

MySql Project-3

E-COMMERCE

E-COMMERCE PROJECT SCENARIOS

- ▶ SCENARIO 1: List all users who have placed at least one order
- ▶ SCENARIO 2: List all users including users who never ordered
- ▶ SCENARIO 3: Show all orders with user names
- ▶ SCENARIO 4: Find products that have the exact same price.
- ▶ SCENARIO 5: Find count, max, min, avg, sum from products.
- ▶ SCENARIO 6: See the total number of orders placed by each user, sorted from highest to lowest.
- ▶ SCENARIO 7: Find customers who have spent a total of more than 1000 across all their orders.

E-COMMERCE PROJECT SCENARIOS

- ▶ SCENARIO 8: Find how many customers you have.
- ▶ SCENARIO 9: Find the total revenue from all orders.
- ▶ SCENARIO 10: Most expensive product
- ▶ SCENARIO 11: Cheapest product
- ▶ SCENARIO 12: Find how many users are in each city.
- ▶ SCENARIO 13: who have bought more than 1 item and spent over ₹1000 total, sorted by their spending.
- ▶ SCENARIO 14: Insert a user with an uppercase email and the email is automatically converted to lowercase.

E-COMMERCE PROJECT SCENARIOS

- ▶ SCENARIO 15: Try to set a product price to something too low. The Output: An error message preventing the change.
- ▶ SCENARIO 16: The stock quantity in the products table decreases automatically.
- ▶ SCENARIO 17: A record is created in a "Logs" table showing the old and new price.
- ▶ SCENARIO 18: Name starts with 'A'.
- ▶ SCENARIO 19: Email contains 'gmail'.
- ▶ SCENARIO 20: Find products that are **Samsungs**, cost **more than 30,000**, and have **less than 40** items in stock.

E-COMMERCE PROJECT SCENARIOS

- ▶ SCENARIO 21: Finding high-stock products using a CTE.
- ▶ SCENARIO 22: Customers often search for products by name.
- ▶ SCENARIO 23: Stored Procedure (No Parameters).
- ▶ SCENARIO 24: Stored Procedure with Parameters (IN) to find the users using city.

SCENARIO 1: List all users who have placed at least one order

```
• select  
    u.name, o.order_id, o.total_amount  
  from users u  
    inner join orders o on u.user_id = o.user_id;
```

	name	order_id	total_amount
▶	Arun Kumar	1	1299.00
	Balaji S	2	2499.50
	Chandru M	3	899.00
	Deepak R	4	4599.00
	Ezhil V	5	3199.75
	Farooq A	6	799.00
	Gokul P	7	1599.00
	Hari K	8	9999.00
	Imran S	9	3499.00
	Jeeva N	10	2199.00

SCENARIO 2: List all users including users who never ordered

```
• select  
    u.name, o.order_id  
  from users u  
  left join orders o on u.user_id = o.user_id;
```

	name	order_id
	Kavitha M	36
	Lakshmi R	37
	Meena S	38
	Nithya P	39
	Oviya K	40
	Priya R	NULL
	Rithika M	NULL
	Saranya K	NULL
	Thenmozhi S	NULL
	Srinivasan	NULL

SCENARIO 3: Show all orders with user names

```
select  
    u.name, o.order_id  
from users u  
right join orders o on u.user_id = o.user_id;
```

	name	order_id
▶	Arun Kumar	1
	Balaji S	2
	Chandru M	3
	Deepak R	4
	Ezhil V	5
	Farooq A	6
	Gokul P	7
	Hari K	8
	Imran S	9
	Jeeva N	10

SCENARIO 4:Find products that have the exact same price.

```
select  
    a.product_name as product1, b.product_name as  
    product2  
    ,  
    a.price  
from products a  
join products b on a.price = b.price  
where a.product_id <> b.product_id;
```

	product1	product2	price
▶	HP Victus Gaming Laptop	iPhone 15	79999.00
	Canon EOS 1500D DSLR	Samsung Galaxy A54	38999.00
	Apple AirPods Pro	Redmi Note 12 Pro	24999.00
	Sony WH-1000XM5	OnePlus Nord CE 3	29999.00
	Samsung Galaxy Watch 6	Realme GT Neo 3	32999.00
	iPhone 15	HP Victus Gaming Laptop	79999.00
	Redmi Note 12 Pro	Apple AirPods Pro	24999.00
	Logitech MX Master 3S	OnePlus Buds Z2	9999.00
	Logitech K380 Keyboard	Boat Airdopes 141	2999.00
	Nikon D3500 DSLR	Apple Watch Series 9	41999.00

