correlation.·one

TECH FOR JOBS

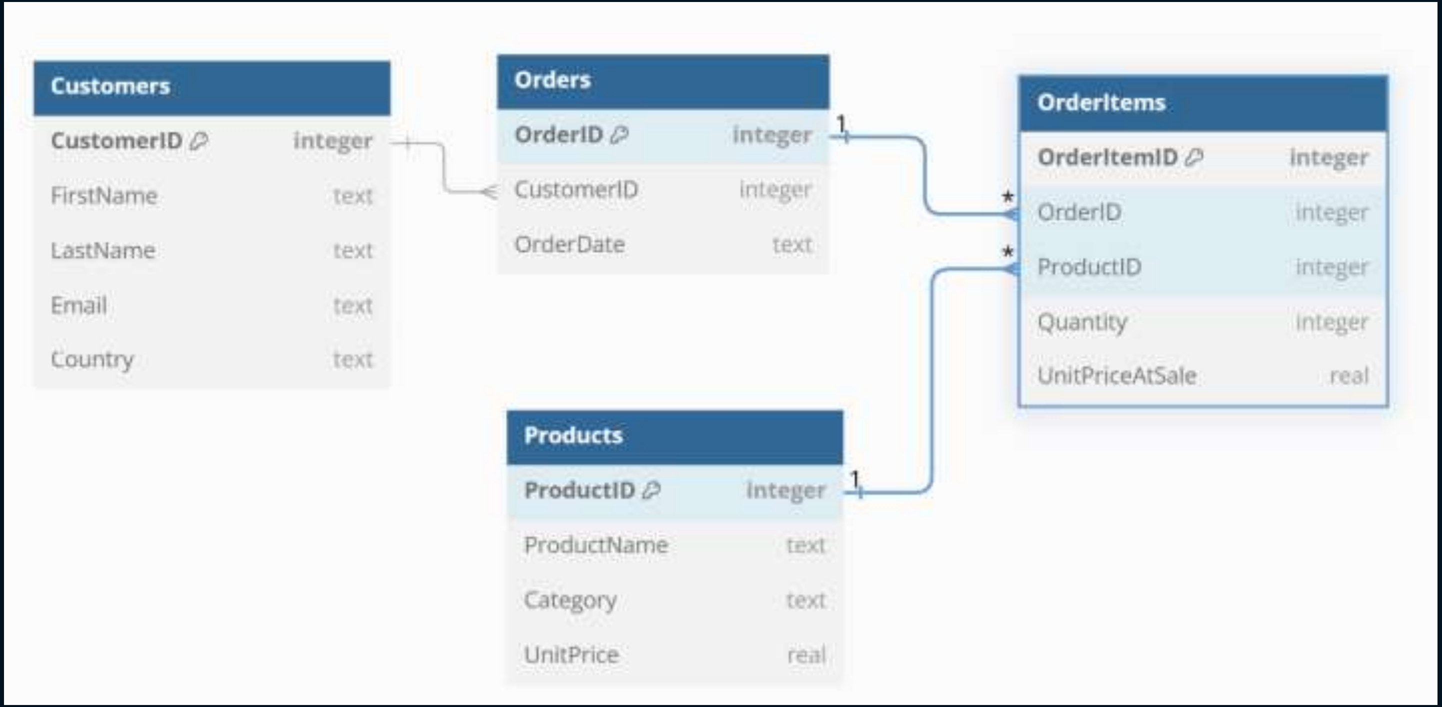
Support Session 7

Ahmad Albaqsami

Agenda

- SQL Basics II
- Exercise
- Handling Null Values in SQL

Our Small Case Study



Customers

CustomerID	integer
FirstName	text
LastName	text
Email	text
Country	text

Orders

OrderID	integer
CustomerID	integer
OrderDate	text

OrderItems

OrderItemID	integer
OrderID	integer
ProductID	integer
Quantity	integer
UnitPriceAtSale	real

