### ****Creating Relationships in MySQL: One-to-One, One-to-Many, and Many-to-Many****

In **MySQL**, relationships between tables are established using **Primary Keys (PKs) and Foreign Keys (FKs)**.

## **1. One-to-One (1:1) Relationship**

✅ A **one-to-one relationship** means that **one record in Table A** is related to **one record in Table B**.  
✅ The **Foreign Key (FK) in Table B** references the **Primary Key (PK) in Table A**.

### ****Example: One Person has One Passport****

CREATE TABLE Persons (

person\_id INT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE Passports (

passport\_id INT PRIMARY KEY,

person\_id INT UNIQUE, -- Ensuring one person has only one passport

FOREIGN KEY (person\_id) REFERENCES Persons(person\_id) ON DELETE CASCADE

);

🔹 **Cascade Delete:** If a person is deleted, their passport is also deleted.  
🔹 **Order of Operations:**

* **INSERT** into Persons first, then Passports.
* **DELETE** from Passports first (if not using ON DELETE CASCADE).

## **2. One-to-Many (1:M) Relationship**

✅ In a **one-to-many relationship**, **one record in Table A** relates to **many records in Table B**.  
✅ The **Foreign Key (FK) in Table B** references the **Primary Key (PK) in Table A**.

### ****Example: One Customer has Many Orders****

CREATE TABLE Customers (

customer\_id INT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE Orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

order\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id) ON DELETE CASCADE

);

🔹 **Cascade Delete:** If a customer is deleted, all their orders are deleted.  
🔹 **Order of Operations:**

* **INSERT** into Customers first, then Orders.
* **DELETE** from Orders first (if not using ON DELETE CASCADE).

## **3. Many-to-Many (M:N) Relationship**

✅ A **many-to-many relationship** requires a **junction table** to link two tables.  
✅ The junction table contains **Foreign Keys (FKs)** from both tables.

### ****Example: Students Enrolled in Multiple Courses****

CREATE TABLE Students (

student\_id INT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE Courses (

course\_id INT PRIMARY KEY,

course\_name VARCHAR(100)

);

CREATE TABLE StudentCourses ( -- Junction Table

student\_id INT,

course\_id INT,

PRIMARY KEY (student\_id, course\_id),

FOREIGN KEY (student\_id) REFERENCES Students(student\_id) ON DELETE CASCADE,

FOREIGN KEY (course\_id) REFERENCES Courses(course\_id) ON DELETE CASCADE

);

🔹 **Cascade Delete:** If a student or course is deleted, their records in StudentCourses are deleted.  
🔹 **Order of Operations:**

* **INSERT** into Students and Courses first, then StudentCourses.
* **DELETE** from StudentCourses first, then Students or Courses (if not using ON DELETE CASCADE).

## **Order of Operations: Insert & Delete**

| **Relationship Type** | **Insert Order** | **Delete Order** |
| --- | --- | --- |
| **One-to-One (1:1)** | Insert into Parent → Insert into Child | Delete from Child → Delete from Parent (if no CASCADE) |
| **One-to-Many (1:M)** | Insert into Parent → Insert into Child | Delete from Child → Delete from Parent (if no CASCADE) |
| **Many-to-Many (M:N)** | Insert into both Parent Tables → Insert into Junction Table | Delete from Junction Table → Delete from Parent Tables (if no CASCADE) |

### ****Final Thoughts****

✔ Always **insert parent first** and **delete child first** unless using ON DELETE CASCADE.  
✔ Use ON DELETE CASCADE for automatic deletion of related records.  
✔ Many-to-Many relationships **always require a junction table**.

### ✅ ****Order of Data Updation in MySQL Relationships****

The **order of data updation** in MySQL relationships is slightly different from the **delete operation** because updates depend on the **relationship type** and **foreign key constraints**.

