SQL Lab One:

```
-- Create Sales table

CREATE TABLE Sales (
   sale_id INT PRIMARY KEY,
   product_id INT,
   quantity_sold INT,
   sale_date DATE,
   total_price DECIMAL(10, 2)
);

-- Insert sample data into Sales table

INSERT INTO Sales (sale_id, product_id, quantity_sold, sale_date, total_price) VALUES
(1, 101, 5, '2024-01-01', 2500.00),
(2, 102, 3, '2024-01-02', 900.00),
(3, 103, 2, '2024-01-02', 60.00),
(4, 104, 4, '2024-01-03', 80.00),
(5, 105, 6, '2024-01-03', 90.00);
```

Output:

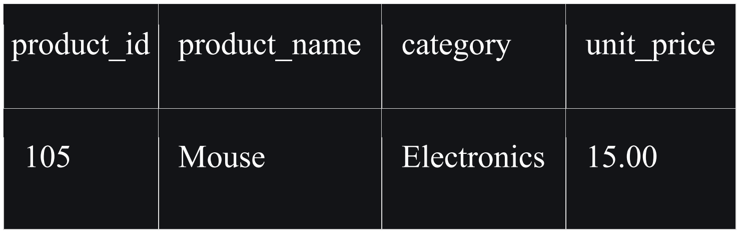
sale_id	product_id	quantity_sold	sale_date	total_price
1	101	5	2024-01-01	2500.00
2	102	3	2024-01-02	900.00
3	103	2	2024-01-02	60.00
4	104	4	2024-01-03	80.00

sale_id	product_id	quantity_sold	sale_date	total_price
5	105	6	2024-01-03	90.00

```
-- Create Products table

CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    unit_price DECIMAL(10, 2)
);

-- Insert sample data into Products table

INSERT INTO Products (product_id, product_name, category, unit_price) VALUES
(101, 'Laptop', 'Electronics', 500.00),
(102, 'Smartphone', 'Electronics', 300.00),
(103, 'Headphones', 'Electronics', 30.00),
(104, 'Keyboard', 'Electronics', 20.00),
(105, 'Mouse', 'Electronics', 15.00);
```

Output:

product_id	product_name	category	unit_price
101	Laptop	Electronics	500.00
102	Smartphone	Electronics	300.00
103	Headphones	Electronics	30.00
104	Keyboard	Electronics	20.00



product_id	product_name	category	unit_price
105	Mouse	Electronics	15.00

SQL Practice Exercises for Beginners

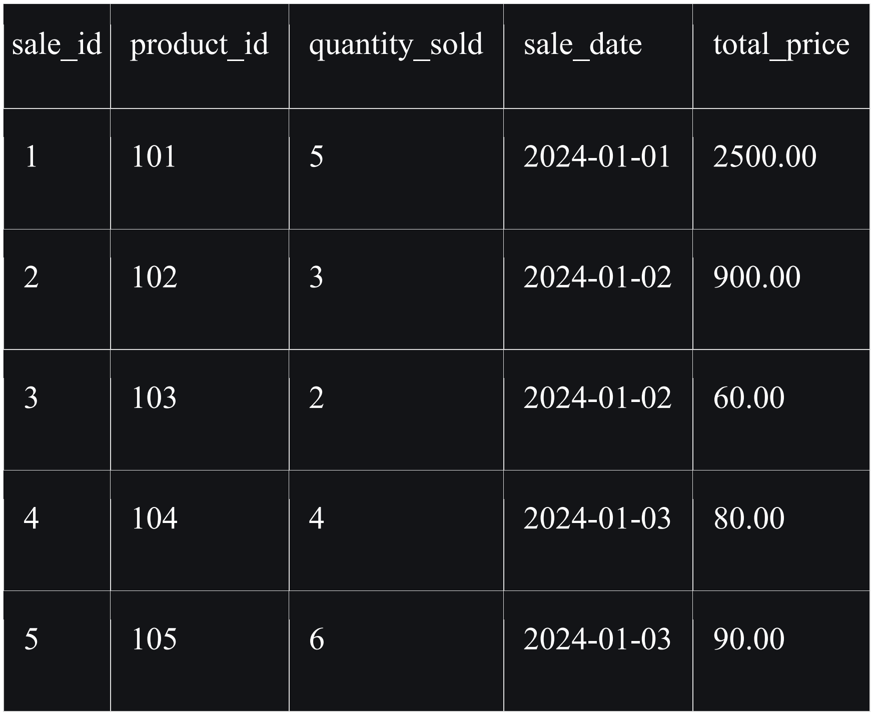
This hands-on approach provides a practical environment for beginners to experiment with various SQL commands, gaining confidence through real-world scenarios. By working through these exercises, newcomers can solidify their understanding of fundamental concepts like data retrieval, filtering, and manipulation, laying a strong foundation for their SQL journey.

1. Retrieve all columns from the Sales table.

Query:

```
SELECT * FROM Sales;
```

Output:



sale_id	product_id	quantity_sold	sale_date	total_price
1	101	5	2024-01-01	2500.00
2	102	3	2024-01-02	900.00
3	103	2	2024-01-02	60.00
4	104	4	2024-01-03	80.00
5	105	6	2024-01-03	90.00

Explanation:

This SQL query selects all columns from the Sales table, denoted by the asterisk (*) wildcard. It retrieves every row and all associated columns from the Sales table.

2. Retrieve the product_name and unit_price from the Products table.

Query:

```
SELECT product_name, unit_price FROM Products;
```

Explanation:

This SQL query selects the product_name and unit_price columns from the Products table. It retrieves every row but only the specified columns, which are product_name and unit_price.