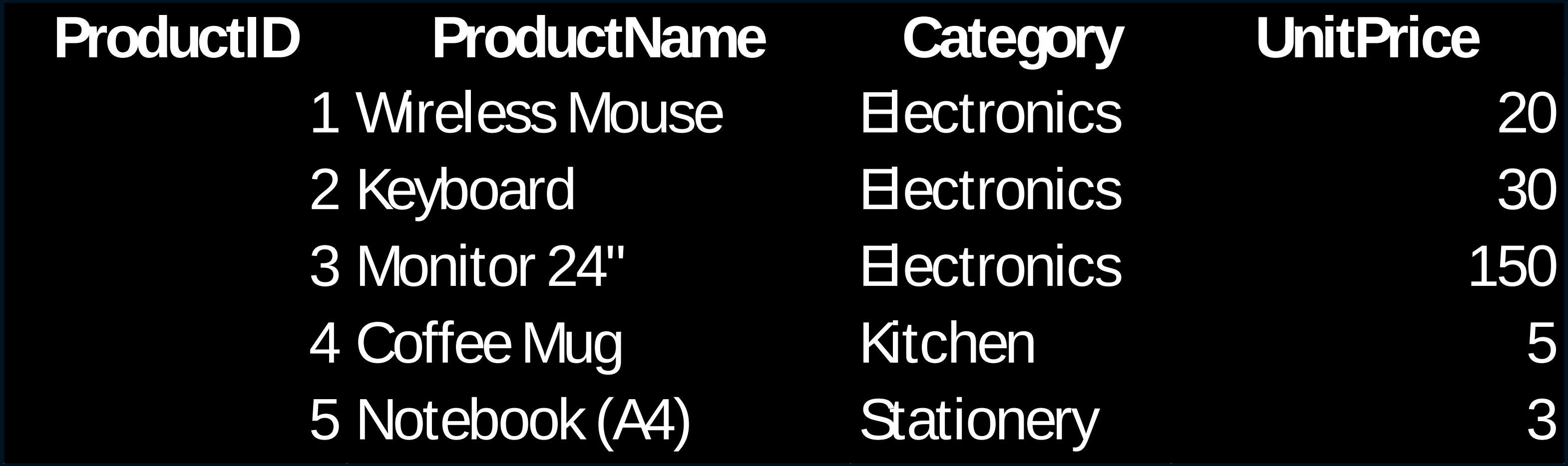
Products

ProductID	integer
ProductName	text
Category	text
UnitPrice	real

Customers table

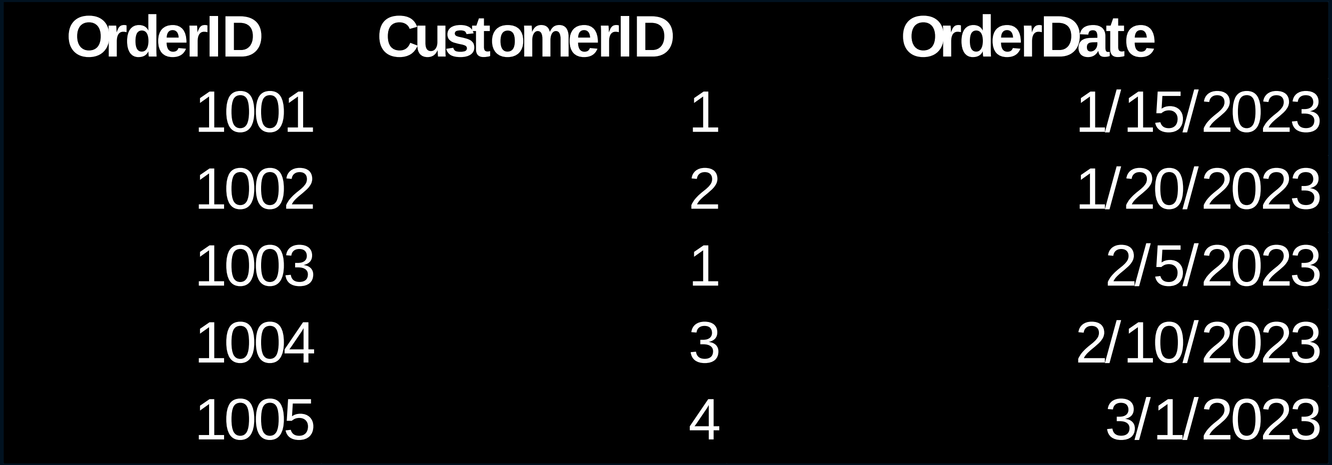
CustomerID	FirstName	LastName	Email	Country
1	John	Doe	john.doe@example.com	USA
2	Jane	Smith	jane.smith@example.com	UK
3	Carlos	Ruiz	carlos.ruiz@example.es	Spain
4	Maria	Garcia	maria.garcia@example.mx	Mexico
5	Li	Wei	li.wei@example.cn	China

Products table



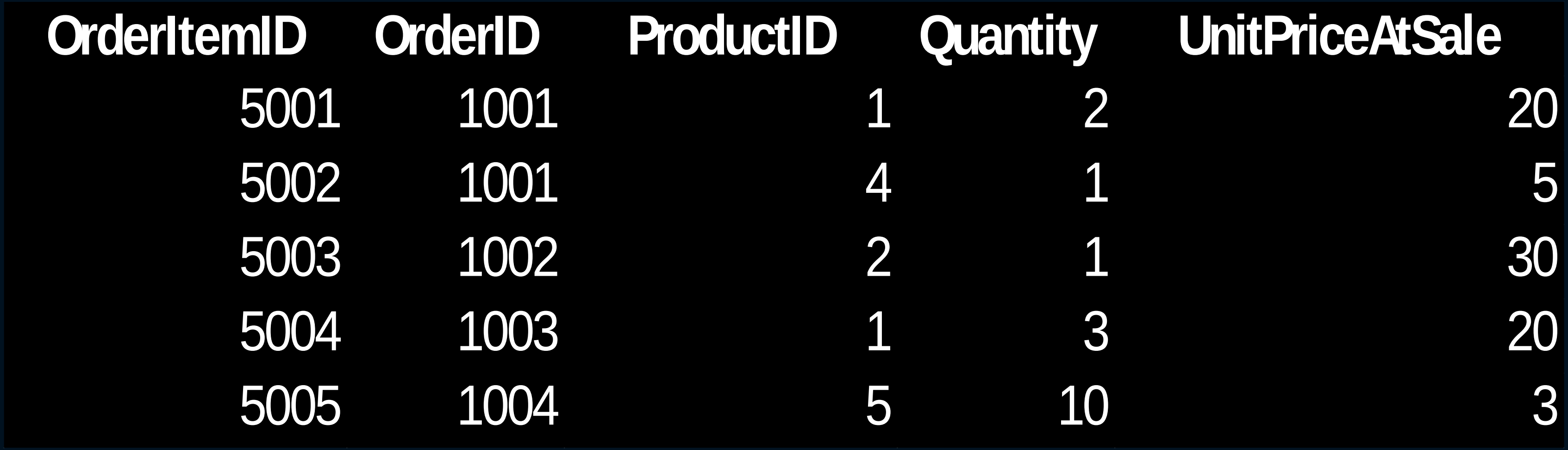
ProductID	ProductName	Category	UnitPrice
1	Wireless Mouse	Electronics	20
2	Keyboard	Electronics	30
3	Monitor 24"	Electronics	150
4	Coffee Mug	Kitchen	5
5	Notebook (A4)	Stationery	3

Orders table



OrderID	CustomerID	OrderDate
1001	1	1/15/2023
1002	2	1/20/2023
1003	1	2/5/2023
1004	3	2/10/2023
1005	4	3/1/2023

