1. Retrieve the order date and day of the week for all orders.

Retrieve the order date and day of the week for all orders

```
SELECT orderDate , DAYOFWEEK(orderDate)
FROM orders;
```

2.List the product names and order dates for products ordered on a Saturday

```
select p.productName
from orders as o left join orderdetails as od using(orderNumber) left join
products as p using(productCode)
where DayName(o.orderDate) like "Saturday"
```

3. Find the number of orders placed on each day of the week

```
select count(*) , DayName(orderDate)
from orders
group by DayName(orderDate)
```

4. Retrieve the customer names and their first order date.

```
select c.customerName, Min(o.orderDate)
from customers as c join orders as o using(customerNumber)
group by c.customerName
```

5. Calculate the total payments received for each customer. Include the customer name and the total payments.

```
select c.customerName , sum(amount) as total_payments
```

```
from customers as c join payments as pa using(customerNumber)
group by c.customerName
```

6. Retrieve the count of orders for each year, and include a grand total count. Display the year and the corresponding order count.

```
select year(orderDate) , count(*)
from orders
group by year(orderDate)
with rollup
```

1. For each year and month, find the total number of orders placed. Additionally, provide a grand total for all orders. Display the results with the count of orders, year, and month.

```
SELECT
   YEAR(orderDate) AS OrderYear,
   MONTH(orderDate) AS OrderMonth,
   COUNT(*) AS OrderCount
FROM orders
GROUP BY OrderYear, OrderMonth WITH ROLLUP;
```

7. Retrieve the total value of products in stock, considering the quantity in stock and the price each. Display the product name and the corresponding total value. Additionally, include a grand total row that represents the overall total value of all products.

```
SELECT productName, sum(quantityInStock * buyPrice) as total
FROM products
GROUP BY productName
WITH ROLLUP;
```

8. Retrieve the products with a total value exceeding $15M. Display the product name and the corresponding total value. Additionally, include a grand total row that represents the overall total value of all products.

```
SELECT productName, sum(quantityInStock * buyPrice) as total
FROM products
GROUP BY productName
WITH ROLLUP
HAVING total > 15000000;
```

9. Retrieve the total quantity of products sold and the total sales amount for each country. Display the country, the total quantity of products sold, and the total sales amount ((quantityOrdered * priceEach)) . Include only countries where the total quantity sold is greater than 2500. Sort the results by the total sales amount in ascending order.

```
SELECT
   country,
   SUM(quantityOrdered) AS total_sold,
   SUM(quantityOrdered * priceEach) AS total_sales
FROM orders
JOIN customers ON orders.customerNumber = customers.customerNumber
JOIN orderdetails ON orders.orderNumber = orderdetails.orderNumber
GROUP BY country
HAVING total_sold > 2500
ORDER BY total_sales ASC;
```

10.Retrieve the number of products in each product lines their text descriptions. Display the product line, the number of products in each line, and the text description. Include only those product lines where the count of products is greater than 10.

```
SELECT
   productlines.productLine,
   COUNT(products.productCode) AS product_count,
   productlines.textDescription
FROM productlines
JOIN products ON productlines.productLine = products.productLine
GROUP BY productlines.productLine
HAVING product_count > 10;
```

11. Retrieve using JOIN the last name and first name of employees working in offices located in the USA.

```
SELECT e.firstname,e.lastname
FROM employees e
JOIN offices o ON e.officecode = e.officecode
HAVING o.country = 'usa';
```

12. Retrieve using Subquerry the last name and first name of employees working in offices located in the USA.

```
SELECT e.firstname,e.lastname
FROM employees e
WHERE officeCode IN (SELECT officeCode FROM offices WHERE country = 'USA');
```

13. Retrieve the customer numbers and payment amounts for customers whose payment amount is below the average payment amount, using a subquery.

```
SELECT customerNumber, amount
FROM payments
WHERE amount < (SELECT AVG(amount) FROM payments);
```

