Assignment-6

Name- Ajay Chaudhary
Batch-Data Engineering (Batch-1)

Total Aggregations using SQL Queries-

The aggregate functions are:

function	**returns**
AVG()	the mean average of the elements in the column
COUNT()	the total number of elements in the column
DISTINCT()	the number of distinct values across the column
MAX()	the largest-value element in the column
MIN()	the smallest-value element in the column
SUM()	the arithmetic total of all values in the column

Table for performing aggregate functions-

```sql
CREATE TABLE Sales (
    SaleID INT PRIMARY KEY,
    ProductName VARCHAR(255),
    SaleDate DATE,
    Quantity INT,
    UnitPrice DECIMAL(10, 2)
);

-- Inserting sample data into the Sales table
INSERT INTO Sales (SaleID, ProductName, SaleDate, Quantity, UnitPrice)
VALUES
    (1, 'Product A', '2024-01-01', 10, 20.00),
    (2, 'Product B', '2024-01-02', 5, 15.50),
    (3, 'Product A', '2024-01-03', 8, 22.50),
    (4, 'Product C', '2024-01-04', 12, 18.75),
    (5, 'Product B', '2024-01-05', 15, 14.00);
```

```sql
SELECT MAX(UnitPrice) AS MaxUnitPrice
FROM Sales;
```

100% 1:24

Result Grid Filter Rows: Search Export:

MaxUnitPrice
22.50

```sql
SELECT MIN(UnitPrice) AS MinUnitPrice
FROM Sales;
```

100% 1:28

Result Grid Filter Rows: Search Export:

MinU...
14.00

```sql
SELECT AVG(Quantity) AS AvgQuantityPerSale
FROM Sales;
```

100% 12:31

Result Grid Filter Rows: Search Export:

AvgQuantityPerSale
10.0000

```sql
SELECT ProductName, SUM(Quantity * UnitPrice) AS TotalSales
FROM Sales
GROUP BY ProductName;
```

100% 1:37

Result Grid Filter Rows: Search Export:

ProductName	TotalSales
Product A	380.00
Product B	287.50
Product C	225.00

```sql
SELECT SaleDate, AVG(Quantity * UnitPrice) AS AvgSalesPerDay
FROM Sales
GROUP BY SaleDate;
```

100% 19:46

Result Grid | Filter Rows: Search | Export:

SaleDate	AvgSalesPerDay
2024-01-02	77.500000
2024-01-03	180.000000
2024-01-04	225.000000
2024-01-05	210.000000

```sql
SELECT ProductName, COUNT(*) AS NumberOfSales
FROM Sales
GROUP BY ProductName;
```

100% 22:41

Result Grid | Filter Rows: Search | Export:

ProductName	NumberOfSales
Product A	2
Product B	2
Product C	1

OVER and PARTITION BY Clause in SQL Queries

Calculate the sum of order amounts partitioned by city

```sql
SELECT
    orderid,
    Orderdate,
    CustomerName,
    Customercity,
    Orderamount,
    SUM(Orderamount) OVER (PARTITION BY Customercity ORDER BY Orderdate) AS TotalOrderAmountByCity
FROM
    Orders;
```

100% 12:79

Result Grid | Filter Rows: Search | Export:

orderid	Orderdate	CustomerName	Customercity	Orderamount	TotalOrderAmountBy...
1	2024-01-01	Customer1	CityA	100.00	100.00
3	2024-01-03	Customer3	CityA	200.25	300.25
6	2024-01-06	Customer6	CityA	220.50	520.75
9	2024-01-09	Customer9	CityA	110.75	631.50
2	2024-01-02	Customer2	CityB	150.50	150.50
5	2024-01-05	Customer5	CityB	180.00	330.50
8	2024-01-08	Customer8	CityB	90.25	420.75
4	2024-01-04	Customer4	CityC	120.75	120.75
7	2024-01-07	Customer7	CityC	130.00	250.75
10	2024-01-10	Customer10	CityC	160.00	410.75