## 🔑 **Order of Data Updation by Relationship Type**

### ****1. One-to-One (1:1) Relationship****

In a one-to-one relationship:

* **Parent Table:** Main table where the primary key is defined (e.g., Persons).
* **Child Table:** Dependent table with a foreign key (e.g., Passports).

### ****Update Order****

✅ First update the **Parent Table** (if primary key or main data is changing).  
✅ Then update the **Child Table** (if child data is changing).

### ****Example:****

UPDATE Persons SET name = 'Ali Ahmed' WHERE person\_id = 1; -- Parent Table First

UPDATE Passports SET passport\_id = 98765 WHERE person\_id = 1; -- Child Table Second

### ****2. One-to-Many (1:M) Relationship****

In a one-to-many relationship:

* **Parent Table:** Table with the primary key (e.g., Customers).
* **Child Table:** Table with the foreign key (e.g., Orders).

### ****Update Order****

✅ First update the **Parent Table** if changing non-key data.  
✅ Update the **Child Table** if changing child data.

👉 If you're updating the **foreign key value in the parent table** and it has a CASCADE option, then:

* Update the **Parent Table** first, and MySQL will automatically update the **Child Table**.

### ****Example:****

UPDATE Customers SET name = 'Ahmed Ali' WHERE customer\_id = 1; -- Parent Table First

UPDATE Orders SET order\_date = '2025-03-08' WHERE customer\_id = 1; -- Child Table Second

### ****3. Many-to-Many (M:N) Relationship****

In a many-to-many relationship:

* Two Parent Tables (e.g., Students and Courses).
* Junction Table holds both **foreign keys** (e.g., StudentCourses).

### ****Update Order****

✅ First update the **Parent Tables** if changing primary or non-key data.  
✅ Then update the **Junction Table** if the foreign key values are changing.

### ****Example:****

UPDATE Students SET name = 'Ali Hassan' WHERE student\_id = 1; -- Parent Table 1

UPDATE Courses SET course\_name = 'Advanced PHP' WHERE course\_id = 101; -- Parent Table 2

UPDATE StudentCourses SET course\_id = 102 WHERE student\_id = 1; -- Junction Table

## 🎯 **Summary Table**

| **Relationship Type** | **Insert Order** | **Update Order** | **Delete Order** |
| --- | --- | --- | --- |
| One-to-One (1:1) | Parent → Child | Parent → Child | Child → Parent |
| One-to-Many (1:M) | Parent → Child | Parent → Child | Child → Parent |
| Many-to-Many (M:N) | Parent → Junction | Parent → Junction | Junction → Parent |

### 💡 Important Points

* Always **update parent first** before child to avoid **foreign key constraint errors**.
* Use **ON UPDATE CASCADE** to automatically update foreign key values in child tables.
* If there is **no cascade constraint**, update manually in the correct order.

**Advance Features:**

MySQL supports functions, views, stored procedures, and triggers just like MS SQL Server, but with some differences in syntax and features. Here's a breakdown of each:

**1. Functions in MySQL**

MySQL allows **user-defined functions (UDFs)** but they are limited to returning a single value.  
**Syntax:**

DELIMITER //

CREATE FUNCTION get\_total(price DECIMAL(10,2), quantity INT)

RETURNS DECIMAL(10,2)

DETERMINISTIC

BEGIN

RETURN price \* quantity;

END //

DELIMITER ;

**Usage:**

SELECT get\_total(100, 2); -- Output: 200

**2. Views in MySQL**

MySQL supports **views** to simplify complex queries.  
**Syntax:**

CREATE VIEW EmployeeView AS

SELECT id, name, salary FROM Employees WHERE salary > 50000;

**Usage:**

SELECT \* FROM EmployeeView;

**3. Stored Procedures in MySQL**

MySQL allows **stored procedures**, which are blocks of SQL code stored in the database.  
**Syntax:**

DELIMITER //

CREATE PROCEDURE GetEmployees()

BEGIN

SELECT \* FROM Employees;