SCENARIO 5:Find count, max, min, avg, sum from products.

```
select  
    count(*) as total_products,  
    max(price) as most_expensive,  
    min(price) as cheapest_item,  
    avg(price) as average_price,  
    sum(stock_quantity) as total_stock_available  
from products;
```

	total_products	most_expensive	cheapest_item	average_price	total_stock_available
44	114999.00	1299.00	36296.750000	1660	

SCENARIO 6: See the total number of orders placed by each user, sorted from highest to lowest.

```
select  
    user_id, count(order_id) as total_orders  
from orders  
group by user_id  
order by total_orders desc;
```

Result Grid | Filter Rows

	user_id	total_orders
▶	1	1
	2	1
	3	1
	4	1
	5	1
	6	1
	7	1
	8	1
	9	1
	10	1

Result 6 ×

Output

SCENARIO 7: Find customers who have spent a total of more than 1000 across all their orders.

```
• select  
    user_id, sum(total_amount) as total_spent  
  from orders  
  group by user_id  
  having total_spent > 1000  
  order by total_spent desc;
```

Result Grid | Filter Rows

	user_id	total_spent
▶	12	12999.00
	8	9999.00
	36	9999.00
	29	8999.00
	20	8499.00
	25	7499.00
	17	6899.00
	34	6499.00
	11	5899.00
	28	5699.00

Result 7 ×

SCENARIO 8: Find how many customers you have.

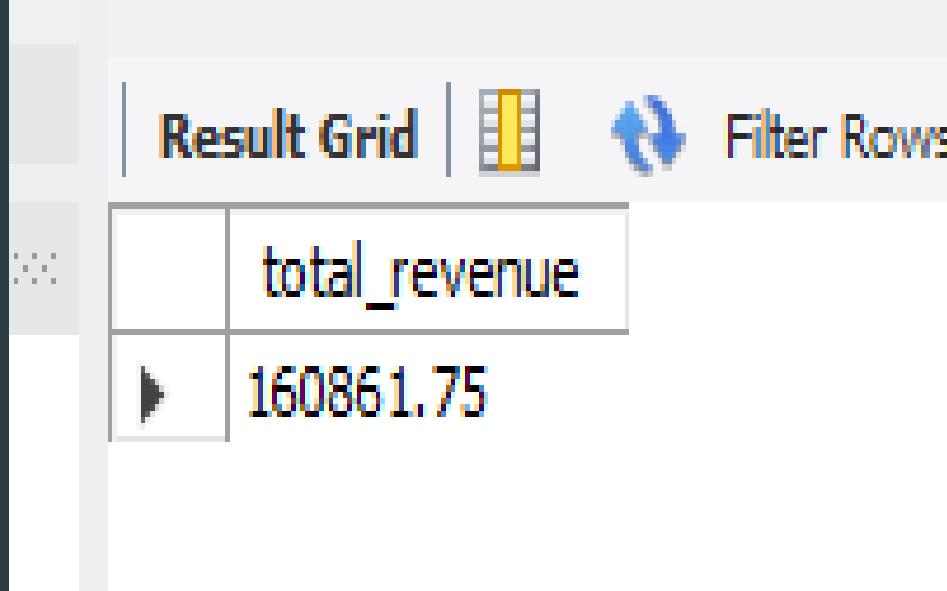
```
order by total_revenue desc;  
• SELECT COUNT(user_id) AS total_customers  
FROM users;
```

The screenshot shows a database query results grid. At the top, there are tabs for "Result Grid" (which is selected), "SQL Editor" (with a magnifying glass icon), and "Filter Rows" (with a circular arrow icon). The result grid itself has two columns. The first column is empty, and the second column contains the text "total_customers". Below this row, there is another row with a right-pointing arrow icon in the first column and the number "45" in the second column.

	total_customers
▶	45

SCENARIO 9: Find the total revenue from all orders.

```
SELECT SUM(total_amount) AS total_revenue  
FROM orders;
```