Total Aggregation using OVER and PARTITION BY in SQL Queries

Suppose we want to find the following values in the Orders table

- Minimum order value in a city
- Maximum order value in a city
- Average order value in a city
- CustomerName and OrderAmount column as well

  We can use the **SQL PARTITION BY** clause with the **OVER** clause to specify the column on which we need to perform aggregation.

```sql
SELECT Customercity,
       CustomerName,
       OrderAmount,
       AVG(Orderamount) OVER(PARTITION BY Customercity) AS AvgOrderAmount,
       MIN(OrderAmount) OVER(PARTITION BY Customercity) AS MinOrderAmount,
       SUM(Orderamount) OVER(PARTITION BY Customercity) TotalOrderAmount
FROM Orders;
```

100% 1:89

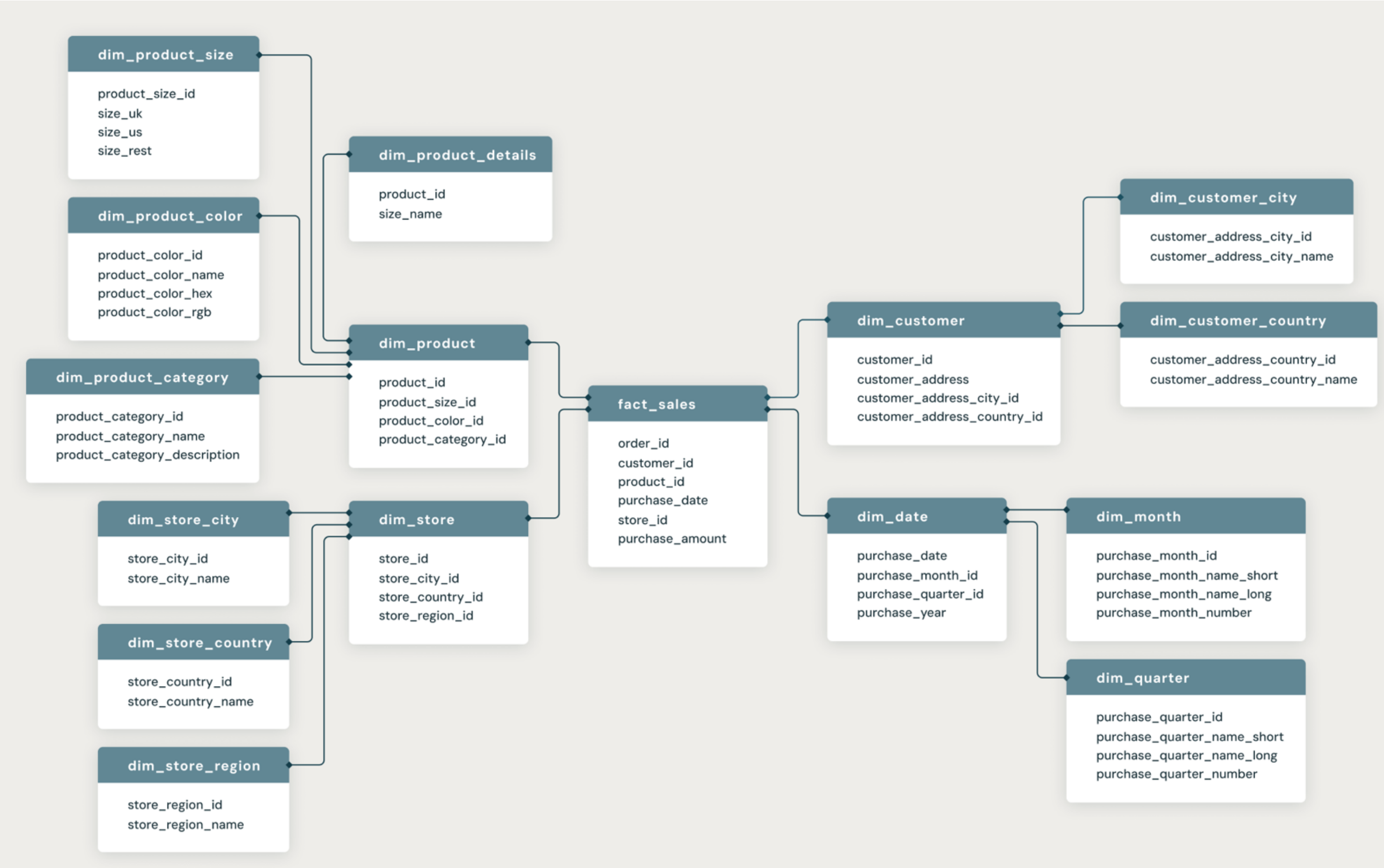
Result Grid Filter Rows: Search Export:

Customercity	CustomerName	OrderAmount	AvgOrderAmount	MinOrderAmou...	TotalOrderAmou...
CityA	Customer1	100.00	157.875000	100.00	631.50
CityA	Customer3	200.25	157.875000	100.00	631.50
CityA	Customer6	220.50	157.875000	100.00	631.50
CityA	Customer9	110.75	157.875000	100.00	631.50
CityB	Customer2	150.50	140.250000	90.25	420.75
CityB	Customer5	180.00	140.250000	90.25	420.75
CityB	Customer8	90.25	140.250000	90.25	420.75
CityC	Customer4	120.75	136.916667	120.75	410.75
CityC	Customer7	130.00	136.916667	120.75	410.75
CityC	Customer10	160.00	136.916667	120.75	410.75

Snowflaking schemas

This particular kind of data warehouse schema is shaped like a snowflake. The snowflake schema aims to normalize the star schema's denormalized data. When the star schema's dimensions are intricate, highly structured, and have numerous degrees of connection, and the kid tables have several parent tables, the snowflake structure emerges. Some of the star schema's common issues are resolved by the snowflake schema.

The snowflake schema can be thought of as a "multi-dimensional" structure. A snowflake schema's central component comprises Fact Tables that link the data inside the Dimension Tables, which then radiate outward like the Star Schema. The snowflake schema, on the other hand, divides the Dimension Tables into several tables, resulting in a snowflake pattern. Up until they are fully normalized, the Dimension Tables are split across multiple tables.