14.Retrieve the count, customer name, and customer number for customers who have not placed any orders. Include a grand total row that represents the overall count. (use subquery)

```
SELECT c.customerNumber, c.customerName, COUNT(orderNumber) AS orderCount
FROM customers c
LEFT JOIN orders ON c.customerNumber = orders.customerNumber
GROUP BY c.customerNumber, c.customerName
WITH ROLLUP;
```

15.Write a SQL query to retrieve customer numbers, names, total sales, and purchase categories from a retail database. The purchase category should be labeled as 'High Value' if the total sales for a customer exceed $100,000, and 'Regular Value' otherwise. Use the tables customers and payments, and include necessary aliases.

```
SELECT customerNumber, customerName, totalSales,
   CASE
      WHEN totalSales > 100000 THEN 'High Value'
      ELSE 'Regular Value'
   END AS purchaseCategory
FROM (SELECT c.customerNumber, c.customerName,
      COALESCE(SUM(p.amount), 0) AS totalSales
```

```
    FROM customers c
    LEFT JOIN payments p ON c.customerNumber = p.customerNumber
    GROUP BY c.customerNumber, c.customerName
) AS customer_sales;
```

16.List the employees and their respective managers employee name as "EmployeeName" and the manager name as "ManagerName".

```
SELECT CONCAT(e1.firstName, ' ', e1.lastName) AS EmployeeName,
      CONCAT(e2.firstName, ' ', e2.lastName) AS ManagerName
FROM employees e1
JOIN employees e2 ON e1.reportsTo = e2.employeeNumber;
```

17.List the employees and their respective managers who have the same job title. Display the employee name as "EmployeeName" and the manager name as "ManagerName".

```
SELECT CONCAT(e1.firstName, ' ', e1.lastName) AS EmployeeName,
      CONCAT(e2.firstName, ' ', e2.lastName) AS ManagerName
FROM employees e1
JOIN employees e2 ON e1.reportsTo = e2.employeeNumber
where e1.jobtitle=e2.jobtitle
```

18.List the employees and their respective managers employee name as "EmployeeName" and the manager name as "ManagerName". Show all the employees even if they don’t have a manager.

```
SELECT CONCAT(e1.firstName, ' ', e1.lastName) AS EmployeeName,
            COALESCE(e2.firstName, 'no manager') AS ManagerName
```

```
FROM employees e1
JOIN employees e2 ON e1.reportsTo = e2.employeeNumber
```

19.List the employees and their respective managers employee name as "EmployeeName" and the manager name as "ManagerName". Show all the employees even if they don’t have a manager.

```
SELECT CONCAT(e1.firstName, ' ', e1.lastName) AS EmployeeName,
        COALESCE(e2.firstName, 'no manager') AS ManagerName
FROM employees e1
JOIN employees e2 ON e1.reportsTo = e2.employeeNumber
```

20.Find the names of all customers who have placed at least one order. Use EXISTS

```
SELECT customerName
FROM customers c
WHERE EXISTS (
   SELECT 1
   FROM orders o
   WHERE o.customerNumber = c.customerNumber
 );
```

21.Retrieve the product names that have been ordered in the 2004 year. Use EXISTS

```
SELECT DISTINCT productName
FROM products p
WHERE EXISTS (
   SELECT 1
```

```
  FROM orderDetails od
    JOIN orders o ON od.orderNumber = o.orderNumber
  WHERE
    od.productCode = p.productCode
    AND YEAR(o.orderDate) = 2004
);
```

++

select productCode , orderNumber , quantityOrdered , rank() over (partition by productCode order BY quantityOrdered) as Rank

FROM orderdetails;

---

1. For each payment, calculate the cumulative sum of the payment amounts ordered by

customerNumber. Include all columns in the result. (hint, use only order by)

SELECT *,SUM(amount) OVER (PARTITION BY customerNumber ORDER BY paymentDate) AS cumulativeSum

FROM payments

ORDER BY customerNumber, paymentDate;

2. For each payment, calculate the running total of the payment amounts within each

customer's transactions, ordered by the payment amount.

