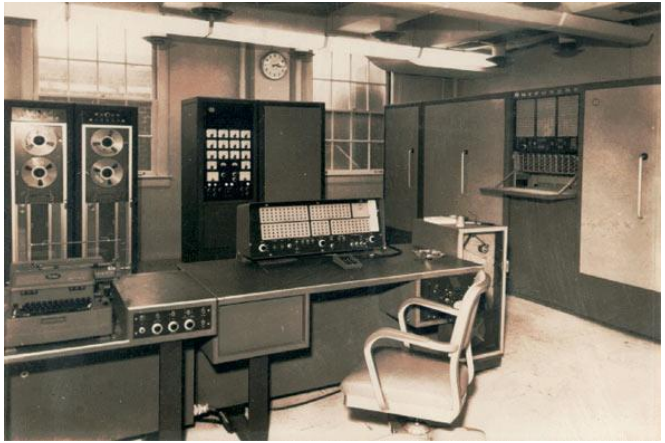
Advanced SQL I

Agenda - Schedule

1. SQL Leetcode Q
2. Window Functions
3. Break
4. COVID-19 SQL Lab



Database systems of the past

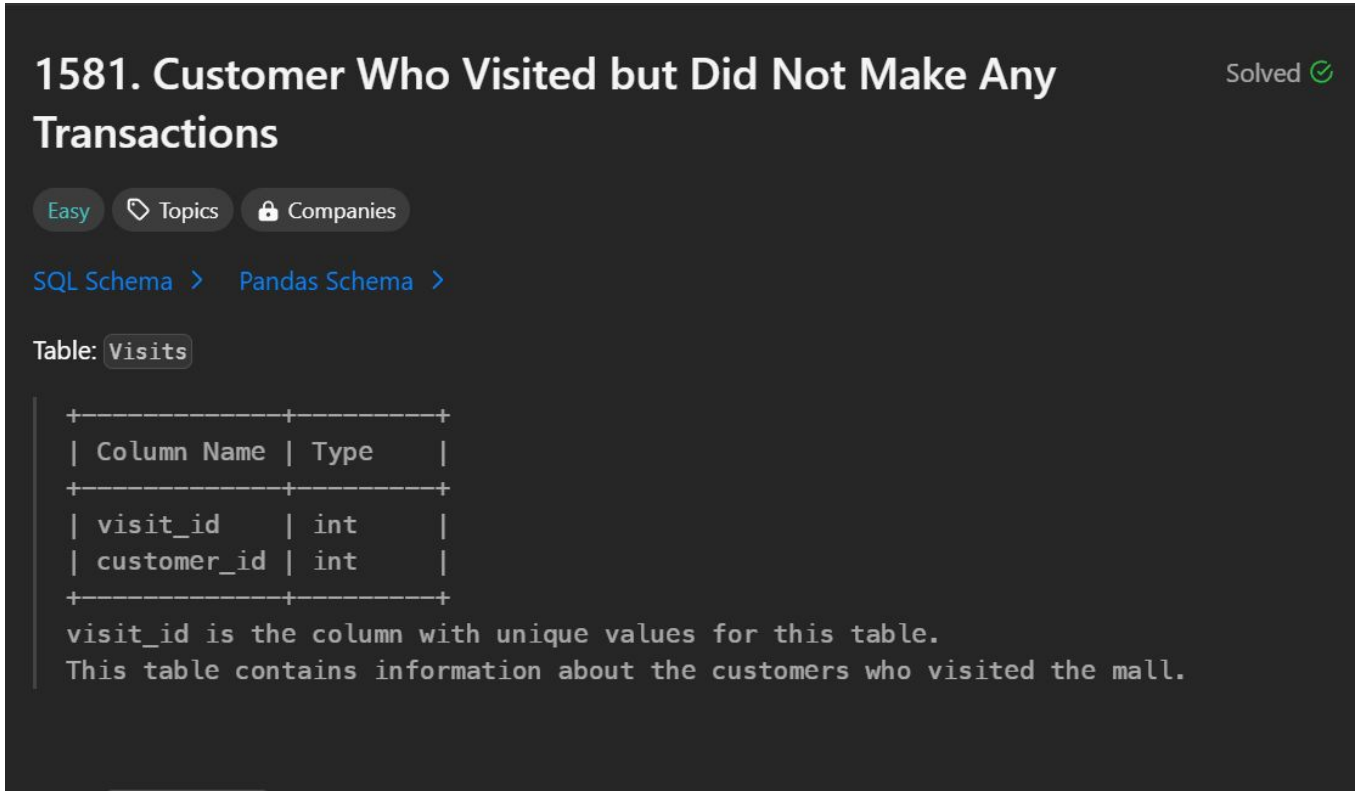
Agenda - Announcements

- **No pre-class quiz**
- **TLAB #3** due 5/14
  - **Early grade due date: 5/7**
  - **Extension due date: 5/13**
- In-class end of phase project is being released **THIS THURSDAY! (We recommend attending this review session)**

Agenda - Goals

- Use SQL window functions (ROW_NUMBER, RANK, DENSE_RANK, NTILE) to analyze and rank data within partitions.
- Apply LAG() and LEAD() functions to compare rows across a sequence.