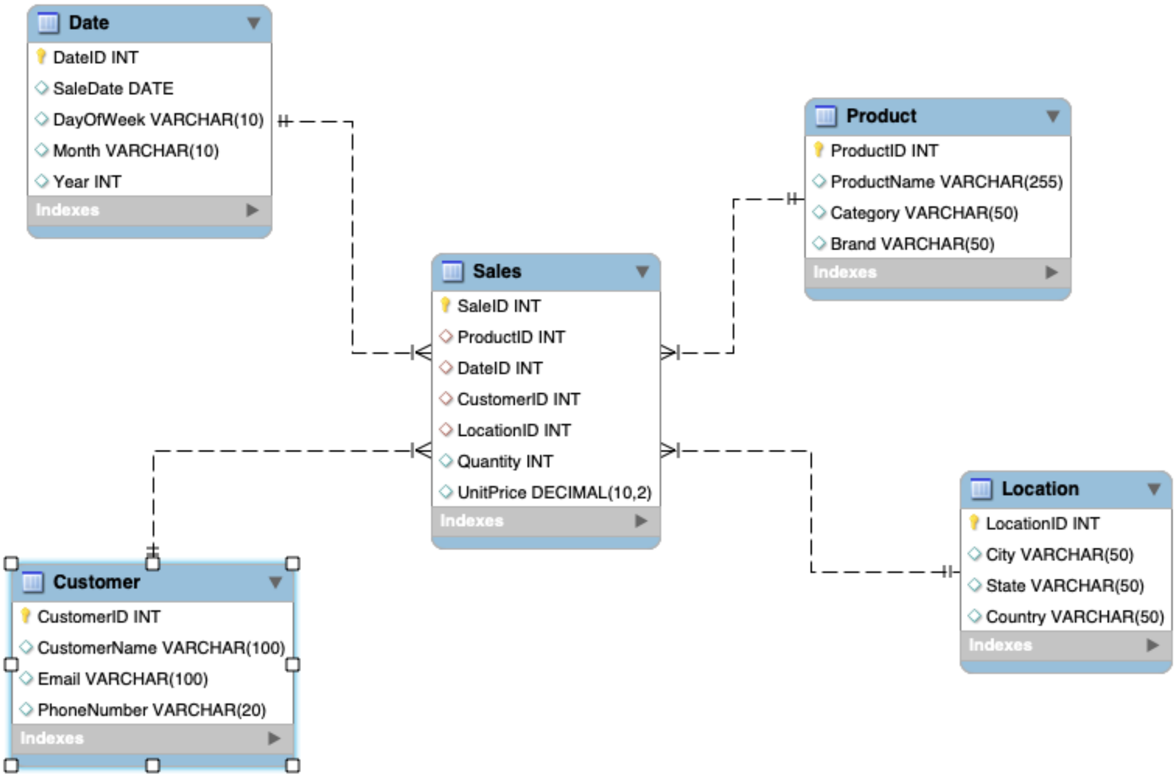
STAR SCHEMA

The star schema is the most straightforward method for arranging data in the data warehouse. Any or even more Fact Tables that index a number of Dimension Tables may be present in the star schema's central area. Dimensions Keys, Values, and Attributes are found in Dimension Tables, which are used to define Dimensions.

The star schema's objective is to distinguish between the descriptive or "DIMENSIONAL" data and the numerical "FACT" data that pertains to a business.

The information displayed in a numerical format, such as cost, speed, weight, and quantity, might be considered fact data. Along with numbers, dimensional data can also contain non-numerical elements like colors, places, names of salespeople and employees, etc.

While the Dimension Data is contained inside the Dimension Tables, the Fact Data is arranged within the Fact Tables. In a star schema, the Fact Tables are the integrating points at the core of a star.



Rules and Restrictions to Group and Filter Data in SQL queries,

GROUP BY enables you to use aggregate functions on groups of data returned from a query.

```
SELECT sales_agent,
       AVG(close_value)
  FROM sales_pipeline
 WHERE sales_pipeline.deal_stage = "Won"
 GROUP BY sales_agent
 ORDER BY AVG(close_value) DESC
```

sales_agent	avg
Elease Gluck	3614.9375
Darcel Schlecht	3304.3381088825213
Rosalina Dieter	3269.4861111111113
Daniell Hammack	3194.9912280701756
James Ascencio	3063.2074074074076
Rosie Papadopoulos	2950.8846153846152
Wilburn Farren	2866.181818181818
Reed Clapper	2827.974193548387
Donn Cantrell	2821.8987341772154

FILTER is a modifier used on an aggregate function to limit the values used in an aggregation. All the columns in the select statement that aren't aggregated should be specified in a GROUP BY clause in the query.

```
SELECT sales_agent,
       COUNT(sales_pipeline.close_value) AS total,
       COUNT(sales_pipeline.close_value)
FILTER(WHERE sales_pipeline.close_value > 1000) AS `over 1000`
  FROM sales_pipeline
 WHERE sales_pipeline.deal_stage = "Won"
 GROUP BY sales_pipeline.sales_agent
```

sales_agent	total	over 1000
Boris Faz	101	70
Maureen Marcano	149	96
Vicki Laflamme	221	111
Donn Cantrell	158	106
Jonathan Berthelot	171	74
Wilburn Farren	55	38
Elease Gluck	80	32
Cassey Cress	163	112
James Ascencio	135	88
Kami Bicknell	174	78

Order of Execution of SQL Queries-

Clause	Order	Description
FROM	1	The query begins with the `FROM` clause, where the database identifies the tables involved and accesses the necessary data.
WHERE	2	The database applies the conditions specified in the `WHERE` clause to filter the data retrieved from the tables in the `FROM` clause.
GROUP BY	3	If a `GROUP BY` clause is present, the data is grouped based on the specified columns, and aggregation functions (such as `SUM()`, `AVG()`, `COUNT()`) are applied to each group.
HAVING	4	The `HAVING` clause filters the aggregated data based on specified conditions.
SELECT	5	The `SELECT` clause defines the columns to be included in the final result set.
ORDER BY	6	If an `ORDER BY` clause is used, the result set is sorted according to the specified columns.