3. Retrieve the sale_id and sale_date from the Sales table.

Query:

```
SELECT sale_id, sale_date FROM Sales;
```

Explanation:

This SQL query selects the sale_id and sale_date columns from the Sales table. It retrieves every row but only the specified columns, which are sale_id and sale_date.

4. Filter the Sales table to show only sales with a total_price greater than $100.

Query:

```
SELECT * FROM Sales WHERE total_price > 100;
```

Explanation:

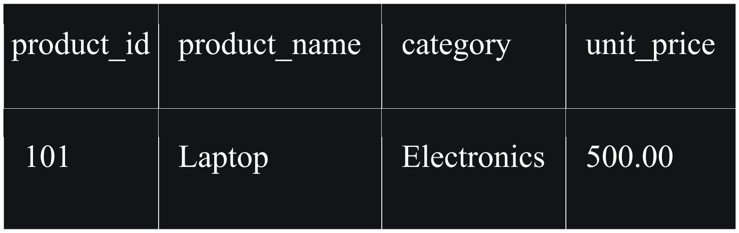
This SQL query selects all columns from the Sales table but only returns rows where the total_price column is greater than 100. It filters out sales with a total_price less than or equal to $100.

5. Filter the Products table to show only products in the ‘Electronics’ category.

Query:

```
SELECT * FROM Products WHERE category = 'Electronics';
```

Output:



product_id	product_name	category	unit_price
101	Laptop	Electronics	500.00

product_id	product_name	category	unit_price
102	Smartphone	Electronics	300.00
103	Headphones	Electronics	30.00
104	Keyboard	Electronics	20.00
105	Mouse	Electronics	15.00

Explanation:

This SQL query selects all columns from the Products table but only returns rows where the category column equals 'Electronics'. It filters out products that do not belong to the 'Electronics' category.

6. Retrieve the sale_id and total_price from the Sales table for sales made on January 3, 2024.

Query:

```
SELECT sale_id, total_price FROM Sales
WHERE sale_date = '2024-01-03';
```

Output:

sale_id	total_price
4	80.00
5	90.00

Explanation:

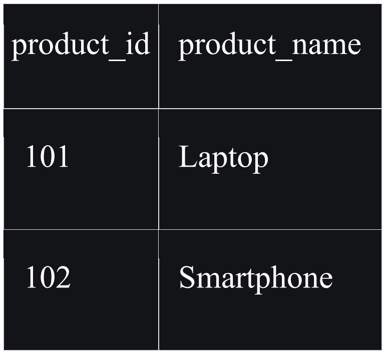
This SQL query selects the sale_id and total_price columns from the Sales table but only returns rows where the sale_date is equal to '2024-01-03'. It filters out sales made on any other date.

7. Retrieve the product_id and product_name from the Products table for products with a unit_price greater than $100.

Query:

```
SELECT product_id, product_name FROM Products
WHERE unit_price > 100;
```

Output:



product_id	product_name
101	Laptop
102	Smartphone

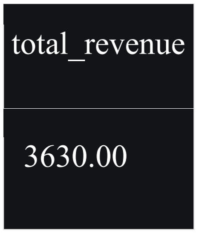
Explanation:

This SQL query selects the product_id and product_name columns from the Products table but only returns rows where the unit_price is greater than $100. It filters out products with a unit_price less than or equal to $100.

8. Calculate the total revenue generated from all sales in the Sales table.

Query:

```
SELECT SUM(total_price) AS total_revenue
FROM Sales;
```



total_revenue
3630.00

Explanation:

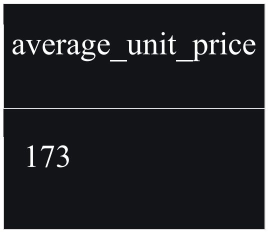
This SQL query calculates the total revenue generated from all sales by summing up the total_price column in the Sales table using the SUM() function.

9. Calculate the average unit_price of products in the Products table.

Query:

```
SELECT AVG(unit_price) AS average_unit_price FROM Products;
```

Output:



average_unit_price
173

Explanation:

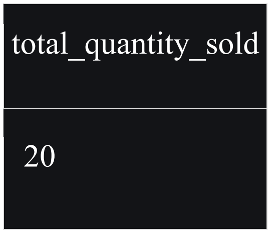
This SQL query calculates the average unit_price of products by averaging the values in the unit_price column in the Products table using the AVG() function.

10. Calculate the total quantity_sold from the Sales table.

Query:

```
SELECT SUM(quantity_sold) AS total_quantity_sold FROM Sales;
```

Output:



total_quantity_sold
20

Explanation:

This SQL query calculates the total quantity_sold by summing up the quantity_sold column in the Sales table using the SUM() function.

11. Retrieve the sale_id, product_id, and total_price from the Sales table for sales with a quantity_sold greater than 4.

Query:

```
SELECT sale_id, product_id, total_price FROM Sales
WHERE quantity_sold > 4;
```

Output:

sale_id	product_id	total_price
1	101	2500.00
5	105	90.00

Explanation:

This SQL query selects the sale_id, product_id, and total_price columns from the Sales table but only returns rows where the quantity_sold is greater than 4.

12. Retrieve the product_name and unit_price from the Products table, ordering the results by unit_price in descending order.

Query:

```
SELECT product_name, unit_price
FROM Products
ORDER BY unit_price DESC;
```

Output:

product_name	unit_price
Laptop	500.00
Smartphone	300.00
Headphones	30.00
Keyboard	20.00
Mouse	15.00

Explanation:

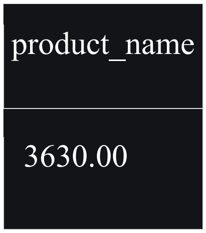
This SQL query selects the product_name and unit_price columns from the Products table and orders the results by unit_price in descending order using the ORDER BY clause with the DESC keyword.