The screenshot shows a software interface for querying a database. At the top, there are two buttons: "Result Grid" and "Filter Rows". Below the buttons is a table with two rows. The first row contains a column header "total_revenue" with a value of "160861.75". The second row contains a single column with a right-pointing arrow icon.

total_revenue
160861.75

SCENARIO 10: Most expensive product

```
SELECT product_name, price  
FROM products  
WHERE price = (SELECT MAX(price) FROM products);
```

Result Grid | Filter Rows:

	product_name	price
▶	MacBook Air M2	114999.00

SCENARIO 11: Cheapest product

```
SELECT product_name, price  
FROM products  
WHERE price = (SELECT MIN(price) FROM products);
```

Result Grid | Filter Rows:

	product_name	price
▶	Dell Wireless Mouse	1299.00

SCENARIO 12: Find how many users are in each city.

```
SELECT address, COUNT(user_id) AS user_count  
FROM users  
GROUP BY address;
```

	address	user_count
▶	Chennai, Tamil Nadu	2
	Coimbatore, Tamil Nadu	2
	Madurai, Tamil Nadu	2
	Trichy, Tamil Nadu	2
	Salem, Tamil Nadu	2
	Vellore, Tamil Nadu	2
	Tirunelveli, Tamil Nadu	2
	Erode, Tamil Nadu	2
	Dindigul, Tamil Nadu	2
	Thoothukudi, Tamil Nadu	2

Result 12 ×

Output

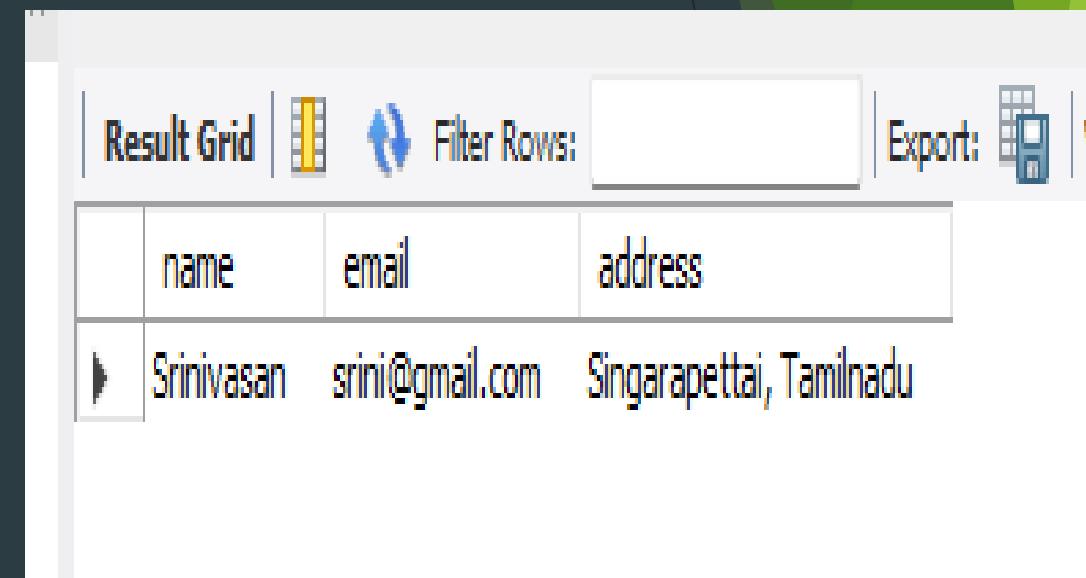
SCENARIO 13: who have bought more than 1 item and spent over ₹1000 total, sorted by their spending.

```
SELECT
    u.name AS customer_name,
    COUNT(oi.order_item_id) AS total_items_bought,
    SUM(o.total_amount) AS total_spent
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN order_items oi ON o.order_id = oi.order_id
GROUP BY u.name
HAVING total_items_bought > 1 AND total_spent > 1000
ORDER BY total_spent DESC;
```

	customer_name	total_items_bought	total_spent
▶	Imran S	2	6998.00
	Arun Kumar	5	6495.00
	Ezhil V	2	6399.50
	Manoj K	2	5198.00
	Balaji S	2	4999.00