END //

DELIMITER ;

**Usage:**

CALL GetEmployees();

**4. Triggers in MySQL**

Triggers are used to execute SQL statements **before or after** an INSERT, UPDATE, or DELETE.  
**Syntax:**

DELIMITER //

CREATE TRIGGER after\_employee\_insert

AFTER INSERT ON Employees

FOR EACH ROW

BEGIN

INSERT INTO Logs(action, timestamp) VALUES ('New Employee Added', NOW());

END //

DELIMITER ;

**Trigger executes automatically when a new employee is added.**

**Key Differences Between MySQL and MS SQL Server**

| **Feature** | **MySQL** | **MS SQL Server** |
| --- | --- | --- |
| Functions | Yes (but no table-valued functions) | Yes |
| Views | Yes | Yes |
| Stored Procedures | Yes | Yes |
| Triggers | Yes | Yes |
| Transactions | Yes (with InnoDB) | Yes |

## **1. Indexes in MySQL**

### ****What is an Index?****

An **index** is a data structure that speeds up data retrieval by reducing the number of rows MySQL needs to scan.

🔹 Without an index: MySQL scans all rows (**full table scan**).  
🔹 With an index: MySQL directly finds the required data (**indexed search**).

### ****When to Use Indexes?****

Use indexes when: ✅ Searching for specific rows (WHERE, JOIN).  
✅ Sorting (ORDER BY).  
✅ Filtering (GROUP BY).

### ****How to Use Indexes?****

#### **Example 1: Creating an Index for Faster Searches**

CREATE INDEX idx\_name ON Employees(name);

Now, searching by name will be **faster**:

SELECT \* FROM Employees WHERE name = 'John Doe';

#### **Example 2: Composite Index (Multi-column Index)**

CREATE INDEX idx\_name\_salary ON Employees(name, salary);

This improves queries like:

SELECT \* FROM Employees WHERE name = 'John Doe' AND salary > 50000;

#### **Types of Indexes in MySQL**

| **Type** | **Description** |
| --- | --- |
| **Primary Index** | Auto-created for PRIMARY KEY |
| **Unique Index** | Ensures uniqueness (UNIQUE KEY) |
| **Composite Index** | Index on multiple columns |
| **Full-text Index** | Used for text search (FULLTEXT) |
| **Spatial Index** | Used for GIS (Geographical data) |

## **2. Clustering in MySQL**

### ****What is Clustering?****

In **clustering**, the database stores rows **physically together** based on their primary key.

✅ In MySQL, **InnoDB** automatically **clusters** data by the **PRIMARY KEY**.  
❌ MySQL does not support custom clustering like MS SQL Server.

### ****When to Use Clustering?****

Use clustering when: ✅ You often query by **primary key**.  
✅ You want to **improve range queries** (BETWEEN, ORDER BY).

### ****How Clustering Works in MySQL?****

* **Table without Primary Key** → MySQL creates an **internal row ID**.
* **Table with Primary Key** → Rows are **clustered by primary key**.

#### **Example of Clustering**

CREATE TABLE Orders (

order\_id INT PRIMARY KEY, -- Data is physically sorted by order\_id

customer\_id INT,

order\_date DATE

) ENGINE=InnoDB;

Now, queries like:

SELECT \* FROM Orders WHERE order\_id = 1001;

are **faster** because the data is **physically sorted**.

### ****Key Differences: Index vs Clustering****

| **Feature** | **Index** | **Clustering** |
| --- | --- | --- |
| **Purpose** | Speeds up search | Organizes physical storage |
| **Used For** | Fast lookups (WHERE, JOIN) | Range queries (BETWEEN, ORDER BY) |
| **Customization** | Can be added/removed | Only one per table (Primary Key) |
| **Multiple?** | Yes (Multiple indexes) | No (Only one clustering key) |

### ****Final Thoughts****

✔ Use **indexes** for faster searches.  
✔ Use **clustering** (by primary key) for efficient storage & retrieval.