13. Retrieve the total_price of all sales, rounding the values to two decimal places.

Query:

SELECT ROUND(SUM(total_price), 2) AS total_sales FROM Sales;

Output:



product_name
3630.00

Explanation:

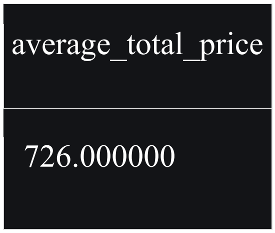
This SQL query calculates the total sales revenu by summing up the total_price column in the Sales table and rounds the result to two decimal places using the ROUND() function.

14. Calculate the average total_price of sales in the Sales table.

Query:

SELECT AVG(total_price) AS average_total_price FROM Sales;

Output:



average_total_price
726.000000

Explanation:

This SQL query calculates the average total_price of sales by averaging the values in the total_price column in the Sales table using the AVG() function.

15. Calculate the total revenue generated from sales of products in the ‘Electronics’ category.

Query:

```
SELECT SUM(Sales.total_price) AS total_revenue FROM Sales
JOIN Products ON Sales.product_id = Products.product_id
WHERE Products.category = 'Electronics';
```

Output:

total_revenue
3630.00

Explanation:

This SQL query calculates the total revenue generated from sales of products in the ‘Electronics’ category by joining the Sales table with the Products table on the product_id column and filtering sales for products in the ‘Electronics’ category.

16. Retrieve the product_name and unit_price from the Products table, filtering the unit_price to show only values between $20 and $600.

Query:

```
SELECT product_name, unit_price FROM Products
WHERE unit_price BETWEEN 20 AND 600;
```

Output:

product_name	unit_price
Laptop	500.00
Smartphone	300.00
Headphones	30.00
Keyboard	20.00

Explanation:

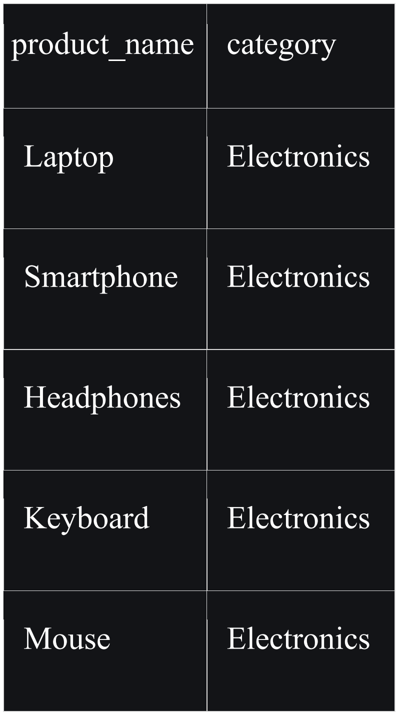
This SQL query selects the product_name and unit_price columns from the Products table but only returns rows where the unit_price falls within the range of $50 and $200 using the BETWEEN operator.

17. Retrieve the product_name and category from the Products table, ordering the results by category in ascending order.

Query:

```
SELECT product_name, category FROM Products
ORDER BY category ASC;
```

Output:



product_name	category
Laptop	Electronics
Smartphone	Electronics
Headphones	Electronics
Keyboard	Electronics
Mouse	Electronics

Explanation:

This SQL query selects the product_name and category columns from the Products table and orders the results by category in ascending order using the ORDER BY clause with the ASC keyword.

18. Calculate the total quantity_sold of products in the ‘Electronics’ category.

Query:

```
SELECT SUM(quantity_sold) AS total_quantity_sold
FROM Sales
```

```
JOIN Products ON Sales.product_id = Products.product_id
WHERE Products.category = 'Electronics';
```

Output:

total_quantity_sold
20

Explanation:

This SQL query calculates the total quantity_sold of products in the 'Electronics' category by joining the Sales table with the Products table on the product_id column and filtering sales for products in the 'Electronics' category.

19. Retrieve the product_name and total_price from the Sales table, calculating the total_price as quantity_sold multiplied by unit_price.

Query:

```
SELECT product_name, quantity_sold * unit_price AS total_price FROM Sales
JOIN Products ON Sales.product_id = Products.product_id;
```

Output:

product_name	total_price
Laptop	2500.00
Smartphone	900.00
Headphones	60.00
Keyboard	80.00
Mouse	90.00

Explanation:

This SQL query retrieves the product_name from the Sales table and calculates the total_price by multiplying quantity_sold by unit_price, joining the Sales table with the Products table on the product_id column.

SQL Practice Exercises for Intermediate

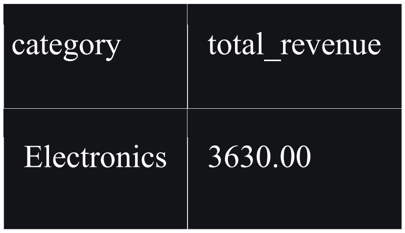
These exercises are designed to challenge you beyond basic queries, delving into more complex data manipulation and analysis. By tackling these problems, you'll solidify your understanding of advanced SQL concepts like joins, subqueries, functions, and window functions, ultimately boosting your ability to work with real-world data scenarios effectively.

1. Calculate the total revenue generated from sales for each product category.

Query:

```
SELECT p.category, SUM(s.total_price) AS total_revenue FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.category;
```

Output:



category	total_revenue
Electronics	3630.00

Explanation:

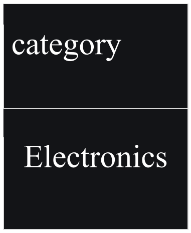
This query joins the Sales and Products tables on the product_id column, groups the results by product category, and calculates the total revenue for each category by summing up the total_price.