SELECT *, SUM(amount) over(PARTITION BY customerNumber ORDER BY amount) as total

FROM `payments`;

3. For each payment, find the minimum and maximum payment amounts within each customer's transactions. Include all columns in the result, along with columns for the minimum and maximum amounts.

```
SELECT*,
  MIN(amount) OVER (PARTITION BY customerNumber) AS min,
  MAX(amount) OVER (PARTITION BY customerNumber) AS max
FROM payments;
```

4. For each order detail, find the minimum and maximum unit prices within each order. Include all columns in the result, along with columns for the minimum and maximum unit prices.

```
SELECT *,
  MIN(priceEach) OVER (PARTITION BY orderNumber) AS minPrice,
  MAX(priceEach) OVER (PARTITION BY orderNumber) AS maxPrice
FROM orderDetails;
```

5. Retrieve detailed information about each order detail, including the product name, product line, and vendor. Additionally, calculate the minimum and maximum unit prices for each order.

```
SELECT
  od.orderNumber,
  od.productCode,
  od.quantityOrdered,
  od.priceEach,
  p.productName,
  p.productLine,
```

```
  p.productVendor,
  MIN(od.priceEach) OVER (PARTITION BY od.orderNumber) AS minPrice,
  MAX(od.priceEach) OVER (PARTITION BY od.orderNumber) AS maxPrice
FROM orderDetails od
JOIN products p ON od.productCode = p.productCode
JOIN productLines pl ON p.productLine = pl.productLine;
```

6. Assign a unique row number to each order detail within each order, ordered by the quantity ordered. Display all columns along with the assigned row number.

```
SELECT *, ROW_NUMBER() OVER (PARTITION BY orderNumber ORDER BY quantityOrdered) AS rowNumber
FROM orderDetails;
```

7. Retrieve detailed information about each order detail, including product name, product line, and vendor. Additionally, assign a unique row number to each order detail within each order, ordered by the quantity ordered.

```
SELECT od.orderNumber, od.productCode, od.quantityOrdered, od.priceEach,
p.productName, p.productLine, p.productVendor,
ROW_NUMBER() OVER (PARTITION BY od.orderNumber ORDER BY od.quantityOrdered) AS rowNumber
FROM orderDetails od JOIN products p ON od.productCode = p.productCode;
```

8. Retrieve detailed information about each order detail, including product details, and assign a unique row number to each order detail within each order. Filter the results to include only orders with a specific product line 'Classic Cars'.

```
SELECT od.orderNumber, od.productCode, od.quantityOrdered, od.priceEach,
p.productName, p.productLine, p.productVendor,
ROW_NUMBER() OVER (PARTITION BY od.orderNumber ORDER BY od.quantityOrdered) AS rowNumber
```

```
FROM orderDetails od JOIN products p ON od.productCode = p.productCode
WHERE p.productLine = 'Classic Cars';
```

9. Retrieve detailed information about each order detail, including product details, and assign a unique row number to each order detail within each order. Filter the results to include only orders where the quantity ordered is greater than 10.

```
SELECT od.orderNumber, od.productCode, od.quantityOrdered, od.priceEach,
p.productName, p.productLine, p.productVendor,
ROW_NUMBER() OVER (PARTITION BY od.orderNumber ORDER BY od.quantityOrdered) AS rowNumber
FROM orderDetails od JOIN products p ON od.productCode = p.productCode
WHERE od.quantityOrdered > 10;
```

10. Retrieve detailed information about each order detail, including the assigned rank based on the quantity ordered.

```
select productCode , orderNumber , quantityOrdered , rank() over (order BY quantityOrdered DESC) as Rank
FROM orderdetails;
```