### ****Advanced MySQL Topics****

Once you're comfortable with the basics of MySQL, you can explore **advanced topics** to optimize performance, improve security, and work with complex queries. Here’s a list of key **advanced MySQL topics**:

### ****1. Query Optimization & Performance Tuning****

✅ **EXPLAIN and ANALYZE** – Understanding query execution plans.  
✅ **Index Optimization** – Using proper indexing strategies (B-Tree, Full-Text, Hash indexes).  
✅ **Query Caching** – Storing query results for faster retrieval.  
✅ **Partitioning & Sharding** – Splitting large tables for better performance.  
✅ **Optimizing Joins & Subqueries** – Using JOIN efficiently instead of nested queries.

🔹 **Example:** Using EXPLAIN to analyze a query

EXPLAIN SELECT \* FROM Orders WHERE customer\_id = 100;

### ****2. Stored Procedures & Functions****

✅ **Stored Procedures** – Precompiled SQL blocks for reusability.  
✅ **User-Defined Functions (UDFs)** – Custom functions for complex calculations.  
✅ **Cursors** – Iterating over query results row by row.

🔹 **Example:** Creating a stored procedure

DELIMITER //

CREATE PROCEDURE GetCustomerOrders(IN customerID INT)

BEGIN

SELECT \* FROM Orders WHERE customer\_id = customerID;

END //

DELIMITER ;

### ****3. Transactions & Concurrency Control****

✅ **ACID Properties** – Ensuring database reliability.  
✅ **Transaction Control Statements** – COMMIT, ROLLBACK, SAVEPOINT.  
✅ **Isolation Levels** – Handling concurrent transactions (READ COMMITTED, SERIALIZABLE, etc.).

🔹 **Example:** Using a transaction

START TRANSACTION;

UPDATE Accounts SET balance = balance - 100 WHERE id = 1;

UPDATE Accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;

### ****4. Triggers & Event Scheduling****

✅ **Triggers** – Automating actions before/after INSERT, UPDATE, DELETE.  
✅ **Scheduled Events** – Running tasks at specified intervals (like CRON jobs).

🔹 **Example:** Creating a trigger

DELIMITER //

CREATE TRIGGER after\_insert\_employee

AFTER INSERT ON Employees

FOR EACH ROW

BEGIN

INSERT INTO Logs(action, timestamp) VALUES ('Employee Added', NOW());

END //

DELIMITER ;

### ****5. JSON & NoSQL Features in MySQL****

✅ **JSON Data Type** – Storing and querying JSON data.  
✅ **JSON Functions** – JSON\_EXTRACT(), JSON\_ARRAYAGG(), JSON\_OBJECT().

🔹 **Example:** Storing JSON in MySQL

CREATE TABLE Products (

id INT PRIMARY KEY,

details JSON

);

INSERT INTO Products VALUES (1, '{"name": "Laptop", "price": 1200}');

SELECT details->>'$.name' FROM Products WHERE id = 1;

### ****6. Replication & High Availability****

✅ **Master-Slave Replication** – Copying data from one server to another.  
✅ **Master-Master Replication** – Bi-directional data synchronization.  
✅ **Failover & Load Balancing** – Using MySQL Cluster for high availability.

🔹 **Example:** Setting up replication

CHANGE MASTER TO MASTER\_HOST='192.168.1.1', MASTER\_USER='replica\_user', MASTER\_PASSWORD='password';

START SLAVE;

### ****7. Security & User Management****

✅ **Roles & Permissions** – Restricting user access.  
✅ **Data Encryption** – Using AES encryption.  
✅ **SQL Injection Prevention** – Using prepared statements.