Clause	Order	Description
LIMIT/OFFSET	7	If `LIMIT` or `OFFSET` clause is present, the result set is restricted to the specified number of rows and optionally offset by a certain number of rows.

```sql
SELECT product_category, AVG(price) AS avg_price
FROM products
WHERE stock_quantity > 0
GROUP BY product_category
HAVING AVG(price) > 50
ORDER BY avg_price DESC
LIMIT 5;
```

Steps for the above query execution-

1. Retrieve data from the products table.
2. Apply the filter condition in the WHERE clause to the data.
3. Group the filtered data by the product_category column and calculate the average price for each group.
4. Filter the grouped data using the HAVING clause condition.
5. Select the product_category column and the calculated average price for the final result set.
6. Sort the result set based on the calculated average price in descending order.
7. Limit the result set to a maximum of 5 rows.

How to calculate Subtotals in SQL Queries-

- The **SELECT** statement specifies the columns to display (**Category**, **Amount**).
- The **SUM(Amount) OVER (PARTITION BY Category)** calculates a subtotal for each row based on the sum of the **Amount** within its **Category** partition.

In this example, the **Subtotal** column will show the sum of **Amount** for each category, providing a subtotal for each row relative to its category.

```sql
SELECT
    ItemName,
    Price,
    Quantity,
    Category,
    SUM(Price * Quantity) OVER (PARTITION BY Category) AS Subtotal
FROM
    Items;
```

Result Grid

ItemName	Price	Quantity	Category	Subtotal
Item A	10.00	2	Category1	65.00
Item B	15.00	3	Category1	65.00
Item C	8.50	5	Category2	54.50
Item D	12.00	1	Category2	54.50

Differences Between UNION EXCEPT and INTERSECT Operators in SQL Server

UNION - Customers and Orders:

- Combine unique records from both tables.

```sql
SELECT CustomerID, CustomerName, Email FROM Customers
UNION
SELECT CustomerID, 'No Name' AS CustomerName, 'No Email' AS Email FROM Orders1;
```

100% 1:155

Result Grid Filter Rows: Search Export:

CustomerID	CustomerName	Email
1	John Doe	john.doe@example.com
2	Jane Smith	jane.smith@example.com
3	Bob Johnson	bob.johnson@example.com
4	Alice Brown	alice.brown@example.com
5	Charlie Lee	charlie.lee@example.com
1	No Name	No Email
2	No Name	No Email

EXCEPT - Customers not placing Orders:

- Retrieve customers who have not placed any orders.

```sql
-- except
SELECT CustomerID, CustomerName, Email FROM Customers
EXCEPT
SELECT c.CustomerID, c.CustomerName, c.Email FROM Customers c
JOIN Orders1 o ON c.CustomerID = o.CustomerID;
```

100% 47:160 1 error found

Result Grid Filter Rows: Search Export:

CustomerID	CustomerName	Email
5	Charlie Lee	charlie.lee@example.com

INTERSECT - Customers placing Orders:

- Retrieve customers who have placed orders.

```sql
SELECT c.CustomerID, c.CustomerName, c.Email
FROM Customers c
INNER JOIN Orders1 o ON c.CustomerID = o.CustomerID;
```

100% 53:172 2 errors found

Result Grid Filter Rows: Search Export:

CustomerID	CustomerName	Email
1	John Doe	john.doe@example.com
2	Jane Smith	jane.smith@example.com
1	John Doe	john.doe@example.com
3	Bob Johnson	bob.johnson@example.com
4	Alice Brown	alice.brown@example.com