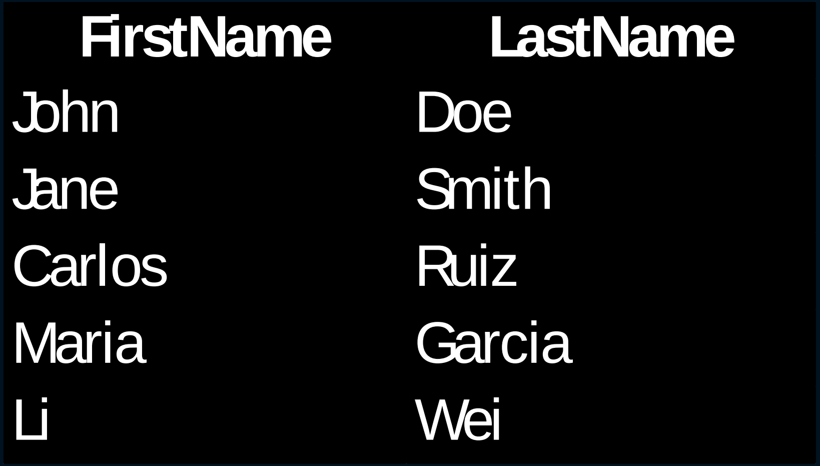
OrderItems table



OrderItemID	OrderID	ProductID	Quantity	UnitPriceAtSale
5001	1001	1	2	20
5002	1001	4	1	5
5003	1002	2	1	30
5004	1003	1	3	20
5005	1004	5	10	3

Recap: Basic Select

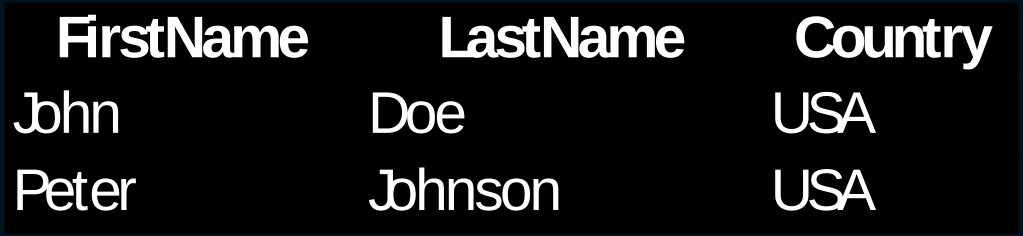
-
  ```
  SELECT FirstName, LastName
   FROM Customers
   LIMIT 5;
  ```



FirstName	LastName
John	Doe
Jane	Smith
Carlos	Ruiz
Maria	Garcia
Li	Wei

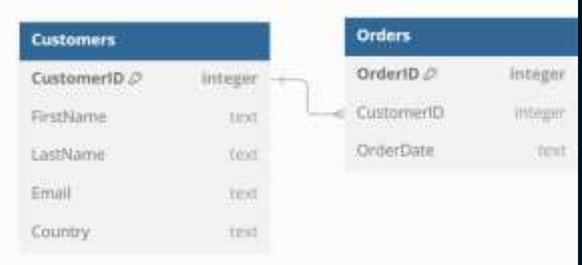
Recap: WHERE

```
SELECT FirstName, LastName, Country
 FROM Customers
 WHERE Country = 'USA';
```



FirstName	LastName	Country
John	Doe	USA
Peter	Johnson	USA

Recap: JOIN



Customers	
CustomerID	integer
FirstName	text
LastName	text
Email	text
Country	text

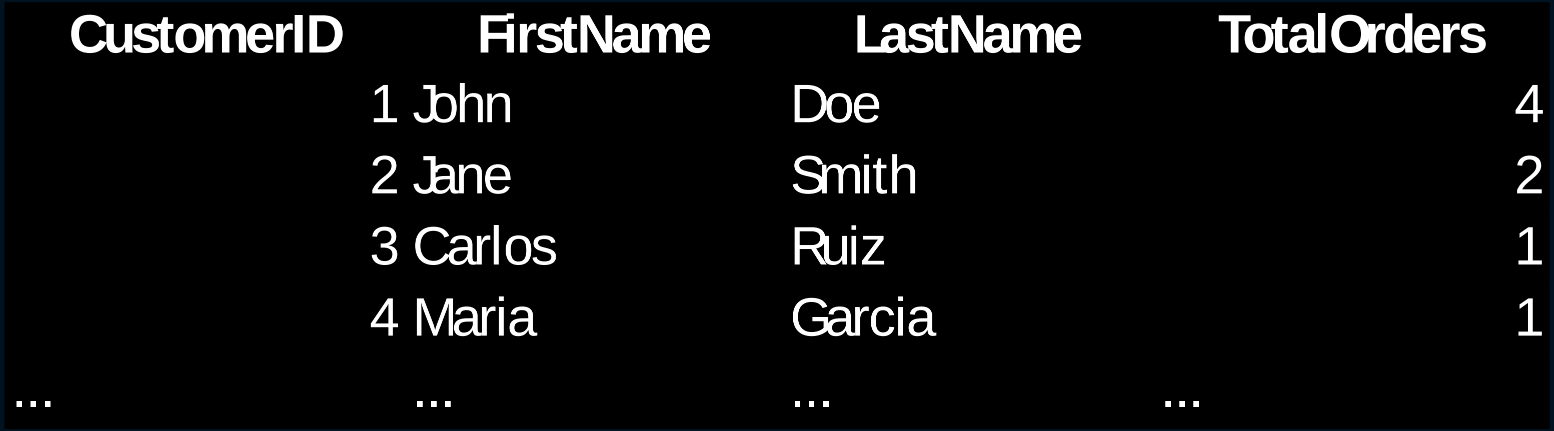
Orders	
OrderID	integer
CustomerID	integer
OrderDate	text

-
  ```
  SELECT C.FirstName, C.LastName, O.OrderDate
   FROM Customers AS C
  JOIN Orders AS O ON C.CustomerID = O.CustomerID
  WHERE C.CustomerID = 1;
  ```

FirstName	LastName	OrderDate
John	Doe	1/15/2023
John	Doe	2/5/2023
John	Doe	3/5/2023
John	Doe	4/15/2023