11. Retrieve detailed information about each order detail, including the assigned dense rank based on the quantity ordered.

```
select productCode , orderNumber , quantityOrdered , DENSE_RANK() over (order BY quantityOrdered DESC) as DENSE_RANK
FROM orderdetails;
```

12. Assign a unique row number to each order detail based on the quantity ordered.

Retrieve detailed information about each order detail, including the assigned row

number.

```
select productCode , orderNumber , quantityOrdered ,    ROW_NUMBER() OVER (ORDER BY quantityOrdered) AS row_number
FROM orderdetails;
```

13. Assign a unique row number to each order detail within each order based on the quantity

ordered. Retrieve detailed information about each order detail, including the assigned

row number within each order.

```
select productCode , orderNumber , quantityOrdered ,    ROW_NUMBER() OVER (PARTITION BY orderNumber ORDER BY quantityOrdered) AS row_number
FROM orderdetails;
```

14. Calculate the cumulative sum of the quantity ordered for each order detail, ordered by

the unit price. Retrieve detailed information about each order detail, including the

cumulative sum of quantity ordered.

```
select productCode , orderNumber , quantityOrdered , SUM(quantityOrdered) OVER (PARTITION BY orderNumber ORDER BY priceEach) AS cumulative_sum
FROM orderdetails;
```

15. For each order detail, calculate the cumulative sum of the quantity ordered within each

order, ordered by the order number. Retrieve information about the order number,

product code, quantity ordered, and the cumulative sum of quantity ordered. Display the

results in ascending order based on the order number.

```
select orderNumber, productCode , quantityOrdered , SUM(quantityOrdered) OVER (PARTITION BY orderNumber ORDER BY priceEach) AS cumulative_sum
FROM orderdetails
```

ORDER BY orderNumber ASC;

16. For each order detail, assign a rank based on the unit price of the product. Retrieve information about the order number, product code, unit price, and the assigned rank.

```
select orderNumber, productCode , priceEach , rank() over (PARTITION by orderNumber order BY priceEach) as Rank
FROM orderdetails
```

17. For each order detail, assign a dense rank based on the unit price of the product. Retrieve information about the order number, product code, unit price, and the assigned dense rank.

```
select orderNumber, productCode , priceEach , DENSE_RANK() over (order BY priceEach) as DENSE_RANK
FROM orderdetails;
```

18. For each order detail, calculate and display the following ranking metrics based on the unit price of the product: row number, rank, and dense rank. Retrieve information about the order number, product code, unit price, row number, rank, and dense rank.

```
select orderNumber, productCode , priceEach ,
  ROW_NUMBER() OVER (PARTITION BY orderNumber ORDER BY priceEach) AS row_number,
  RANK() OVER (PARTITION BY orderNumber ORDER BY priceEach) AS rank,
  DENSE_RANK() OVER (PARTITION BY orderNumber ORDER BY priceEach) AS dense_rank
FROM orderdetails;
```

19. Assign a dense rank to each order detail based on the descending unit price of the

product. Retrieve information about the order number, product code, unit price, and the assigned dense rank. Sort the results by the unit price in descending order.

```
select orderNumber, productCode , priceEach ,
  DENSE_RANK() OVER (PARTITION BY orderNumber ORDER BY priceEach) AS dense_rank
FROM orderdetails
ORDER BY priceEach DESC;
```

20. For each order detail with a unit price less than 100, assign a dense rank based on the descending unit price of the product. Retrieve information about the order number, product code, unit price, and the assigned dense rank. Sort the results by the unit price in descending order.

```
select orderNumber, productCode , priceEach ,
  DENSE_RANK() OVER (ORDER BY priceEach DESC) AS dense_rank
FROM orderdetails
WHERE priceEach < 100
ORDER BY priceEach DESC;
```

---

1. Retrieve the product name, buy price, and the maximum buy price across all products for each row. Sort the results in descending order based on the buy price.

```
SELECT productName, buyPrice, max(buyPrice) over(PARTITION by buyPrice) as MaxBuyPrice
FROM products
order by buyprice DESC;
```