🔹 **Example:** Creating a secure user

CREATE USER 'backup\_user'@'localhost' IDENTIFIED BY 'securepassword';

GRANT SELECT, LOCK TABLES ON mydb.\* TO 'backup\_user'@'localhost';

### ****8. Full-Text Search & Indexing****

✅ **Full-Text Index** – Faster text-based searches.  
✅ **MATCH AGAINST** – Searching large text fields efficiently.

🔹 **Example:** Using full-text search

CREATE TABLE Articles (

id INT PRIMARY KEY,

content TEXT,

FULLTEXT (content)

);

SELECT \* FROM Articles WHERE MATCH(content) AGAINST ('database optimization');

### ****9. Partitioning & Large Data Handling****

✅ **Range Partitioning** – Dividing data into multiple partitions.  
✅ **Hash Partitioning** – Distributing data across partitions using hashing.

🔹 **Example:** Range Partitioning

CREATE TABLE Orders (

order\_id INT NOT NULL,

order\_date DATE NOT NULL,

PRIMARY KEY (order\_id, order\_date)

) PARTITION BY RANGE(YEAR(order\_date)) (

PARTITION p1 VALUES LESS THAN (2022),

PARTITION p2 VALUES LESS THAN (2023),

PARTITION p3 VALUES LESS THAN (2024)

);

### ****10. MySQL with Big Data & Cloud****

✅ **MySQL with Hadoop & Spark** – Integrating with big data tools.  
✅ **MySQL on AWS & Google Cloud** – Using managed MySQL services.  
✅ **Database Backups & Recovery** – Automating database backups.

🔹 **Example:** Exporting a MySQL database

mysqldump -u root -p mydatabase > backup.sql

## **Conclusion**

✔ Mastering these **advanced MySQL topics** will help you optimize performance, improve security, and handle complex data scenarios.  
✔ If you’re looking to **specialize**, focus on **query optimization, security, and replication** first.

Query Optimization:

### ✅ ****How to Optimize Queries in MySQL?****

Query optimization in MySQL is crucial for improving database performance, reducing execution time, and handling large datasets efficiently. Below are **best practices** with **examples**.

## 🔹 **1. Use Indexing for Faster Lookups**

Indexes improve query speed by allowing MySQL to find data **without scanning the entire table**.

### ****Example: Without Index (Slow Query)****

SELECT \* FROM Orders WHERE customer\_id = 101;

🚨 **Problem:** If customer\_id is not indexed, MySQL performs a **full table scan**.

### ****Solution: Add an Index****

CREATE INDEX idx\_customer ON Orders(customer\_id);

Now, MySQL **quickly** locates rows without scanning the entire table.

## 🔹 **2. Use** EXPLAIN **to Analyze Queries**

EXPLAIN helps understand how MySQL executes queries and suggests improvements.

### ****Example: Analyzing a Query****

EXPLAIN SELECT \* FROM Orders WHERE customer\_id = 101;

🔹 **Output (Simplified)**

| **id** | **select\_type** | **table** | **type** | **possible\_keys** | **key** | **rows** | **Extra** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | SIMPLE | Orders | ref | idx\_customer | idx\_customer | 10 | Using index |

✔ **Key Insights:**

* idx\_customer is used, meaning the query is optimized.
* If type = ALL, it means a **full table scan** is happening (bad performance).

## 🔹 **3. Avoid** SELECT \***, Specify Columns**

Fetching **only required columns** reduces memory usage and execution time.

### ****Example: Bad Query (Slow)****

SELECT \* FROM Customers WHERE city = 'Lahore';

🚨 **Problem:** Fetches all columns, even unused ones.

### ****Optimized Query****

SELECT customer\_id, name FROM Customers WHERE city = 'Lahore';

✔ Fetches only required data, improving performance.

## 🔹 **4. Use Proper Joins & Avoid Nested Queries**

Joins are **faster** than subqueries because they retrieve data in one step.

### ****Example: Bad Query (Using Subquery)****

SELECT name FROM Customers WHERE customer\_id IN (SELECT customer\_id FROM Orders WHERE amount > 5000);

🚨 **Problem:** Executes the subquery **first**, then the outer query.

