TASK 4 SQL FOR DATA ANALYSIS

CREATE DATABASE SalesManagementDB;

USE SalesManagementDB;

Data Insertion & Table Creation Sets up the main `Sales` table and populates it with 50 rows of dummy transactional data for immediate use.
(CREATE/INSERT)**Basic Query: High-Value Sales**.

```
85
86  CREATE TABLE Product_Details (
87      ProductName VARCHAR(50) PRIMARY KEY,
88      Supplier VARCHAR(50),
89      IsHighMargin BOOLEAN
90  );
91
92  INSERT INTO Product_Details (ProductName, Supplier, IsHighMargin) VALUES
93  ('Laptop', 'TechCorp', 1),
94  ('Mobile', 'GigaComm', 1),
95  ('Sofa', 'HomeFurnish', 0),
96  ('Chair', 'HomeFurnish', 0),
97  ('Rice', 'AgriSupply', 1),
98  ('Shirt', 'FashionCo', 1),
99  ('Notebook', 'OfficePlus', 0),
100 ('Fridge', 'CoolingInc', 1),
101 ('New Product A', 'FutureTech', 1),
102 ('Old Stock Z', 'Warehouse', 0);
103
104
```

Output

Action Output

#	Time	Action	Message
4	14:34:29	CREATE TABLE Sales (SaleID INT, Product VARCHAR(50), Category VARCHAR(50), Qua...	0 row(s) affected
5	14:34:30	INSERT INTO Sales (SaleID, Product, Category, Quantity, Price, SaleDate) VALUES (1, 'La...	50 row(s) affected Records: 50 Duplicates: 0 Warnings: 0

Demonstrates the use of **`SELECT`**, **`WHERE`**, and **`ORDER BY`** clauses to find and sort high-revenue transactions in the 'Electronics' category

```
106 SELECT
107     S.SaleID,
108     S.Product,
109     S.Category,
110     PD.Supplier
111 FROM Sales AS S
112 INNER JOIN Product_Details AS PD ON S.Product = PD.ProductName;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleID	Product	Category	Supplier
1	Laptop	Electronics	TechCorp
2	Mobile	Electronics	GigaComm
6	Sofa	Furniture	HomeFurnish
7	Chair	Furniture	HomeFurnish
11	Rice	Grocery	AgriSupply
16	Shirt	Clothing	FashionCo
23	Notebook	Stationery	OfficePlus

WHERE AND ORDER BY

```sql
-- a. SELECT, WHERE, ORDER BY: Find high-value Electronics sales, sorted by newest date
SELECT
    SaleID,
    Product,
    Quantity,
    Price,
    (Quantity * Price) AS TotalRevenue,
    SaleDate
FROM Sales
WHERE Category = 'Electronics' AND (Quantity * Price) > 100000 -- Filter by Category and Revenue
ORDER BY SaleDate DESC; -- Sort by the most recent date
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleID	Product	Quantity	Price	TotalRevenue	SaleDate
29	Camera	3	45000.00	135000.00	2025-06-04
26	Fridge	2	55000.00	110000.00	2025-06-01
5	TV	3	40000.00	120000.00	2025-01-18
3	Tablet	7	30000.00	210000.00	2025-01-12
2	Mobile	10	25000.00	250000.00	2025-01-11
1	Laptop	5	60000.00	300000.00	2025-01-10

Limit to 1000 rows

```sql
-- c. Subquery: Find all sales where the Quantity sold is above the overall average quantity
SELECT
    SaleID,
    Product,
    Category,
    Quantity
FROM Sales
WHERE Quantity > (
    -- Inner Query: Calculates the average quantity of all sales
    SELECT AVG(Quantity)
    FROM Sales
)
ORDER BY Quantity DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleID	Product	Category	Quantity
15	Eggs	Grocery	50
22	Pen	Stationery	50
14	Milk	Grocery	40
25	Eraser	Stationery	40
36	Onions	Grocery	35

Uses **GROUP BY** and aggregate functions (**SUM**, **AVG**) to calculate total quantity sold, total revenue, and average unit price for each major product category.

GROUP BY

```
154 •  CREATE VIEW Monthly_Revenue_Summary AS
155    SELECT
156        DATE_FORMAT(SaleDate, '%Y-%m') AS SaleMonth, -- Changed from STRFTIME to DATE_FORMAT for MySQL compatibility
157        COUNT(SaleID) AS TotalSalesCount,
158        SUM(Quantity * Price) AS MonthlyRevenue
159    FROM Sales
160    GROUP BY 1
161    ORDER BY 1;
162
163 •  SELECT
164        SaleMonth,
165        MonthlyRevenue
166    FROM Monthly_Revenue_Summary
167    WHERE MonthlyRevenue > 500000;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleMonth	MonthlyRevenue
2025-01	910000.00