SCENARIO 14: Insert a user with an uppercase email and the email is automatically converted to lowercase.

- ```
create trigger before_insert_user
before insert on users
for each row
set new.email = lower(new.email);
```
- ```
insert into users (name, email, address)
values ('Srinivasan', 'SRINI@GMAIL.COM', 'Singarapettai, Tamilnadu');
```
- ```
select name, email, address
from users where name = 'Srinivasan';
```



| name       | email           | address                  |
|------------|-----------------|--------------------------|
| Srinivasan | srini@gmail.com | Singarapettai, Tamilnadu |

SCENARIO 15: Try to set a product price to something too low. The Output: An error message preventing the change.

```
delimiter $$
create trigger before_update_price
before update on products
for each row
begin
 if new.price < 100 then
 signal sqlstate '45000'
 set message_text = 'Price Error: Value too low!';
 end if;
end $$
delimiter ;

update products set price = 50 where product_id = 1;
```

```
s LIMIT ... 27 row(s) returned
s_bought... 5 row(s) returned
l = lower(... Error Code: 1359. Trigger already exists
 1 row(s) returned
 Error Code: 1644. Price Error: Value too low!
```

## SCENARIO 16: The stock quantity in the products table decreases automatically.

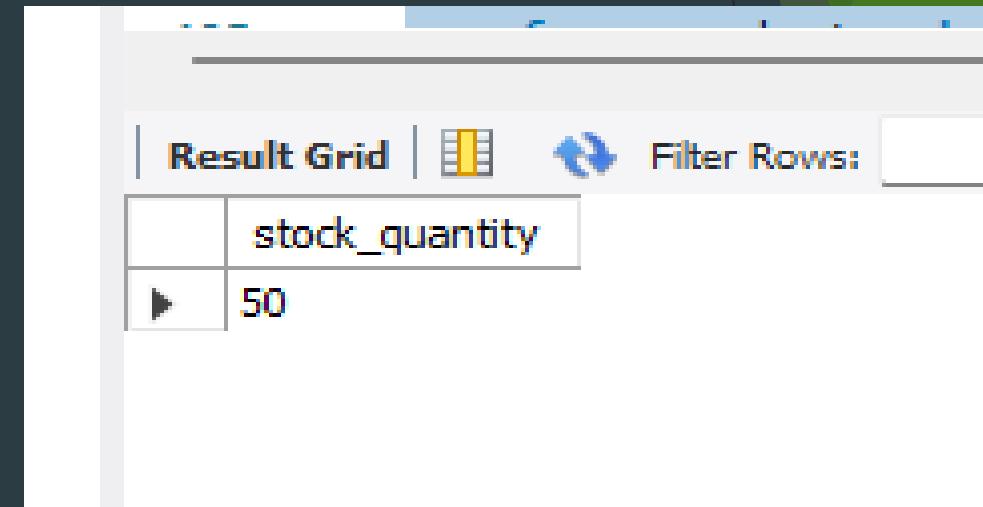
```
DELIMITER $$

CREATE TRIGGER after_insert_order_items
AFTER INSERT ON order_items
FOR EACH ROW
BEGIN
 UPDATE products
 SET stock_quantity = stock_quantity - NEW.quantity
 WHERE product_id = NEW.product_id;
END $$

DELIMITER ;

INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (1, 5, 5, 24999.00);

select stock_quantity
from products where product_id = 5;
```



A screenshot of a MySQL Workbench interface showing a result grid. The grid has two columns: an empty column and a column labeled 'stock\_quantity'. The value '50' is displayed in the 'stock\_quantity' column. There are navigation arrows at the bottom left of the grid.