Aggregation (COUNT)

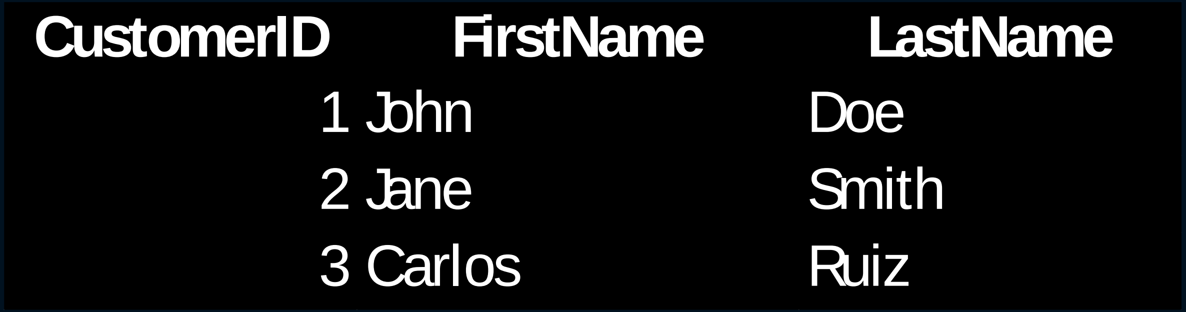
-
  ```
  SELECT C.CustomerID, C.FirstName, C.LastName, COUNT(O.OrderID) AS
  TotalOrders
  FROM Customers AS C
  JOIN Orders AS O ON C.CustomerID = O.CustomerID
  GROUP BY C.CustomerID;
  ```



CustomerID	FirstName	LastName	TotalOrders
1	John	Doe	4
2	Jane	Smith	2
3	Carlos	Ruiz	1
4	Maria	Garcia	1
...	...	...	...

Aggregation…. Let’s break it breakdown (Step 1)

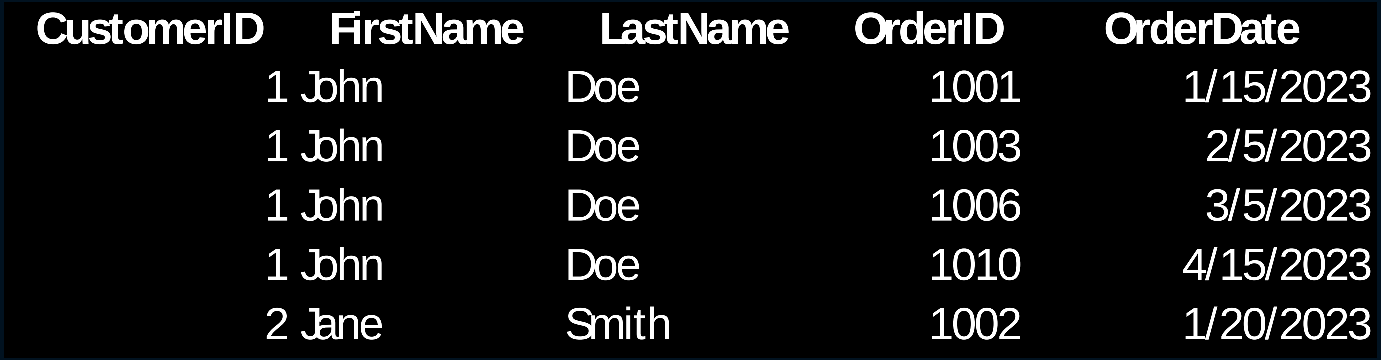
- Begin by selecting basic customer information.
- SELECT CustomerID, FirstName, LastName
  FROM Customers
  LIMIT 3;



CustomerID	FirstName	LastName
1	John	Doe
2	Jane	Smith
3	Carlos	Ruiz

Aggregation…. Let’s break it breakdown (Step 2)

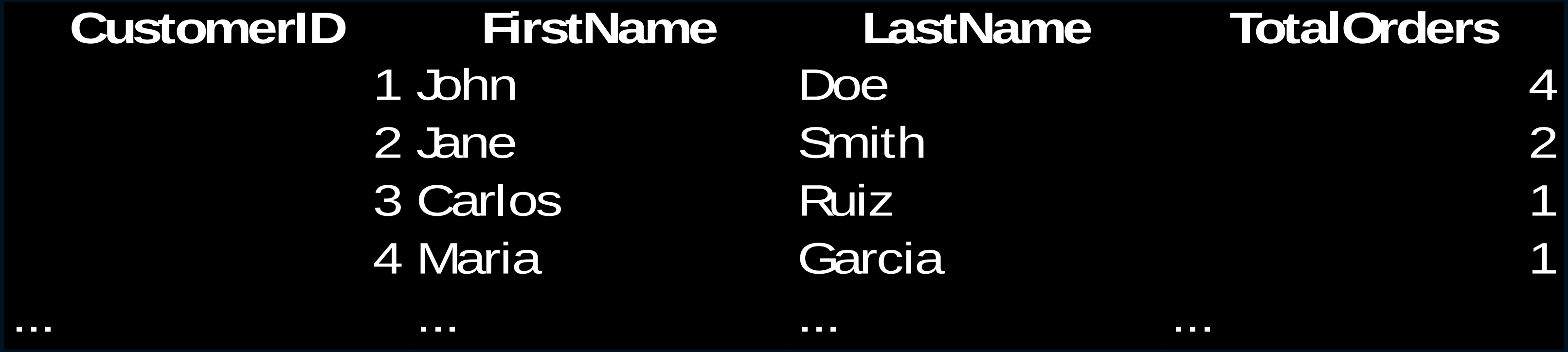
- Now include the Orders table to see each customer’s orders. Note that this will show multiple rows per customer if they have multiple orders.
-
  ```
  SELECT C.CustomerID, C.FirstName, C.LastName, O.OrderID, O.OrderDate
   FROM Customers AS C
   JOIN Orders AS O ON C.CustomerID = O.CustomerID
   ORDER BY C.CustomerID
   LIMIT 5;
  ```