GROUP BY, ORDER BY

```
73
74
75     -- a, d. GROUP BY, SUM, AVG: Calculate Total Revenue and Average Unit Price per Category
76 •   SELECT
77         Category,
78         SUM(Quantity) AS TotalQuantitySold,
79         SUM(Quantity * Price) AS TotalRevenue, -- SUM is an aggregate function
80         AVG(Price) AS AverageUnitPrice         -- AVG is an aggregate function
81     FROM Sales
82     GROUP BY Category
83     ORDER BY TotalRevenue DESC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Category	TotalQuantitySold	TotalRevenue	AverageUnitPrice
Electronics	57	1357000.00	31900.000000
Furniture	44	557400.00	26680.000000
Clothing	96	109100.00	1780.000000
Grocery	285	18340.00	94.600000
Stationery	235	14550.00	70.500000

JOINS:

INNER JOIN:

```
106 •   SELECT
107         S.SaleID,
108         S.Product,
109         S.Category,
110         PD.Supplier
111     FROM Sales AS S
112     INNER JOIN Product_Details AS PD ON S.Product = PD.ProductName;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleID	Product	Category	Supplier
1	Laptop	Electronics	TechCorp
2	Mobile	Electronics	GigaComm
6	Sofa	Furniture	HomeFurnish
7	Chair	Furniture	HomeFurnish
11	Rice	Grocery	AgriSupply
16	Shirt	Clothing	FashionCo
23	Notebook	Stationery	OfficePlus

LEFT JOIN:

```
115     -- ** LEFT JOIN: Keep ALL records from the LEFT table (Sales) **
116     -- Shows every sale, and includes the supplier if a match is found (NULL otherwise).
117 •   SELECT
118         S.SaleID,
119         S.Product,
120         S.Category,
121         PD.Supplier
122     FROM Sales AS S
123     LEFT JOIN Product_Details AS PD ON S.Product = PD.ProductName;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleID	Product	Category	Supplier
1	Laptop	Electronics	TechCorp
2	Mobile	Electronics	GigaComm
3	Tablet	Electronics	NULL
4	Headphones	Electronics	NULL
5	TV	Electronics	NULL
6	Sofa	Furniture	HomeFurnish
7	Chair	Furniture	HomeFurnish

RIGHT JOIN:

Limit to 1000 rows

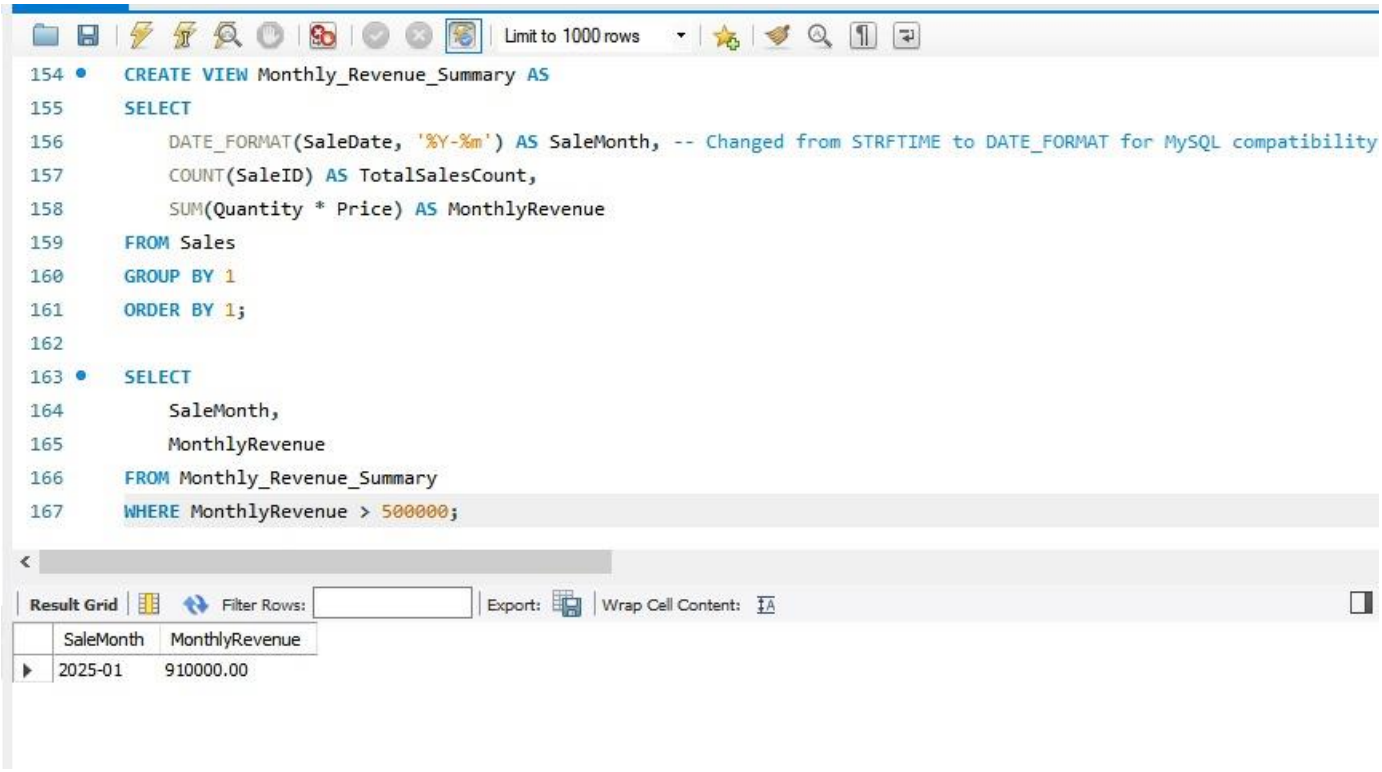
```
125     -- ** RIGHT JOIN: Keep ALL records from the RIGHT table (Product_Details) **
126     -- Shows all products in Product_Details, including "New Product A" and "Old Stock Z"
127     -- which haven't been sold yet (SaleID and Category columns will be NULL).
128 •   SELECT
129         PD.ProductName,
130         PD.Supplier,
131         S.SaleDate,
132         S.SaleID
133     FROM Sales AS S
134     RIGHT JOIN Product_Details AS PD ON S.Product = PD.ProductName
135     ORDER BY S.SaleID;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductName	Supplier	SaleDate	SaleID
New Product A	FutureTech	NULL	NULL
Old Stock Z	Warehouse	NULL	NULL
Laptop	TechCorp	2025-01-10	1
Mobile	GigaComm	2025-01-11	2
Sofa	HomeFurnish	2025-02-01	6
Chair	HomeFurnish	2025-02-02	7
Rice	AgriSupply	2025-03-01	11