- Use subqueries to calculate differences between values (e.g., current vs. previous order amounts).

SQL Leetcode Q

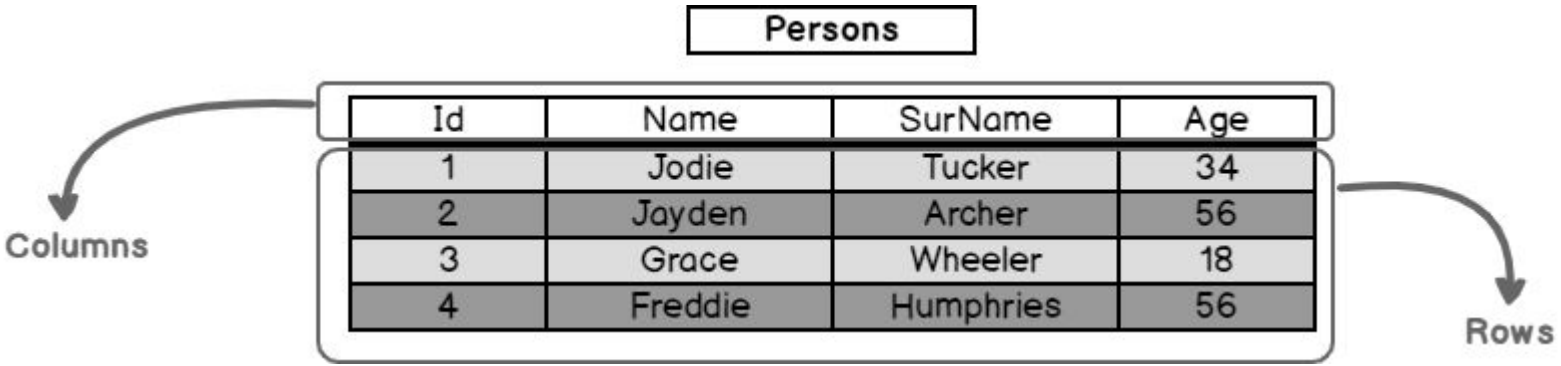


Take 5-10 minutes to complete Customers Who Visited but Did Not Make Any Transactions: https://leetcode.com/problems/customer-who-visited-but-did-not-make-any-transactions/description/?envType=study-plan-v2&envId=top-sql-50

SQL Review

SQL Overview/Review

Each column can have different types, here are some of the main ones:
INTEGER
VARCHAR(length) - this is equivalent to a Python string but we can set a maximum length
FLOAT
BOOL