CustomerID	FirstName	LastName	OrderID	OrderDate
1	John	Doe	1001	1/15/2023
1	John	Doe	1003	2/5/2023
1	John	Doe	1006	3/5/2023
1	John	Doe	1010	4/15/2023
2	Jane	Smith	1002	1/20/2023

Aggregation…. Let’s break it breakdown (Step 3)

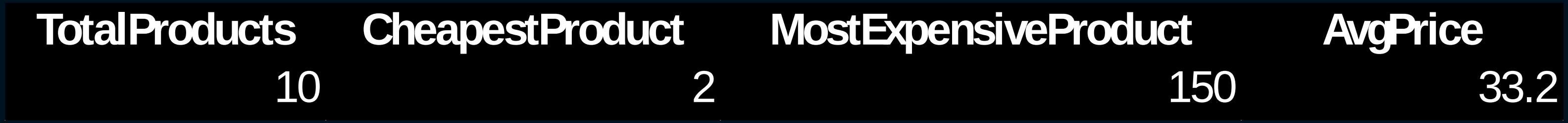
- Use GROUP BY to count the total number of orders for each customer.
-
  ```
  SELECT C.CustomerID, C.FirstName, C.LastName, COUNT(O.OrderID) AS
  TotalOrders
  FROM Customers AS C
  JOIN Orders AS O ON C.CustomerID = O.CustomerID
  GROUP BY C.CustomerID;
  ```



CustomerID	FirstName	LastName	TotalOrders
1	John	Doe	4
2	Jane	Smith	2
3	Carlos	Ruiz	1
4	Maria	Garcia	1
…	…	…	…

Aggregates (COUNT, SUM, AVG, MIN, MAX)

-
  ```
  SELECT
      COUNT(*) AS TotalProducts,
      MIN(UnitPrice) AS CheapestProduct,
      MAX(UnitPrice) AS MostExpensiveProduct,
      AVG(UnitPrice) AS AvgPrice
   FROM Products;
  ```



TotalProducts	CheapestProduct	MostExpensiveProduct	AvgPrice
10	2	150	33.2

Common Table Expression (CTE)

- A temporary, named result set in SQL that can be referenced within the same query. It is used to improve query readability and modularity, especially for complex queries.
- **Temporary Scope**: The CTE exists only during the execution of the query in which it is defined.
- **Improves Readability**: It simplifies complex SQL by breaking it into modular, reusable components.
- **Recursive Capability**: CTEs can be recursive, allowing them to process hierarchical or iterative data.

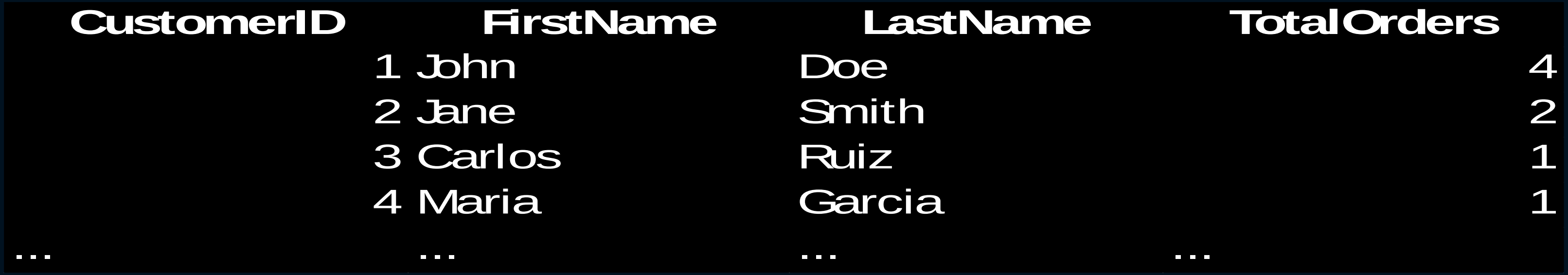
WITH

- Sytanx

```
WITH CTE_Name AS (
    SELECT column1, column2, ...
    FROM some_table
    WHERE conditions
    )
   SELECT *
   FROM CTE_Name;
```

CTE Example

```
● WITH CustomerOrderCounts AS (
    SELECT C.CustomerID, C.FirstName, C.LastName, COUNT(O.OrderID) AS TotalOrders
     FROM Customers AS C
    JOIN Orders AS O ON C.CustomerID = O.CustomerID
    GROUP BY C.CustomerID;
  )
SELECT *
FROM CustomerOrderCounts;
```



CustomerID	FirstName	LastName	TotalOrders
1	John	Doe	4
2	Jane	Smith	2
3	Carlos	Ruiz	1
4	Maria	Garcia	1
...	...	...	...

Views

- A View is a **virtual** table based on a SELECT query
- **Virtual Table**: A view does not store data permanently; it derives its data from the underlying tables in the query.
- **Dynamic**: The data in a view is always up-to-date because it reflects the current state of the underlying tables.