### ****Optimized Query (Using JOIN)****

SELECT DISTINCT Customers.name

FROM Customers

JOIN Orders ON Customers.customer\_id = Orders.customer\_id

WHERE Orders.amount > 5000;

✔ **Faster Execution** by using JOIN instead of a subquery.

## 🔹 **5. Use** LIMIT **to Restrict Data**

Fetching all rows when only a few are needed wastes resources.

### ****Example: Without LIMIT (Slow)****

SELECT \* FROM Orders ORDER BY order\_date DESC;

🚨 **Problem:** Fetches all records, even if not needed.

### ****Optimized Query****

SELECT \* FROM Orders ORDER BY order\_date DESC LIMIT 10;

✔ Fetches only **10 records**, improving response time.

## 🔹 **6. Use** GROUP BY **Efficiently**

Avoid using GROUP BY on unnecessary columns.

### ****Example: Bad Query****

SELECT customer\_id, order\_date, COUNT(\*) FROM Orders GROUP BY customer\_id, order\_date;

🚨 **Problem:** Groups by multiple columns unnecessarily.

### ****Optimized Query****

SELECT customer\_id, COUNT(\*) FROM Orders GROUP BY customer\_id;

✔ Uses **fewer columns**, improving efficiency.

## 🔹 **7. Optimize** ORDER BY **with Indexing**

Sorting large data sets without indexes is slow.

### ****Example: Without Index (Slow)****

SELECT \* FROM Orders ORDER BY order\_date;

🚨 **Problem:** Full table scan to sort the data.

### ****Solution: Add an Index****

CREATE INDEX idx\_order\_date ON Orders(order\_date);

✔ **Now sorting is fast!**

## 🔹 **8. Use** HAVING **Only When Necessary**

Use WHERE instead of HAVING whenever possible.

### ****Example: Bad Query (Using**** HAVING****)****

SELECT customer\_id, COUNT(\*) FROM Orders

GROUP BY customer\_id

HAVING customer\_id > 10;

🚨 **Problem:** HAVING filters data **after aggregation**, slowing performance.

### ****Optimized Query (Using**** WHERE****)****

SELECT customer\_id, COUNT(\*) FROM Orders

WHERE customer\_id > 10

GROUP BY customer\_id;

✔ Uses WHERE, reducing unnecessary processing.

## 🔹 **9. Use** UNION ALL **Instead of** UNION **When Possible**

UNION removes duplicates, which is slow. UNION ALL is **faster** if duplicates don’t matter.

### ****Example: Using**** UNION ****(Slow)****

SELECT name FROM Customers WHERE city = 'Lahore'

UNION

SELECT name FROM Suppliers WHERE city = 'Lahore';

🚨 **Problem:** MySQL checks for duplicates.

### ****Optimized Query (****UNION ALL****)****

SELECT name FROM Customers WHERE city = 'Lahore'

UNION ALL

SELECT name FROM Suppliers WHERE city = 'Lahore';

✔ **Faster** because it skips duplicate checks.

## 🔹 **10. Optimize** INSERT **with Bulk Inserts**

Inserting multiple rows in **one query** is faster than inserting **one by one**.

### ****Example: Bad Approach (Multiple Queries)****

INSERT INTO Products (name, price) VALUES ('Laptop', 1200);

INSERT INTO Products (name, price) VALUES ('Phone', 800);

🚨 **Problem:** Each INSERT query runs separately.

### ****Optimized Query (Bulk Insert)****

INSERT INTO Products (name, price)

VALUES ('Laptop', 1200), ('Phone', 800);

✔ **Faster execution** by reducing query overhead.

## 🔹 **11. Use Proper Data Types**

Using incorrect data types wastes storage and slows queries.