SQL Overview/Review

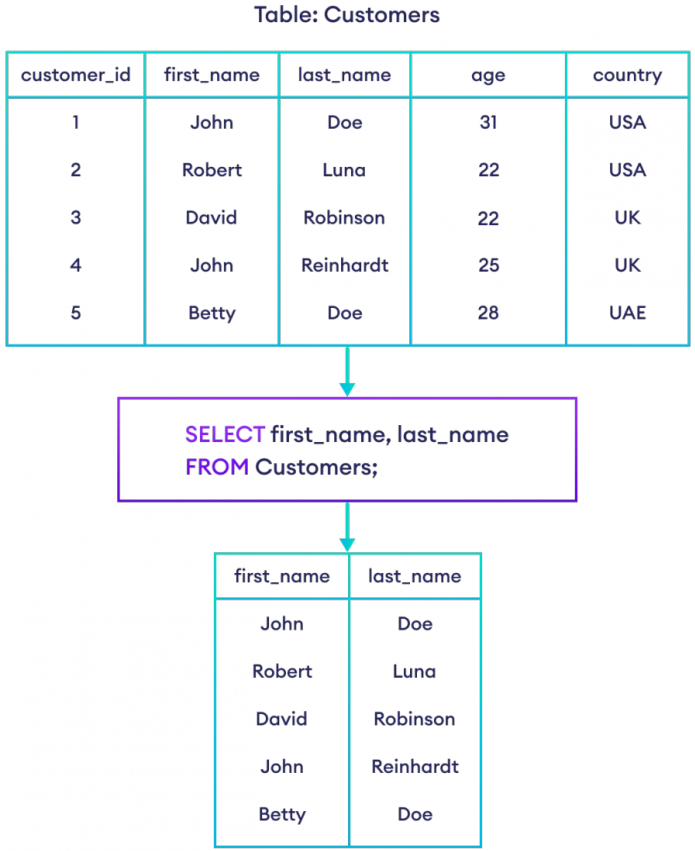
In order to get data from our table we MUST

SELECT *columns*
FROM *table*

When it is just *one* table, we don't need to specify the name

When we do **joins**, we need to specify the name if the columns names are shared

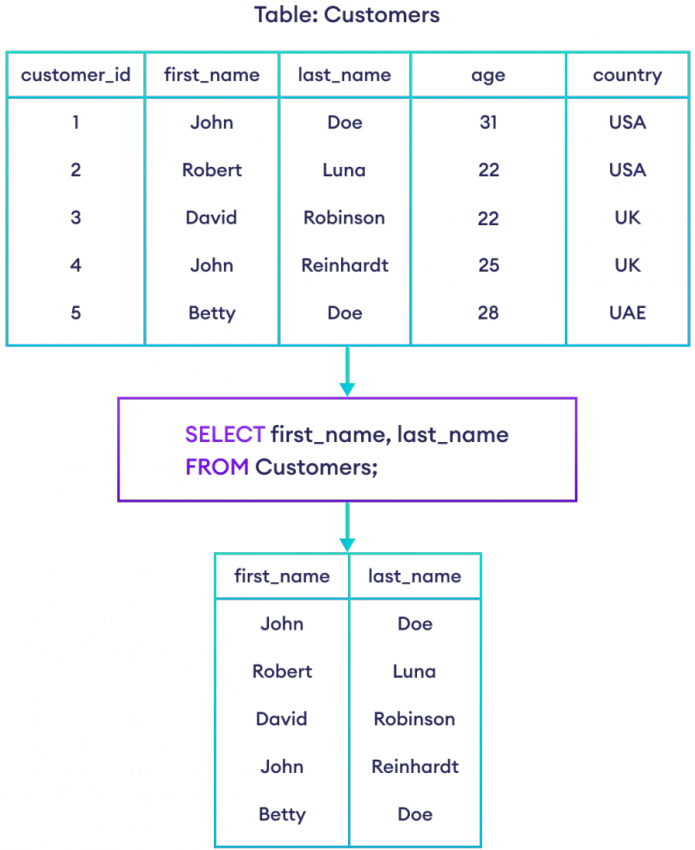
However, **every table must be explicitly named**



SQL Overview/Review

We can also alias columns **and** tables, so on the right we could o

```
SELECT first_name AS first,
       last_name AS last
FROM Customers AS c
```



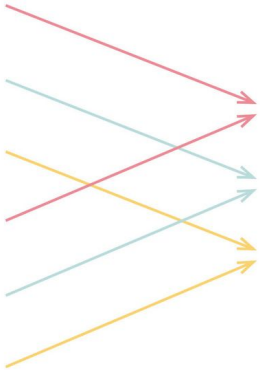
SQL GROUP BY

We use GROUP BY to combine (group) on column variables to get a result
The below query could be

```
SELECT genre,
SUM(qty)
FROM books
GROUP BY genre
```