VIEW

- Syntax

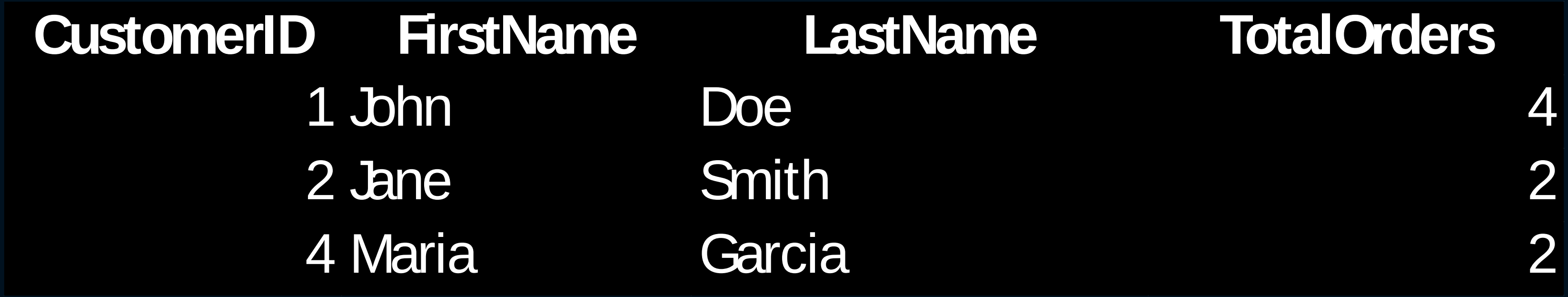
```
CREATE VIEW ViewName AS
SELECT column1, column2, ...
FROM TableName
...;
```

VIEW example

-
  ```
  CREATE VIEW v_CustomerOrderCounts AS
  SELECT C.CustomerID, C.FirstName, C.LastName, COUNT(O.OrderID) AS
  TotalOrders
  FROM Customers AS C
  JOIN Orders AS O ON C.CustomerID = O.CustomerID
  GROUP BY C.CustomerID;
  ```

VIEW example

-
  ```
  SELECT *
   FROM v_CustomerOrderCounts
   WHERE TotalOrders > 1;
  ```



CustomerID	FirstName	LastName	TotalOrders
1	John	Doe	4
2	Jane	Smith	2
4	Maria	Garcia	2

CASE

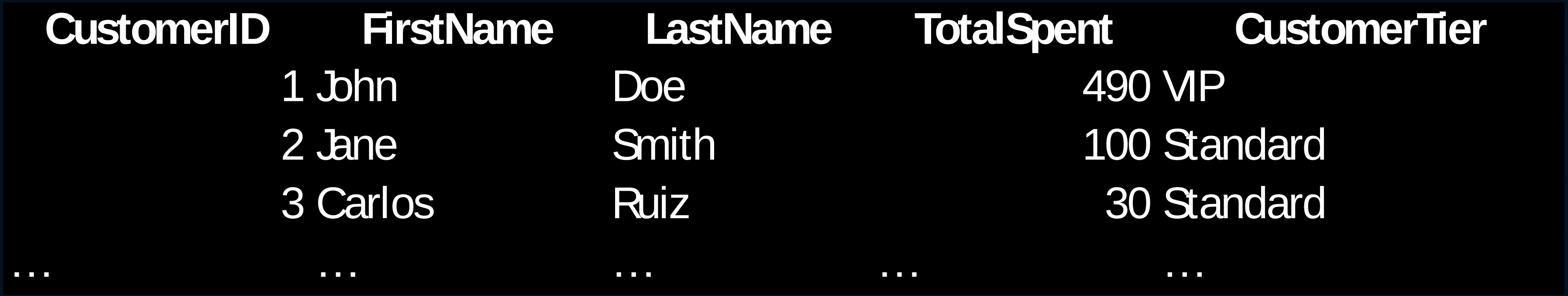
- conditional expression that provides **if-then-else** logic to queries
- CASE statement is commonly used in the SELECT, WHERE, ORDER BY, and GROUP BY clauses.
- SYNTAX

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE resultN
END AS alias_column_name
```

CASE example

```
●  SELECT C.CustomerID, C.FirstName, C.LastName,
   SUM(OI.Quantity * OI.UnitPriceAtSale) AS TotalSpent,
   CASE
       WHEN SUM(OI.Quantity * OI.UnitPriceAtSale) > 400 THEN "VIP"
WHEN SUM(OI.Quantity * OI.UnitPriceAtSale) >= 200 THEN “Preferred”
       ELSE “Standard”
       END AS CustomerTier
    FROM Customers AS C
   JOIN Orders AS O ON C.CustomerID = O.CustomerID
   JOIN OrderItems AS OI ON O.OrderID = OI.OrderID
   GROUP BY C.CustomerID;
```

CASE example



CustomerID	FirstName	LastName	TotalSpent	CustomerTier
1	John	Doe	490	VIP
2	Jane	Smith	100	Standard
3	Carlos	Ruiz	30	Standard
…	…	…	…	…

HAVING

- The HAVING clause filters data after it has been grouped by the GROUP BY clause. It applies conditions to aggregated results.
- Used for filtering grouped data.
- Can use aggregate functions (like SUM, COUNT, etc.).
- WHERE is used to filter raw data before grouping, and HAVING is used to filter the grouped results

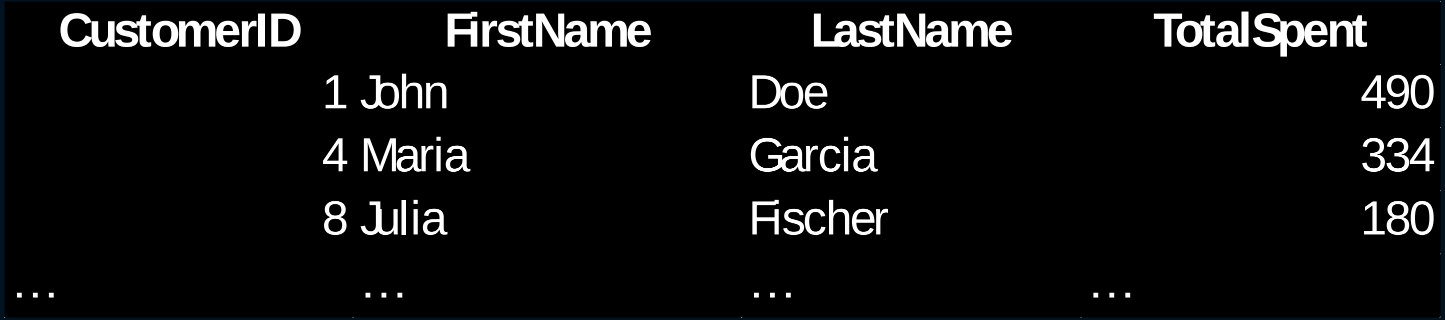
HAVING

- Syntax

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
GROUP BY column1, column2
HAVING condition;
```