2. For each product, retrieve the product name, buy price, and the minimum buy price across all products. Sort the results in ascending order based on the buy price.

```
SELECT productName, buyPrice, min(buyPrice) over(PARTITION by buyPrice) as minBuyPrice
FROM products
order by buyprice DESC;
```

3. Retrieve all payment details along with the maximum and minimum payment amounts across all payments. Sort the results in descending order based on the payment date.

```
SELECT *,MAX(amount) over() as max , MIN(amount) over() as min
FROM `payments`
order by paymentDate;
```

4. Retrieve payment details for the year 2004, including the payment amount, along with the maximum, minimum, and average payment amounts across all payments in that year. Sort the results in ascending order based on the payment date.

```
SELECT *,MAX(amount) over(), MIN(amount) over(), AVG(amount) over()
FROM `payments`
where year(paymentDate)=2004
order by paymentDate;
```

5. For each payment, retrieve all details and include a column indicating the cumulative sum of the payment amounts within the same year.

```
SELECT *, sum(amount) over(PARTITION by year(paymentDate) ORDER by paymentDate) as
sum_in_year
FROM `payments`;
```

6. For each payment, retrieve all details and include a column indicating the

cumulative sum of the payment amounts within the same year, ordered by the

month of the payment date.

```
SELECT *, sum(amount) over(PARTITION by month(paymentDate) ORDER by paymentDate) as sum_in_month
FROM `payments`;
```

7. For each payment, retrieve all details and include a column indicating the

cumulative sum of the payment amounts within the same year and month.

```
SELECT *, sum(amount) over(PARTITION by month(paymentDate) ORDER by paymentDate) as sum_in_month
FROM `payments`;
```

8. Retrieve details for each employee, including employee number, first name, and

office code. Additionally, include a column indicating the number of employees in

the same office for each employee. Sort the results in descending order based

on the number of employees in the office.

```
SELECT employeeNumber, firstName, officeCode,
count(employeeNumber) over(PARTITION by officeCode) as the_sum
FROM `employees`
order BY the_sum DESC;
```

9. Retrieve details for each employee, including employee number, first name,

office code, city of the office, the number of employees in the same office, and

the total number of employees across all offices. Sort the results in ascending order based on the number of employees in the office.

```
SELECT e.employeeNumber, e.firstName, e.officeCode,o.city,
count(e.employeeNumber) over(PARTITION by e.officeCode) as the_sum,
count(e.employeeNumber) over() as the_total
FROM `employees` e
JOIN offices o ON e.officeCode = o.officeCode
order BY the_sum ASC;
```

10.List all the details of orders along with the total quantity ordered for each order. Include orders where the total quantity ordered is greater than a specified value (e.g., 150). Display the order number, product code, quantity ordered, and the total quantity ordered for each order.

```
SELECT o.orderNumber, p.productcode, od.quantityOrdered, SUM(od.quantityOrdered) as total
FROM `orders` o
JOIN orderdetails od ON o.orderNumber = od.orderNumber
JOIN products p ON od.productcode = p.productcode
GROUP BY o.orderNumber
HAVING  SUM(od.quantityOrdered) > 150;
```

11. Retrieve the order numbers, the count of orders for each order, and the corresponding product codes. Include only orders where the count of products ordered is greater than 1.

```
SELECT o.orderNumber, COUNT(o.orderNumber) as total, GROUP_CONCAT(DISTINCT p.productCode) product_code
FROM `orders` o
JOIN orderdetails od ON o.orderNumber = od.orderNumber
JOIN products p ON od.productcode = p.productcode
GROUP BY o.orderNumber
HAVING  total>1;
```

12. List the details of each order line along with the rounded average of the product of quantityOrdered and priceEach for each order line number. Ensure that the rounded average is displayed with one decimal place.

```
SELECT *, ROUND(AVG(quantityOrdered * priceEach), 1) AS rounded_average
FROM `orderdetails`
GROUP BY orderNumber, orderLineNumber, quantityOrdered, priceEach;
```