Note how the genre column “collapses” into the unique values and that is added together

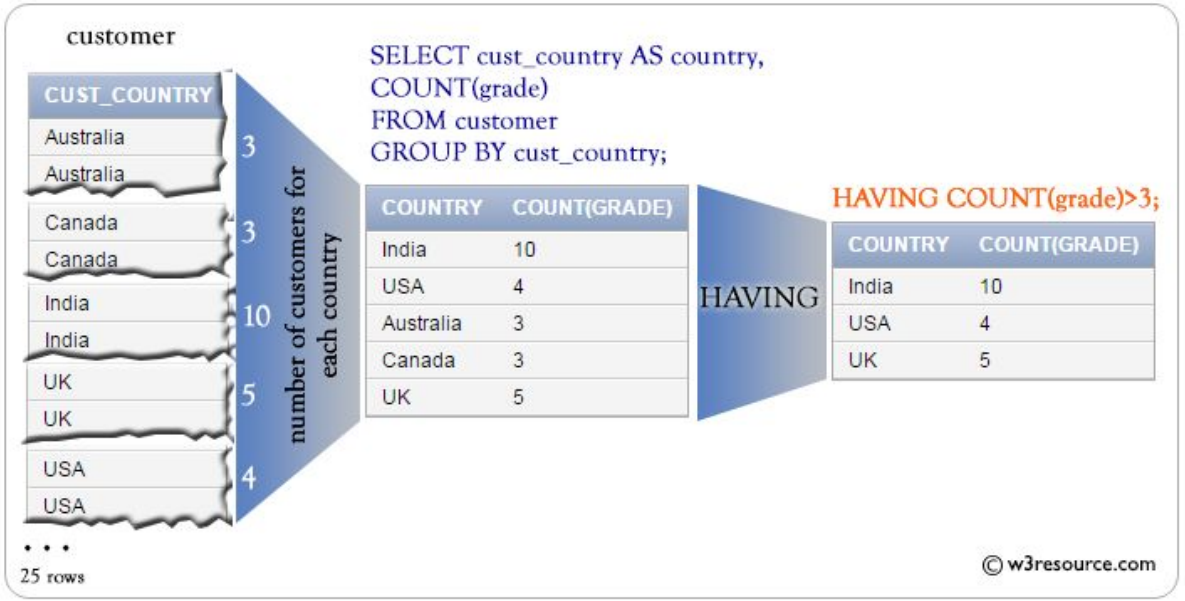
title	genre	qty
book 1	adventure	4
book 2	fantasy	5
book 3	romance	2
book 4	adventure	3
book 5	fantasy	3
book 6	romance	1



genre	total
adventure	7
fantasy	8
romance	3

SQL Filtering

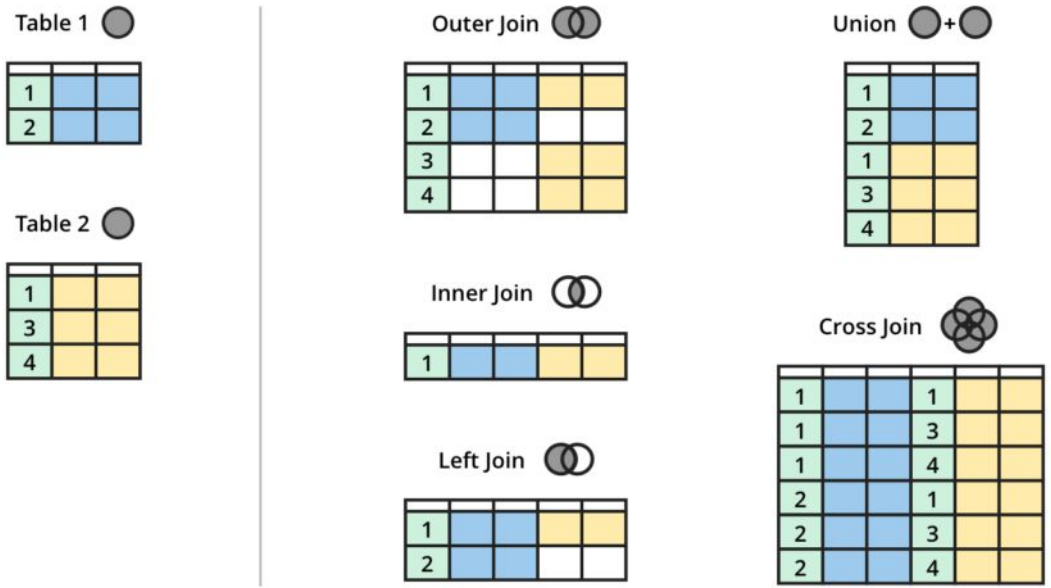
HAVING must be used with a **GROUP BY** statement, if we try to use **HAVING** without **GROUP BY** we will get an error