### ****Example: Bad Schema****

CREATE TABLE Users (

user\_id VARCHAR(255), -- Bad choice (should be INT)

name VARCHAR(255),

age VARCHAR(10) -- Bad choice (should be INT)

);

🚨 **Problem:**

* user\_id should be INT (Primary Key).
* age should be INT, not VARCHAR(10).

### ****Optimized Schema****

CREATE TABLE Users (

user\_id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(100),

age INT

);

✔ **Efficient storage & faster indexing.**

## ✅ **Final Summary: Key Optimization Techniques**

| **❌ Bad Practice** | **✅ Optimized Approach** |
| --- | --- |
| No Indexes | Use Indexes (CREATE INDEX) |
| SELECT \* | Select only needed columns |
| Subqueries | Use JOIN instead |
| No LIMIT | Use LIMIT to reduce rows |
| HAVING for Filtering | Use WHERE instead |
| UNION | Use UNION ALL when possible |
| Multiple Inserts | Use Bulk Insert |

## 🎯 **Final Thoughts**

✔ **Indexes** and **JOINs** are the most important optimizations.  
✔ Always analyze queries using **EXPLAIN** before optimizing.  
✔ Avoid **unnecessary full-table scans** by filtering data early.  
✔ Optimize **schema design** for better performance.

Example:

### 🔥 ****Real-World Scenario for MySQL Query Optimization****

Let’s assume we are developing an **eCommerce website** with the following database tables:

* **Customers** (customer details)
* **Orders** (customer orders)
* **Products** (product details)
* **OrderItems** (items within each order)

## **📌 Scenario: Slow Query Issue**

A report requires fetching **customer details with their latest orders**, but the current query is very slow.

### ****🔴 Initial Slow Query****

SELECT Customers.customer\_id, Customers.name, Orders.order\_id, Orders.order\_date

FROM Customers

JOIN Orders ON Customers.customer\_id = Orders.customer\_id

WHERE Orders.order\_date = (SELECT MAX(order\_date) FROM Orders WHERE Orders.customer\_id = Customers.customer\_id);

🚨 **Problems:**

1. **Subquery for MAX(order\_date) runs for every row**, causing high CPU usage.
2. **No indexing**, leading to a full table scan.
3. **Large dataset filtering inefficiency**.

## **✅ Optimized Solution**

### ****1️⃣ Add Indexes for Faster Lookups****

CREATE INDEX idx\_order\_date ON Orders(order\_date);

CREATE INDEX idx\_customer\_order ON Orders(customer\_id, order\_date);

✔ **Why?** Now, MySQL can quickly locate the latest order using the index.

### ****2️⃣ Use**** JOIN ****Instead of Subquery****

Instead of running a subquery for MAX(order\_date), use a **JOIN** with ORDER BY and LIMIT.

SELECT Customers.customer\_id, Customers.name, Orders.order\_id, Orders.order\_date

FROM Customers

JOIN Orders ON Customers.customer\_id = Orders.customer\_id

JOIN (

SELECT customer\_id, MAX(order\_date) AS latest\_order

FROM Orders

GROUP BY customer\_id

) latest ON Orders.customer\_id = latest.customer\_id AND Orders.order\_date = latest.latest\_order;

✔ **Why?**

* The **subquery (latest) runs only once** instead of per row.
* **Indexed search speeds up filtering**.
* **JOINs** improve performance over nested queries.

### ****📊 Performance Comparison****

| **Query** | **Execution Time (Seconds)** | **Rows Scanned** |
| --- | --- | --- |
| **Initial Query (Subquery)** | 🚨 4.5 sec | ❌ 100,000+ |
| **Optimized Query (Index + JOIN)** | ✅ 0.7 sec | ✅ 5,000 |

🔹 **Result:** 🚀 **5x Faster!**

## **🔥 Conclusion**

🔹 **Indexes reduce scan time**.  
🔹 **JOINs are better than subqueries**.  
🔹 **Filtering early (GROUP BY, ORDER BY)** optimizes performance.