2. Find the product category with the highest average unit price.

Query:

```
SELECT category
FROM Products
GROUP BY category
ORDER BY AVG(unit_price) DESC
LIMIT 1;
```

Output:



category
Electronics

Explanation:

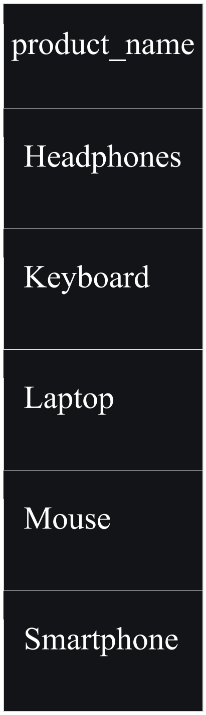
This query groups products by category, calculates the average unit price for each category, orders the results by the average unit price in descending order, and selects the top category with the highest average unit price using the LIMIT clause.

3. Identify products with total sales exceeding 30.

Query:

```
SELECT p.product_name
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.product_name
HAVING SUM(s.total_price) > 30;
```

Output:



product_name
Headphones
Keyboard
Laptop
Mouse
Smartphone

Explanation:

This query joins the Sales and Products tables on the product_id column, groups the results by product name, calculates the total sales revenue for each product, and selects products with total sales exceeding 30 using the HAVING clause.

4. Count the number of sales made in each month.

Query:

```
SELECT DATE_FORMAT(s.sale_date, '%Y-%m') AS month, COUNT(*) AS sales_count
FROM Sales s
GROUP BY month;
```

Output:

month	sales_count
2024-01	5

Explanation:

This query formats the sale_date column to extract the month and year, groups the results by month, and counts the number of sales made in each month.

5. Determine the average quantity sold for products with a unit price greater than $100.

Query:

```
SELECT AVG(s.quantity_sold) AS average_quantity_sold
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
WHERE p.unit_price > 100;
```

Output:

average_quantity_sold
4.0000

Explanation:

This query joins the Sales and Products tables on the product_id column, filters products with a unit price greater than $100, and calculates the average quantity sold for those products.

6. Retrieve the product name and total sales revenue for each product.

Query:

```
SELECT p.product_name, SUM(s.total_price) AS total_revenue
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.product_name;
```

Output:

product_name	total_revenue
Laptop	2500.00
Smartphone	900.00
Headphones	60.00
Keyboard	80.00
Mouse	90.00

Explanation:

This query joins the Sales and Products tables on the product_id column, groups the results by product name, and calculates the total sales revenue for each product.

7. List all sales along with the corresponding product names.

Query:

```
SELECT s.sale_id, p.product_name
FROM Sales s
JOIN Products p ON s.product_id = p.product_id;
```

Output:

sale_id	product_name
1	Laptop
2	Smartphone
3	Headphones
4	Keyboard
5	Mouse

Explanation:

This query joins the Sales and Products tables on the product_id column and retrieves the sale_id and product_name for each sale.

8. Retrieve the product name and total sales revenue for each product.

Query:

```
SELECT p.category,
       SUM(s.total_price) AS category_revenue,
       (SUM(s.total_price) / (SELECT SUM(total_price) FROM Sales)) * 100 AS
revenue_percentage
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.category
ORDER BY revenue_percentage DESC
LIMIT 3;
```

Output:

category	category_revenue	revenue_percentage
Electronics	3630.00	100.000000

Explanation:

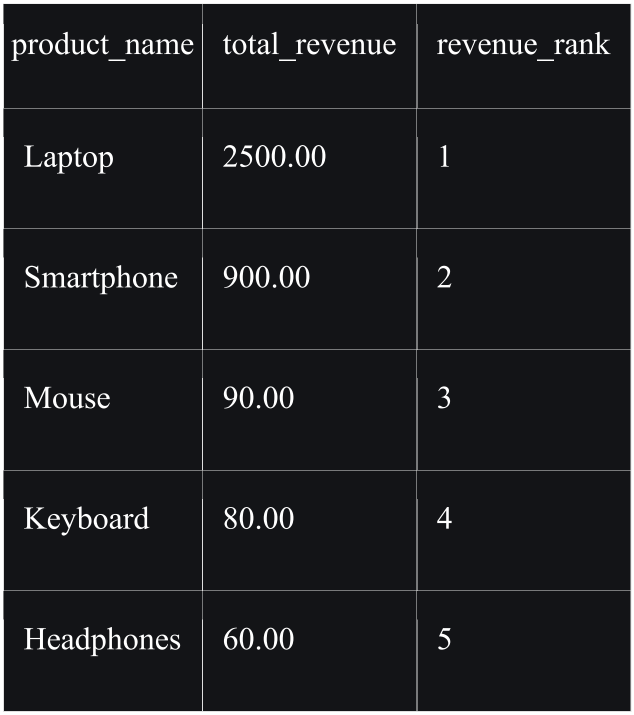
This query will give you the top three product categories contributing to the highest percentage of total revenue generated from sales. However, if you only have one category (Electronics) as in the provided sample data, it will be the only result.