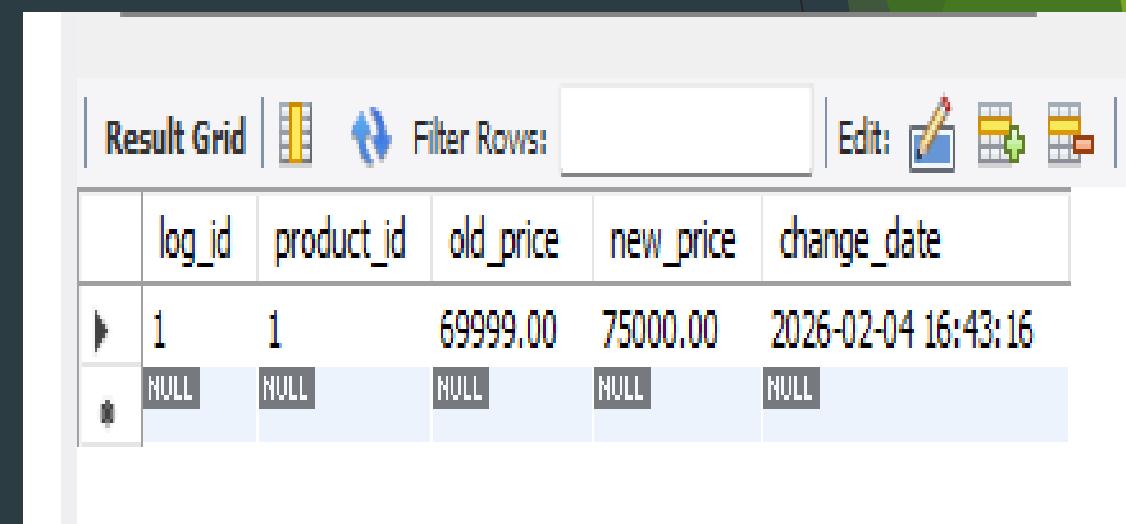
|   | stock_quantity |
|---|----------------|
| ▶ | 50             |

## SCENARIO 17: A record is created in a "Logs" table showing the old and new price.

```
CREATE TRIGGER after_price_change
AFTER UPDATE ON products
FOR EACH ROW
INSERT INTO price_log (product_id, old_price, new_price)
VALUES (OLD.product_id, OLD.price, NEW.price);

UPDATE products SET price = 75000 WHERE product_id = 1;

SELECT * FROM price_log;
```



The screenshot shows a MySQL Workbench interface with a result grid. The grid has columns for log\_id, product\_id, old\_price, new\_price, and change\_date. There is one visible row with values: log\_id 1, product\_id 1, old\_price 69999.00, new\_price 75000.00, and change\_date 2026-02-04 16:43:16. The other rows in the table are shown as empty (NULL).

|   | log_id | product_id | old_price | new_price | change_date         |
|---|--------|------------|-----------|-----------|---------------------|
| ▶ | 1      | 1          | 69999.00  | 75000.00  | 2026-02-04 16:43:16 |
| ◀ | NULL   | NULL       | NULL      | NULL      | NULL                |

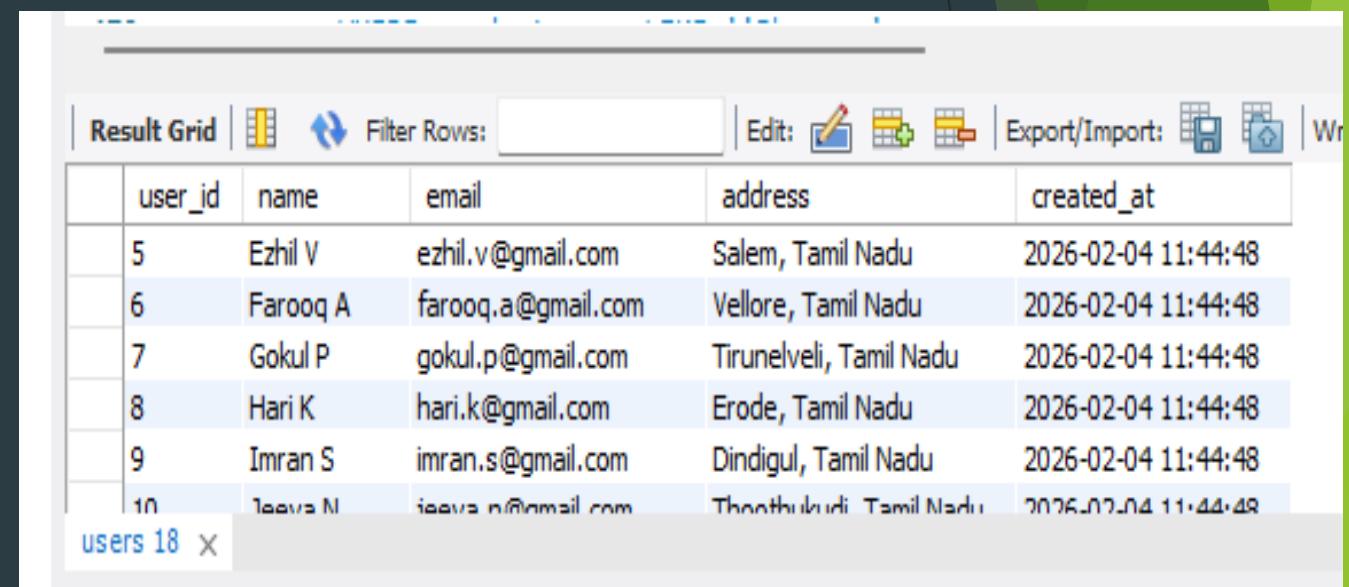
## SCENARIO 18: Name starts with 'A'

```
SELECT * FROM users
WHERE name LIKE 'A%';
```