13. Retrieve the details for each order line, including order number, order line number, quantity ordered, price each, the average value for the order line (quantityOrdered * priceEach) as (AVG_for_Line) , the total value per order (total_per_order), and the gap between the average and total values. Sort the results by order number.

```
SELECT ordernumber, orderLineNumber, quantityOrdered, priceEach,
(quantityOrdered * priceEach) AS AVG_for_Line,
SUM(quantityOrdered * priceEach) OVER (PARTITION BY orderNumber) AS total_per_order,
(SUM(quantityOrdered * priceEach) OVER (PARTITION BY orderNumber) - (quantityOrdered * priceEach)) AS gap
FROM `orderdetails`
```

```
ORDER BY orderNumber;
```

14. For each order line, retrieve the order number, order line number, quantity ordered, price each, the rounded average value for the order line (AVG_for_Line), the total value per order (total_per_order), and a comparison indication ('Higher', 'Lower', or 'Equal') based on the total value's relationship to the average. Sort the results by order number.

```
SELECT ordernumber, orderLineNumber, quantityOrdered, priceEach,
(quantityOrdered * priceEach) AS AVG_for_Line,
SUM(quantityOrdered * priceEach) OVER (PARTITION BY orderNumber) AS total_per_order,
CASE
   WHEN (quantityOrdered * priceEach) < SUM(quantityOrdered * priceEach) OVER (PARTITION BY orderNumber) THEN 'Lower'
   WHEN (quantityOrdered * priceEach) > SUM(quantityOrdered * priceEach) OVER (PARTITION BY orderNumber) THEN 'Higher'
   ELSE 'Equal'
END AS comparison_indication
FROM `orderdetails`
ORDER BY orderNumber;
```

15. Retrieve details for each order, including order number, customer name, order status, order line number, quantity ordered, price each, and additional columns: the average for each order line, the total per order line, and a comparison indicating if the total per order line is higher, lower, or equal to the average for that order line. Sort the results based on the order number.

```
############
SELECT o.orderNumber, c.customerName, o.status, od.orderLineNumber, od.quantityOrdered,
od.priceEach,
```

```
(od.quantityOrdered * od.priceEach) AS AVG_for_Line,

AVG(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber) AS avg_per_order,

SUM(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber, od.orderLineNumber)
AS total_per_order_line,

CASE

   WHEN SUM(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber,
od.orderLineNumber) <                    AVG(od.quantityOrdered * od.priceEach) OVER
(PARTITION BY o.orderNumber)

   THEN 'Lower'

   WHEN SUM(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber,
od.orderLineNumber) >                    AVG(od.quantityOrdered * od.priceEach) OVER (PARTITION BY
o.orderNumber)

   THEN 'Higher'

   ELSE 'Equal'

END AS comparison_indication

FROM orders o

JOIN customers c ON o.customerNumber = c.customerNumber

JOIN orderdetails od ON o.orderNumber = od.orderNumber

ORDER BY o.orderNumber;

###########


SELECT o.orderNumber, c.customerName, o.status, od.orderLineNumber, od.quantityOrdered,
od.priceEach,

(od.quantityOrdered * od.priceEach) AS AVG_for_Line,

AVG(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber) AS avg_per_order,

SUM(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber) AS
total_per_order_line,

CASE

   WHEN SUM(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber) <
AVG(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber)

   THEN 'Lower'
```

```
    WHEN SUM(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber) >
AVG(od.quantityOrdered * od.priceEach) OVER (PARTITION BY o.orderNumber)
    THEN 'Higher'
    ELSE 'Equal'
END AS comparison_indication
FROM orders o
JOIN customers c ON o.customerNumber = c.customerNumber
JOIN orderdetails od ON o.orderNumber = od.orderNumber
ORDER BY o.orderNumber;
```