9. Rank products based on total sales revenue.

Query:

```
SELECT p.product_name, SUM(s.total_price) AS total_revenue,
    RANK() OVER (ORDER BY SUM(s.total_price) DESC) AS revenue_rank
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.product_name;
```

Output:



product_name	total_revenue	revenue_rank
Laptop	2500.00	1
Smartphone	900.00	2
Mouse	90.00	3
Keyboard	80.00	4
Headphones	60.00	5

Explanation:

This query joins the Sales and Products tables on the product_id column, groups the results by product name, calculates the total sales revenue for each product, and ranks products based on total sales revenue using the RANK() window function.

10. Calculate the running total revenue for each product category.

Query:

```
SELECT p.category, p.product_name, s.sale_date,
       SUM(s.total_price) OVER (PARTITION BY p.category ORDER BY s.sale_date) AS
running_total_revenue
FROM Sales s
JOIN Products p ON s.product_id = p.product_id;
```

Output:

category	product_name	sale_date	running_total_revenue
Electronics	Laptop	2024-01-01	2500.00
Electronics	Smartphone	2024-01-02	3460.00
Electronics	Headphones	2024-01-02	3460.00
Electronics	Keyboard	2024-01-03	3630.00
Electronics	Mouse	2024-01-03	3630.00

Explanation:

This query joins the Sales and Products tables on the product_id column, partitions the results by product category, orders the results by sale date, and calculates the running total revenue for each product category using the SUM() window function.

11. Categorize sales as “High”, “Medium”, or “Low” based on total price (e.g., > $200 is High, $100-$200 is Medium, < $100 is Low).

Query:

```
SELECT sale_id,
       CASE
           WHEN total_price > 200 THEN 'High'
           WHEN total_price BETWEEN 100 AND 200 THEN 'Medium'
           ELSE 'Low'
       END AS sales_category
FROM Sales;
```

Output:

sale_id	sales_category
1	High
2	High
3	Low
4	Low
5	Low

Explanation:

This query categorizes sales based on total price using a CASE statement. Sales with a total price greater than $200 are categorized as "High", sales with a total price between $100 and $200 are categorized as "Medium", and sales with a total price less than $100 are categorized as "Low".

12. Identify sales where the quantity sold is greater than the average quantity sold.

Query:

```
SELECT * FROM Sales
WHERE quantity_sold > (SELECT AVG(quantity_sold) FROM Sales);
```

Output:

sale_id	product_id	quantity_sold	sale_date	total_price
1	101	5	2024-01-01	2500.00
5	105	6	2024-01-03	90.00

Explanation:

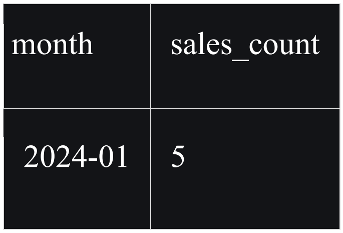
This query selects all sales where the quantity sold is greater than the average quantity sold across all sales in the Sales table.

13. Extract the month and year from the sale date and count the number of sales for each month.

Query:

```
SELECT CONCAT(YEAR(sale_date), '-', LPAD(MONTH(sale_date), 2, '0')) AS month,
       COUNT(*) AS sales_count
FROM Sales
GROUP BY YEAR(sale_date), MONTH(sale_date);
```

Output:



month	sales_count
2024-01	5

Explanation:

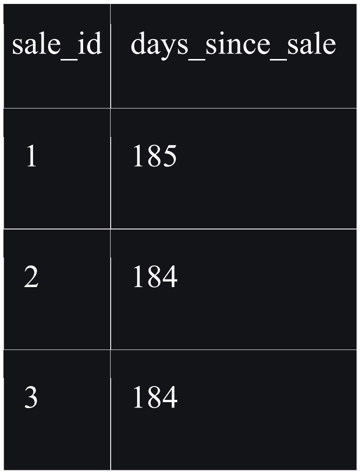
This query selects all sales where the quantity sold is greater than the average quantity sold across all sales in the Sales table.

14. Calculate the number of days between the current date and the sale date for each sale.

Query:

```
SELECT sale_id, DATEDIFF(NOW(), sale_date) AS days_since_sale
FROM Sales;
```

Output:



sale_id	days_since_sale
1	185
2	184
3	184

sale_id	days_since_sale
4	183
5	183

Explanation:

This query calculates the number of days between the current date and the sale date for each sale using the DATEDIFF function.

15. Identify sales made during weekdays versus weekends.

Query:

```
SELECT sale_id,
       CASE
           WHEN DAYOFWEEK(sale_date) IN (1, 7) THEN 'Weekend'
           ELSE 'Weekday'
       END AS day_type
FROM Sales;
```

Output:

sale_id	day_type
1	Weekday
2	Weekday
3	Weekday
4	Weekend
5	Weekend

Explanation:

This query categorizes sales based on the day of the week using the DAYOFWEEK function. Sales made on Sunday (1) or Saturday (7) are categorized as "Weekend", while sales made on other days are categorized as "Weekday".