Result 5

VIEW IN SQL

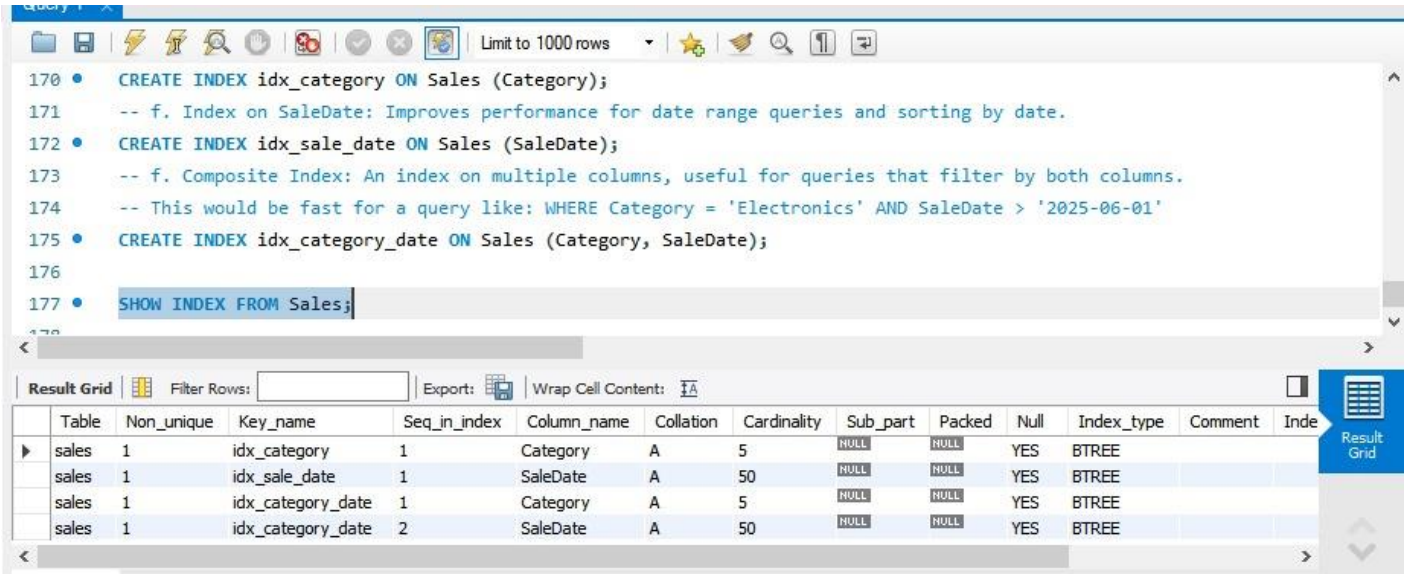


```
154 • CREATE VIEW Monthly_Revenue_Summary AS
155   SELECT
156       DATE_FORMAT(SaleDate, '%Y-%m') AS SaleMonth, -- Changed from STRFTIME to DATE_FORMAT for MySQL compatibility
157       COUNT(SaleID) AS TotalSalesCount,
158       SUM(Quantity * Price) AS MonthlyRevenue
159   FROM Sales
160   GROUP BY 1
161   ORDER BY 1;
162
163 • SELECT
164       SaleMonth,
165       MonthlyRevenue
166   FROM Monthly_Revenue_Summary
167   WHERE MonthlyRevenue > 500000;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleMonth	MonthlyRevenue
2025-01	910000.00

INDEX IN SQL:



```
170 • CREATE INDEX idx_category ON Sales (Category);
171   -- f. Index on SaleDate: Improves performance for date range queries and sorting by date.
172 • CREATE INDEX idx_sale_date ON Sales (SaleDate);
173   -- f. Composite Index: An index on multiple columns, useful for queries that filter by both columns.
174   -- This would be fast for a query like: WHERE Category = 'Electronics' AND SaleDate > '2025-06-01'
175 • CREATE INDEX idx_category_date ON Sales (Category, SaleDate);
176
177 • SHOW INDEX FROM Sales;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Inde
sales	1	idx_category	1	Category	A	5	NULL	NULL	YES	BTREE		
sales	1	idx_sale_date	1	SaleDate	A	50	NULL	NULL	YES	BTREE		
sales	1	idx_category_date	1	Category	A	5	NULL	NULL	YES	BTREE		
sales	1	idx_category_date	2	SaleDate	A	50	NULL	NULL	YES	BTREE		

- **`SELECT`**: The absolute core of SQL. It specifies **which columns** (or calculated values) you want to retrieve from the database.
- **`FROM`**: Indicates **which table(s)** the data should be retrieved from. Always follows `SELECT`.
- **`WHERE`**: Filters the rows. It specifies **which records** (rows) meet a certain condition (e.g., `Category = 'Electronics'`) and should be included in the final result.
- **`ORDER BY`**: Sorts the final result set. It specifies the column(s) by which the data should be **ordered** (ascending or descending).

Aggregation and Grouping

- **`GROUP BY`**: **Combines rows** with the same values into summary rows. This is essential when using aggregate functions.
- **Aggregate Functions** (`SUM`, `AVG`, `COUNT`): Perform a calculation on a set of values (often within a group) and return a single summary value.
  - **`SUM`**: Calculates the total value of a numeric column.
  - **`AVG`**: Calculates the average value of a numeric column.
  - **`COUNT`**: Counts the number of rows or non-null values.

Querying and Structure

- **`JOINS`**: Combines data from **two or more tables** based on a related column between them (like `Product`).
  - **`INNER JOIN`**: Returns only the rows that have **matching values** in both tables.
  - **`LEFT JOIN`**: Returns **all rows** from the left table, and the matched rows from the right table (shows `NULL` if no match).
  - **`RIGHT JOIN`**: Returns **all rows** from the right table, and the matched rows from the left table (shows `NULL` if no match).
- **Subqueries**: A query embedded (nested) inside another SQL query. They are used to perform operations where the result of one query is needed as **input for the main query** (e.g., finding sales above the calculated average).
- **`CREATE VIEW`**: Creates a **virtual table** based on the result set of an SQL query. It saves a complex query under a simple name so you can reuse it easily without typing out the full code every time.
- **`CREATE INDEX`**: A performance tool. It creates a special **lookup structure** on one or more columns that allows the database engine to find data much faster, especially for columns used in `WHERE`, `JOIN`, and `ORDER BY` clauses.