|   | user_id | name       | email                | address             | created_at          |
|---|---------|------------|----------------------|---------------------|---------------------|
| ▶ | 1       | Arun Kumar | arun.kumar@gmail.com | Chennai, Tamil Nadu | 2026-02-04 11:44:48 |
|   | 27      | Anitha R   | anitha.r@gmail.com   | Chennai, Tamil Nadu | 2026-02-04 11:44:48 |
| ● | NULL    | NULL       | NULL                 | NULL                | NULL                |

## SCENARIO 19: Email contains 'gmail'.

```
SELECT * FROM users
WHERE email LIKE '%gmail%';
```

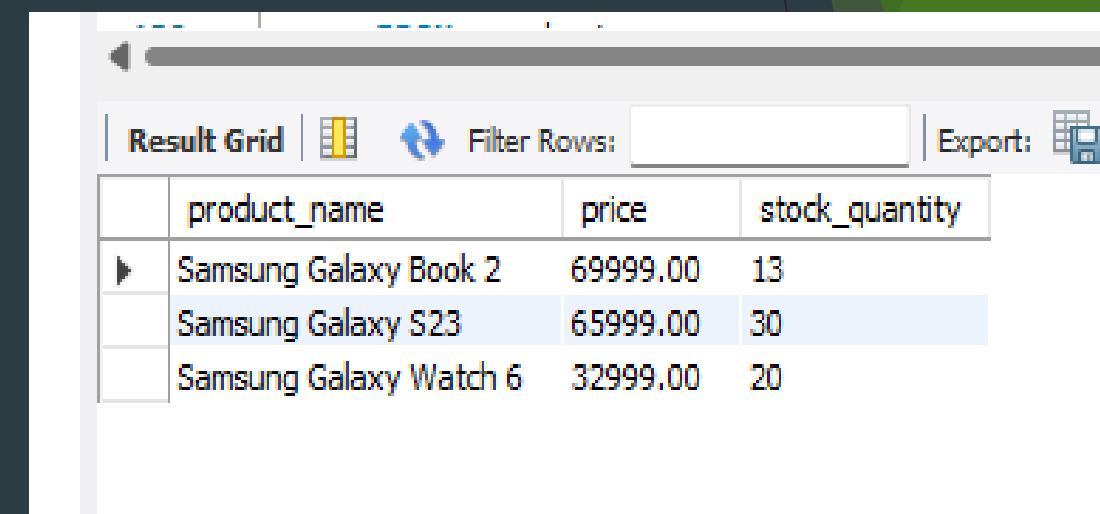


The screenshot shows a MySQL Workbench interface with a result grid titled 'users'. The grid displays 18 rows of data with columns: user\_id, name, email, address, and created\_at. The 'email' column contains entries such as 'ezhil.v@gmail.com', 'farooq.a@gmail.com', 'gokul.p@gmail.com', 'hari.k@gmail.com', 'imran.s@gmail.com', and 'jeesu.n@gmail.com', all of which contain the substring 'gmail'.

|          | user_id  | name               | email                   | address             | created_at |
|----------|----------|--------------------|-------------------------|---------------------|------------|
| 5        | Ezhil V  | ezhil.v@gmail.com  | Salem, Tamil Nadu       | 2026-02-04 11:44:48 |            |
| 6        | Farooq A | farooq.a@gmail.com | Vellore, Tamil Nadu     | 2026-02-04 11:44:48 |            |
| 7        | Gokul P  | gokul.p@gmail.com  | Tirunelveli, Tamil Nadu | 2026-02-04 11:44:48 |            |
| 8        | Hari K   | hari.k@gmail.com   | Erode, Tamil Nadu       | 2026-02-04 11:44:48 |            |
| 9        | Imran S  | imran.s@gmail.com  | Dindigul, Tamil Nadu    | 2026-02-04 11:44:48 |            |
| 10       | Jeesu N  | jeesu.n@gmail.com  | Thoothukudi, Tamil Nadu | 2026-02-04 11:44:48 |            |
| users 18 |          |                    |                         |                     |            |