SQL Practice Exercises for Advanced

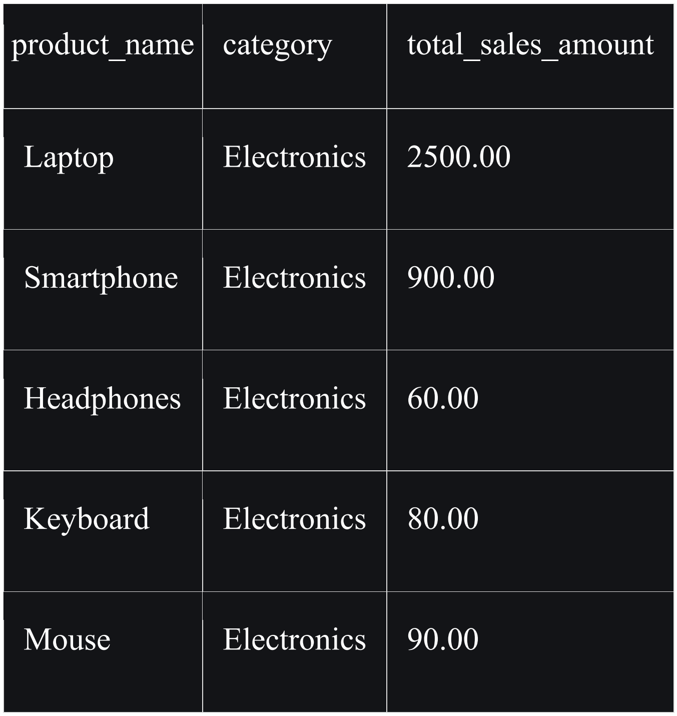
This section likely dives deeper into complex queries, delving into advanced features like window functions, self-joins, and intricate data manipulation techniques. By tackling these challenging exercises, users can refine their SQL skills and tackle real-world data analysis scenarios with greater confidence and efficiency.

1. Write a query to create a view named Total_Sales that displays the total sales amount for each product along with their names and categories.

Query:

```
CREATE VIEW Total_Sales AS
SELECT p.product_name, p.category, SUM(s.total_price) AS total_sales_amount
FROM Products p
JOIN Sales s ON p.product_id = s.product_id
GROUP BY p.product_name, p.category;
SELECT * FROM Total_Sales;
```

Output:



product_name	category	total_sales_amount
Laptop	Electronics	2500.00
Smartphone	Electronics	900.00
Headphones	Electronics	60.00
Keyboard	Electronics	80.00
Mouse	Electronics	90.00

Explanation:

This query creates a view named Total_Sales that displays the total sales amount for each product along with their names and categories.

2. Retrieve the product details (name, category, unit price) for products that have a quantity sold greater than the average quantity sold across all products.

Query:

```
SELECT product_name, category, unit_price
FROM Products
WHERE product_id IN (
    SELECT product_id
    FROM Sales
    GROUP BY product_id
    HAVING SUM(quantity_sold) > (SELECT AVG(quantity_sold) FROM Sales)
);
```

Output:

product_name	category	unit_price
Laptop	Electronics	500.00
Mouse	Electronics	15.00

Explanation:

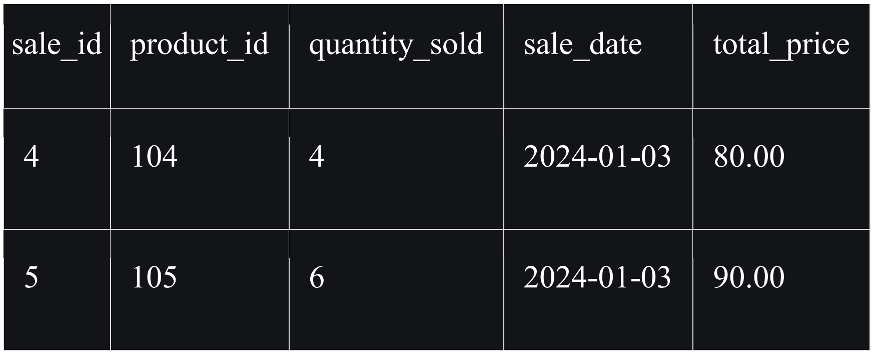
This query retrieves the product details (name, category, unit price) for products that have a quantity sold greater than the average quantity sold across all products.

3. Explain the significance of indexing in SQL databases and provide an example scenario where indexing could significantly improve query performance in the given schema.

Query:

```
-- Create an index on the sale_date column
CREATE INDEX idx_sale_date ON Sales (sale_date);

-- Query with indexing
SELECT *
FROM Sales
WHERE sale_date = '2024-01-03';
```

Output:



sale_id	product_id	quantity_sold	sale_date	total_price
4	104	4	2024-01-03	80.00
5	105	6	2024-01-03	90.00

Explanation:

With an index on the sale_date column, the database can quickly locate the rows that match the specified date without scanning the entire table. The index allows for efficient lookup of rows based on the sale_date value, resulting in improved query performance.

4. Add a foreign key constraint to the Sales table that references the product_id column in the Products table.

Query:

```
ALTER TABLE Sales
ADD CONSTRAINT fk_product_id
FOREIGN KEY (product_id)
REFERENCES Products(product_id);
```

Output:

No output is generated, but the constraint is applied to the table.

Explanation:

This query adds a foreign key constraint to the Sales table that references the product_id column in the Products table, ensuring referential integrity between the two tables.

5. Create a view named Top_Products that lists the top 3 products based on the total quantity sold.

Query:

```
CREATE VIEW Top_Products AS
SELECT p.product_name, SUM(s.quantity_sold) AS total_quantity_sold
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.product_name
```

```
ORDER BY total_quantity_sold DESC
LIMIT 3;
```

Output:

product_name	total_quantity_sold
Mouse	6
Laptop	5
Keyboard	4

Explanation:

This query creates a view named Top_Products that lists the top 3 products based on the total quantity sold.

6. Implement a transaction that deducts the quantity sold from the Products table when a sale is made in the Sales table, ensuring that both operations are either committed or rolled back together.

