEFT JOIN (or LEFT OUTER JOIN)

Returns **all rows from the left table (`employees`)**, and **matching rows from the right table (`departments`)**.

Query

```
sql
CopyEdit
SELECT employees.name, employees.salary, departments.department_name
FROM employees
LEFT JOIN departments ON employees.department_id = departments.department_id;
```

Result

name	salary	department_name
Alice	50000	HR
Bob	60000	IT
Charlie	70000	HR
David	80000	Finance

Explanation:

- Similar to `INNER JOIN`, but if there were an employee **without a matching department**, it would still be included with `NULL` in `department_name`.

RIGHT JOIN (or RIGHT OUTER JOIN)

Returns **all rows from the right table (`departments`)**, and **matching rows from the left table (`employees`)**.

Query

```
sql
CopyEdit
SELECT employees.name, employees.salary, departments.department_name
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.department_id;
```

Result

name	salary	department_name
Alice	50000	HR
Bob	60000	IT
Charlie	70000	HR
David	80000	Finance
NULL	NULL	Marketing

Explanation:

- All **departments** are shown.
- Since **Marketing (department_id = 40)** has **no employees**, `NULL` appears in `name` and `salary`.

FULL JOIN (or FULL OUTER JOIN)

Returns **all rows from both tables**. If there's no match, `NULL` is used.

Query

```
sql
CopyEdit
SELECT employees.name, employees.salary, departments.department_name
FROM employees
FULL JOIN departments ON employees.department_id = departments.department_id;
```

Result

name	salary	department_name
Alice	50000	HR
Bob	60000	IT
Charlie	70000	HR
David	80000	Finance
NULL	NULL	Marketing

Explanation:

- Combines **LEFT JOIN** and **RIGHT JOIN** results.

MySQL Note:

- `FULL JOIN` is **not natively supported** in MySQL.
- Instead, use `UNION`:

```
sql
CopyEdit
SELECT employees.name, employees.salary, departments.department_name
FROM employees
LEFT JOIN departments ON employees.department_id = departments.department_id
UNION
SELECT employees.name, employees.salary, departments.department_name
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.department_id;
```

SELF JOIN

A `SELF JOIN` joins a table **to itself**.

Example: Finding Employees with the Same Salary

```
sql
CopyEdit
SELECT e1.name AS Employee1, e2.name AS Employee2, e1.salary
FROM employees e1
JOIN employees e2 ON e1.salary = e2.salary AND e1.id <> e2.id;
```

Explanation:

- The table is aliased (`e1`, `e2`).
- `e1.salary = e2.salary` ensures employees with the **same salary** are matched.
- `e1.id <> e2.id` prevents joining an employee to **itself**.

Summary of Joins

Join Type	Returns
INNER JOIN	Only matching rows from both tables
LEFT JOIN	All rows from the left table + matching rows from the right table
RIGHT JOIN	All rows from the right table + matching rows from the left table
FULL JOIN	All rows from both tables (MySQL requires `UNION`)
SELF JOIN	Joins a table to itself

Key Takeaways

- **`SELECT` is the fundamental command** to retrieve data.
- **`JOIN`s allow combining data** from multiple tables.
- **`INNER JOIN` is the most common** (returns only matching rows).
- **`LEFT JOIN` is used when you want to keep all left table rows.**

- **Use `SELF JOIN` for comparing rows within the same table.**

Want help with query optimization? Let me know!