**SCENARIO 20:** Find products that are **Samsungs**, cost **more than 30,000**, and have **less than 40** items in stock.

```
SELECT product_name, price, stock_quantity
FROM products
WHERE product_name LIKE 'Samsung%'
 AND price > 30000
 AND stock_quantity < 40;
```



The screenshot shows a MySQL Workbench interface with a result grid. The grid has four columns: product\_name, price, and stock\_quantity. There are three rows of data, each representing a Samsung product. The second row, which contains the Samsung Galaxy S23, is highlighted with a light blue background.

|   | product_name           | price    | stock_quantity |
|---|------------------------|----------|----------------|
| ▶ | Samsung Galaxy Book 2  | 69999.00 | 13             |
| ▶ | Samsung Galaxy S23     | 65999.00 | 30             |
| ▶ | Samsung Galaxy Watch 6 | 32999.00 | 20             |

## SCENARIO 21: Finding high-stock products using a CTE.

```
WITH HighStockProducts AS (
 SELECT product_name, stock_quantity, price
 FROM products
 WHERE stock_quantity > 50
)
SELECT * FROM HighStockProducts
WHERE price < 10000;
```

|   | product_name              | stock_quantity | price   |
|---|---------------------------|----------------|---------|
| ▶ | OnePlus Buds Z2           | 60             | 9999.00 |
|   | Realme Buds Air 5         | 80             | 4999.00 |
|   | Boat Airdopes 141         | 120            | 2999.00 |
|   | Noise ColorFit Pro 4      | 90             | 3999.00 |
|   | Fire-Boltt Ninja Call Pro | 150            | 1999.00 |
|   | Boat Docksers 450         | 110            | 1400.00 |

Result 20 ×

## SCENARIO 22: Customers often search for products by name.

```
CREATE INDEX idx_pname ON products(product_name);
```

```
SELECT * FROM products
```

```
WHERE product_name = 'iPhone 15';
```

|   | product_id | product_name | price    | stock_quantity |
|---|------------|--------------|----------|----------------|
| ▶ | 2          | iPhone 15    | 79999.00 | 18             |
| * | NULL       | NULL         | NULL     | NULL           |

## SCENARIO 23: Stored Procedure (No Parameters)

```
DELIMITER $$

CREATE PROCEDURE GetAllProducts()
BEGIN
 SELECT * FROM products;
END $$

DELIMITER ;
```

CALL GetAllProducts();

|   | product_id | product_name       | price    | stock_quantity |
|---|------------|--------------------|----------|----------------|
| ▶ | 1          | iPhone 14          | 75000.00 | 25             |
|   | 2          | iPhone 15          | 79999.00 | 18             |
|   | 3          | Samsung Galaxy S23 | 65999.00 | 30             |
|   | 4          | Samsung Galaxy A54 | 38999.00 | 40             |
|   | 5          | Redmi Note 12 Pro  | 24999.00 | 50             |
|   | 6          | OnePlus Nord CE 3  | 29999.00 | 35             |

## SCENARIO 24: Stored Procedure with Parameters (IN) to find the users using city.

```
DELIMITER $$

CREATE PROCEDURE GetUsersByCity(IN city_name VARCHAR(100))
BEGIN
 SELECT * FROM users
 WHERE address LIKE CONCAT('%', city_name, '%');
END $$

DELIMITER ;

CALL GetUsersByCity('coimbatore');
```

|   | user_id | name      | email               | address                | created_at          |
|---|---------|-----------|---------------------|------------------------|---------------------|
| ▶ | 2       | Balaji S  | balaji.s@gmail.com  | Coimbatore, Tamil Nadu | 2026-02-04 11:44:48 |
|   | 28      | Bhavani S | bhavani.s@gmail.com | Coimbatore, Tamil Nadu | 2026-02-04 11:44:48 |

Thank You