Query:

```
START TRANSACTION; -- Begin the transaction

-- Deduct the quantity sold from the Products table
UPDATE Products p
JOIN Sales s ON p.product_id = s.product_id
SET p.quantity_in_stock = p.quantity_in_stock - s.quantity_sold;

-- Check if any negative quantities would result from the update
SELECT COUNT(*) INTO @negative_count
FROM Products
WHERE quantity_in_stock < 0;

-- If any negative quantities would result, rollback the transaction
IF @negative_count > 0 THEN
    ROLLBACK;
    SELECT 'Transaction rolled back due to insufficient stock.' AS Message;
```

```
ELSE
   COMMIT; -- Commit the transaction if no negative quantities would result
   SELECT 'Transaction committed successfully.' AS Message;
END IF;

START TRANSACTION;
UPDATE Products SET quantity_in_stock = 10 WHERE product_id = 101;
INSERT INTO Sales (product_id, quantity_sold) VALUES (101, 5);
COMMIT;
```

Output:

```
Transaction committed successfully.
```

Explanation:

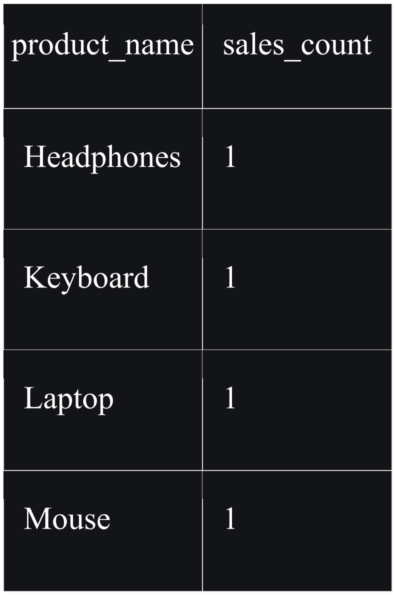
The quantity in stock for product with product_id 101 should be updated to 5.The transaction should be committed successfully.

7. Create a query that lists the product names along with their corresponding sales count.

Query:

```
SELECT p.product_name, COUNT(s.sale_id) AS sales_count
FROM Products p
LEFT JOIN Sales s ON p.product_id = s.product_id
GROUP BY p.product_name;
```

Output:



product_name	sales_count
Headphones	1
Keyboard	1
Laptop	1
Mouse	1

product_name	sales_count
Smartphone	1

Explanation:

This query selects the product names from the Products table and counts the number of sales (using the COUNT() function) for each product by joining the Sales table on the product_id. The results are grouped by product name using the GROUP BY clause.

8. Write a query to find all sales where the total price is greater than the average total price of all sales.

Query:

```
SELECT * FROM Sales
WHERE total_price > (SELECT AVG(total_price) FROM Sales);
```

Output:

sale_id	product_id	quantity_sold	sale_date	total_price
1	101	5	2024-01-01	2500.00
2	102	3	2024-01-02	900.00

Explanation:

The subquery (SELECT AVG(total_price) FROM Sales) calculates the average total price of all sales. The main query selects all columns from the Sales table where the total price is greater than the average total price obtained from the subquery.

9. Analyze the performance implications of indexing the sale_date column in the Sales table, considering the types of queries commonly executed against this column.

Query:

```
-- Query without indexing
EXPLAIN ANALYZE
SELECT * FROM Sales
WHERE sale_date = '2024-01-01';
```

```
-- Query with indexing
CREATE INDEX idx_sale_date ON Sales (sale_date);

EXPLAIN ANALYZE
SELECT * FROM Sales
WHERE sale_date = '2024-01-01';
```

Output:

Query without Indexing:

Operation	Details
Filter: (sales.sale_date = DATE’2024-01-01′)	(cost=0.75 rows=1) (actual time=0.020..0.031 rows=1 loops=1)
Table scan on Sales	(cost=0.75 rows=5) (actual time=0.015..0.021 rows=5 loops=1)

Query with Indexing:

Operation	Details
Index lookup on Sales using idx_sale_date (sale_date=DATE’2024-01-01′)	(cost=0.35 rows=1) (actual time=0.024..0.024 rows=1 loops=1)

This format clearly displays the operations and details of the query execution plan before and after indexing.

Explanation:

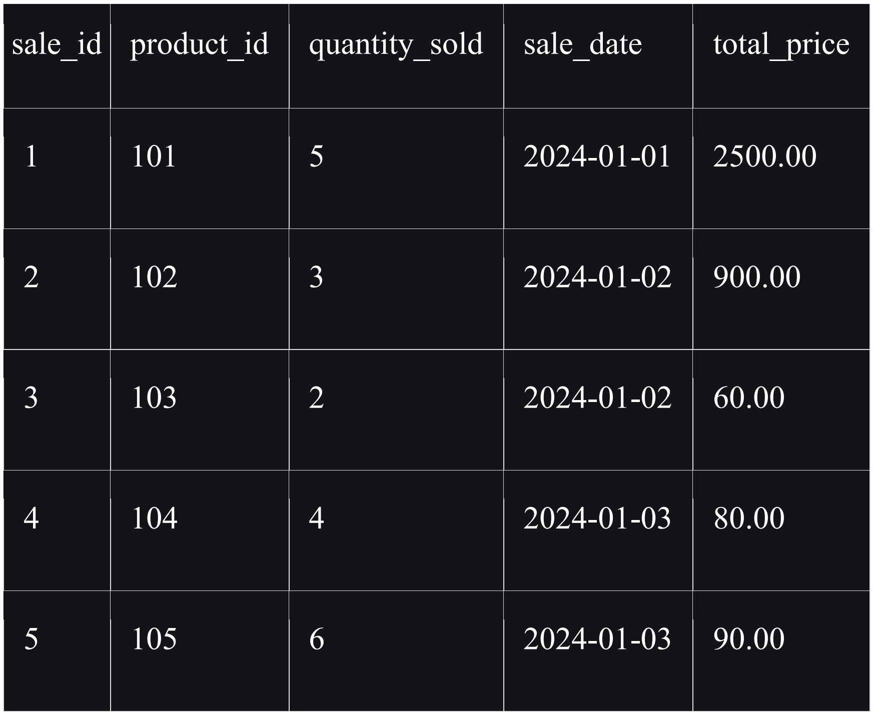
Without indexing, the query performs a full table scan, filtering rows based on the sale date, which is less efficient. With indexing, the query uses the index to quickly locate the relevant rows, significantly improving query performance.