HAVING example

```
SELECT C.CustomerID, C.FirstName, C.LastName,
 SUM(OI.Quantity * OI.UnitPriceAtSale) AS TotalSpent
FROM Customers AS C
JOIN Orders AS O ON C.CustomerID = O.CustomerID
JOIN OrderItems AS OI ON O.OrderID = OI.OrderID
GROUP BY C.CustomerID
HAVING SUM(OI.Quantity * OI.UnitPriceAtSale) > 100;
```



CustomerID	FirstName	LastName	TotalSpent
1	John	Doe	490
4	Maria	Garcia	334
8	Julia	Fischer	180
…	…	…	…

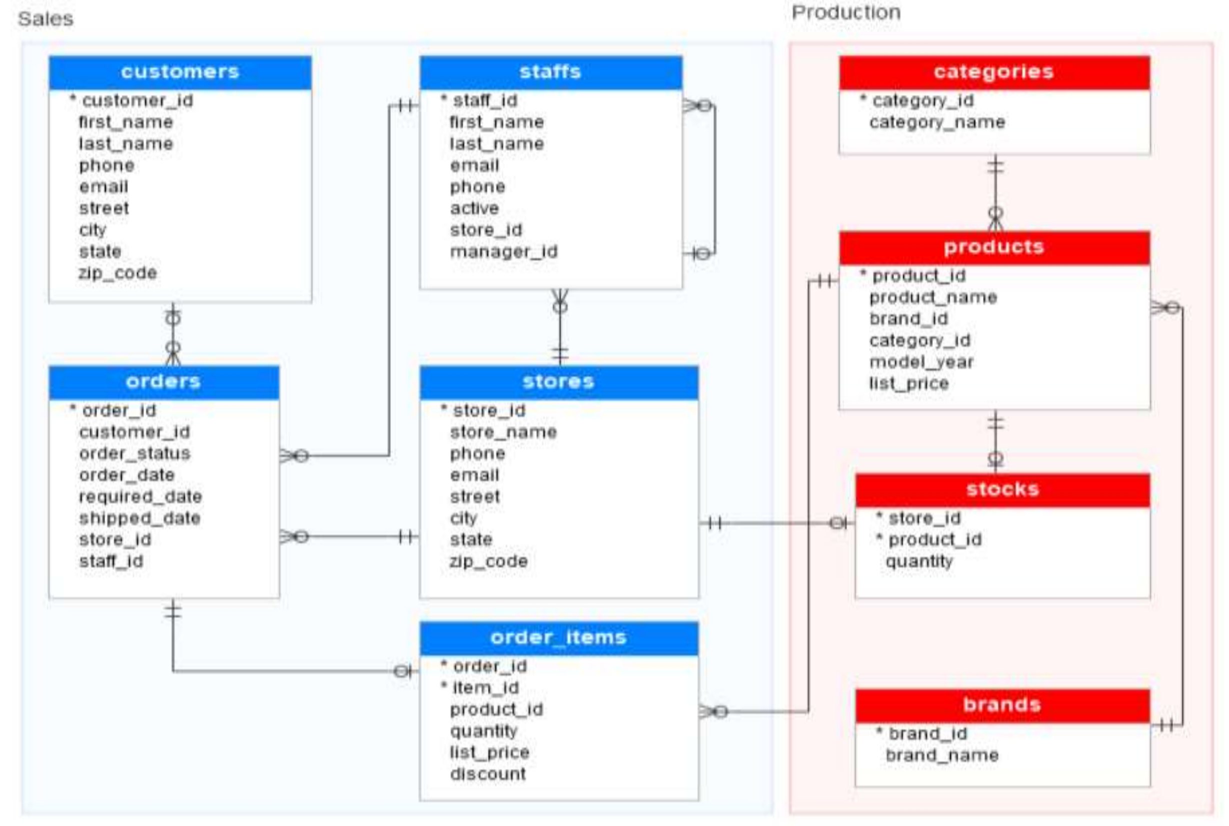
Putting It All Together (CTE + CASE + HAVING)

-
  ```
   WITH CustomerTotals AS (
   SELECT C.CustomerID, C.FirstName, C.LastName,
         SUM(OI.Quantity * OI.UnitPriceAtSale) AS TotalSpent
    FROM Customers AS C
   JOIN Orders AS O ON C.CustomerID = O.CustomerID
   JOIN OrderItems OI ON O.OrderID = OI.OrderID
   GROUP BY C.CustomerID
   HAVING SUM(OI.Quantity * OI.UnitPriceAtSale) > 100
  )
  SELECT CustomerID, FirstName, LastName, TotalSpent,
  CASE
      WHEN TotalSpent > 400 THEN "VIP"
      WHEN TotalSpent >= 200 THEN "Preferred"
      ELSE "Standard"
   END AS CustomerTier
   FROM CustomerTotals;
  ```

Order of execution

- **FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT**

Case Study



Sales

customers
* customer_id
first_name
last_name
phone
email
street
city
state
zip_code

staffs
* staff_id
first_name
last_name
email
phone
active
store_id
manager_id

orders
* order_id
customer_id
order_status
order_date
required_date
shipped_date
store_id
staff_id

stores
* store_id
store_name
phone
email
street
city
state
zip_code

order_items
* order_id
* item_id
product_id
quantity
list_price
discount

Production

categories
* category_id
category_name

products
* product_id
product_name
brand_id
category_id
model_year
list_price

stocks
* store_id
* product_id
quantity

brands
* brand_id
brand_name

Handling NULL Values in SQL

- Definition: Represents the absence of a value or unknown data.
- Not Equal to:
    - Zero
    - Empty String
    - Any Specific Value

Why Use NULL?