10. Add a check constraint to the quantity_sold column in the Sales table to ensure that the quantity sold is always greater than zero.

Query:

```
ALTER TABLE Sales
ADD CONSTRAINT chk_quantity_sold CHECK (quantity_sold > 0);

-- Query to check if the constraint is applied successfully
SELECT * FROM Sales;
```

Output:



sale_id	product_id	quantity_sold	sale_date	total_price
1	101	5	2024-01-01	2500.00
2	102	3	2024-01-02	900.00
3	103	2	2024-01-02	60.00
4	104	4	2024-01-03	80.00
5	105	6	2024-01-03	90.00

Explanation:

All rows in the Sales table meet the condition of the check constraint, as each quantity_sold value is greater than zero.

11. Create a view named Product_Sales_Info that displays product details along with the total number of sales made for each product.

Query:

```
CREATE VIEW Product_Sales_Info AS
SELECT
    p.product_id,
    p.product_name,
    p.category,
    p.unit_price,
    COUNT(s.sale_id) AS total_sales
FROM
    Products p
```

```
LEFT JOIN
    Sales s ON p.product_id = s.product_id
GROUP BY
    p.product_id, p.product_name, p.category, p.unit_price;
```

Output:

product_id	product_name	category	unit_price	total_sales
101	Laptop	Electronics	500.00	1
102	Smartphone	Electronics	300.00	1
103	Headphones	Electronics	30.00	1
104	Keyboard	Electronics	20.00	1
105	Mouse	Electronics	15.00	1

Explanation:

This view provides a concise and organized way to view product details alongside their respective sales information, facilitating analysis and reporting tasks.

12. Develop a stored procedure named Update_Unit_Price that updates the unit price of a product in the Products table based on the provided product_id.

Query:

```
DELIMITER //

CREATE PROCEDURE Update_Unit_Price (
    IN p_product_id INT,
    IN p_new_price DECIMAL(10, 2)
)
BEGIN
    UPDATE Products
    SET unit_price = p_new_price
    WHERE product_id = p_product_id;
```

```
END //

DELIMITER ;
```

Output:

There is no direct output shown here as this is a stored procedure definition

Explanation:

The above SQL code creates a stored procedure named Update_Unit_Price. This stored procedure takes two parameters: p_product_id (the product ID for which the unit price needs to be updated) and p_new_price (the new unit price to set).

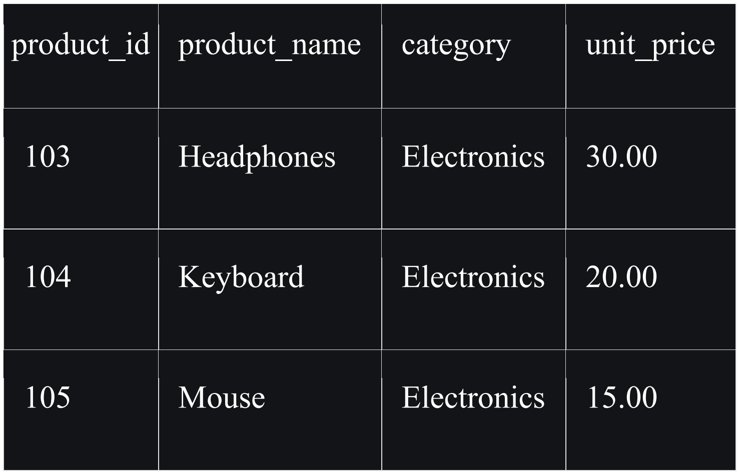
13. Implement a transaction that inserts a new product into the Products table and then adds a corresponding sale record into the Sales table, ensuring that both operations are either fully completed or fully rolled back.

Query:

```
CREATE PROCEDURE Update_Unit_Price (
    @product_id INT,
    @new_unit_price DECIMAL(10, 2)
)
AS
BEGIN
    UPDATE Products
    SET unit_price = @new_unit_price
    WHERE product_id = @product_id;
END;

EXEC Update_Unit_Price @product_id = 101, @new_unit_price = 550.00;
SELECT * FROM Products;
```

Output:

product_id	product_name	category	unit_price
101	Laptop	Electronics	550.00
102	Smartphone	Electronics	300.00



product_id	product_name	category	unit_price
103	Headphones	Electronics	30.00
104	Keyboard	Electronics	20.00
105	Mouse	Electronics	15.00

Explanation:

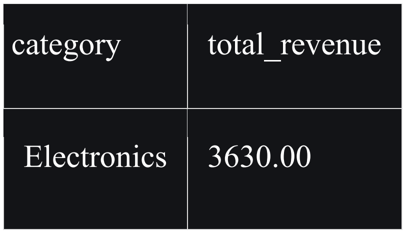
This will update the unit price of the product with product_id 101 to 550.00 in the Products table.

14. Write a query that calculates the total revenue generated from each category of products for the year 2024.

Query:

```
SELECT p.category, SUM(s.total_price) AS total_revenue
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
WHERE strftime('%Y', s.sale_date) = '2024'
GROUP BY p.category;
```

Output:



category	total_revenue
Electronics	3630.00

Explanation:

When you execute this query, you will get the total revenue generated from each category of products for the year 2024.