- Indicates missing or unknown data.
- Differentiates between:
    - Unknown values (NULL)
    - Explicit values (e.g., 0, empty)

IS NULL

- Syntax

```
SELECT column_name
FROM table_name
WHERE column_name IS NULL;

SELECT column_name
FROM table_name
WHERE column_name IS NOT NULL;
```

Aggregate Functions with NULLs

- SUM, AVG: Ignore NULL values.
- COUNT: Counts rows, NULL or not.
- MIN, MAX: Ignore NULLs when finding extremes.

Calculating NULL Percentages

- Count Rows

```
SELECT COUNT(*) AS total_rows,
       SUM(CASE
               WHEN column1 IS NULL THEN 1
               ELSE 0
               END) AS null_count
FROM your_table;
```

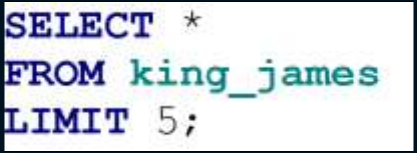
- Percentage

```
SELECT null_count / total_rows * 100 AS null_percentage
FROM (...above subquery...);
```

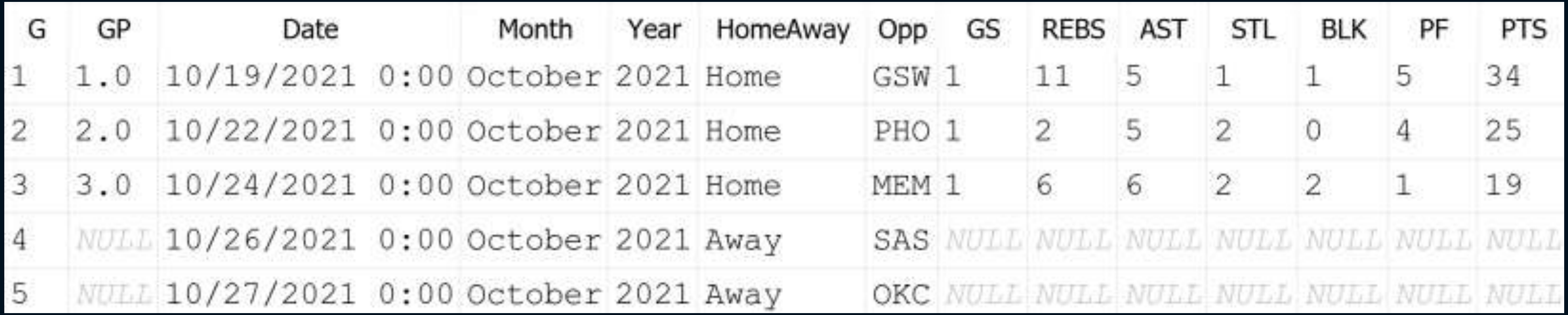
Key Takeaways

- NULL is critical for data representation.
- Effective handling ensures accurate insights.
- Aggregate functions are NULL-aware by default.

Case study

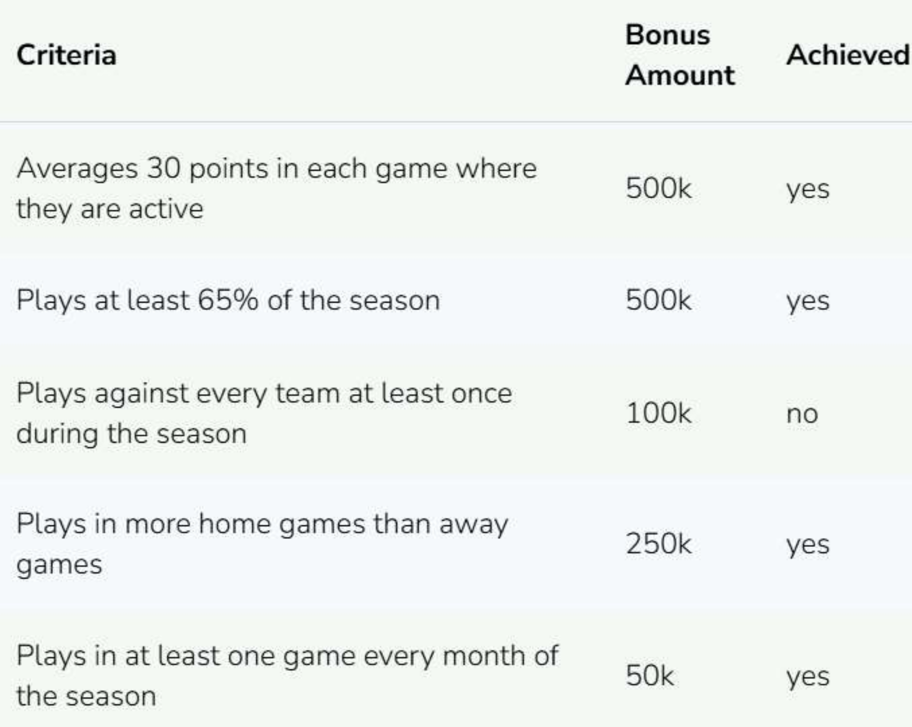


```
SELECT *
FROM king_james
LIMIT 5;
```



G	GP	Date	Month	Year	HomeAway	Opp	GS	REBS	AST	STL	BLK	PF	PTS
1	1.0	10/19/2021 0:00	October	2021	Home	GSW	1	11	5	1	1	5	34
2	2.0	10/22/2021 0:00	October	2021	Home	PHO	1	2	5	2	0	4	25
3	3.0	10/24/2021 0:00	October	2021	Home	MEM	1	6	6	2	2	1	19
4	NULL	10/26/2021 0:00	October	2021	Away	SAS	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	NULL	10/27/2021 0:00	October	2021	Away	OKC	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Case Study Objectives



Criteria	Bonus Amount	Achieved
Averages 30 points in each game where they are active	500k	yes
Plays at least 65% of the season	500k	yes
Plays against every team at least once during the season	100k	no
Plays in more home games than away games	250k	yes
Plays in at least one game every month of the season	50k	yes