SQL JOINS

Combining Data Tables – SQL Joins Explained

A JOIN clause in SQL is used to combine rows from two or more tables, based on a related column between them.



SQL JOINS

We join by using a **Primary Key** from one table that is stored as a **Foreign Key** in another table

Here - the Customers Table has a customer_id, this is the **primary key** because it uniquely identifies each User

The Orders table has the **foreign key customer** because it relates to a **foreign table**: **customer**

We connect the primary key ID on Customers to Orders to get Customer Info on the order

Window Functions

Window Functions

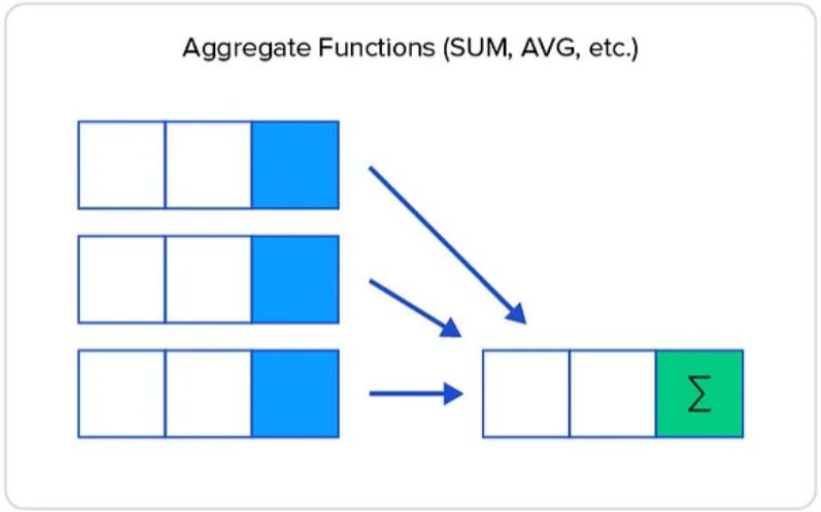
The idea behind window functions is that they look at our **data in sliding "windows"**

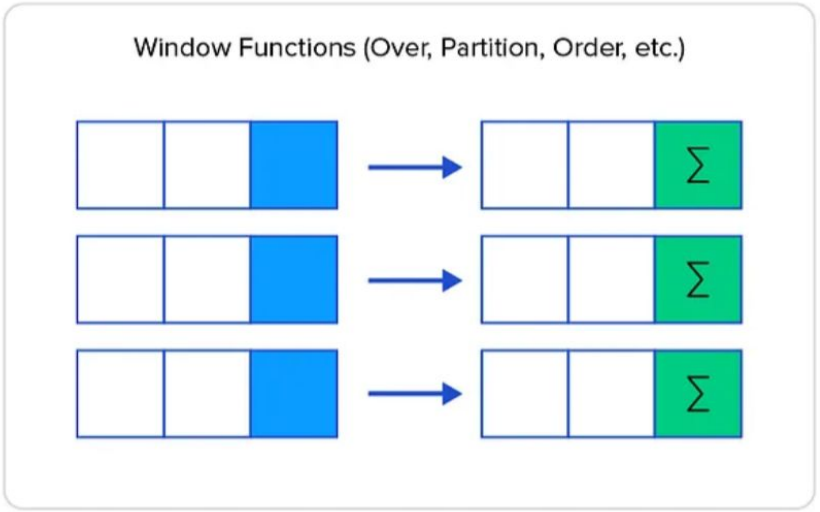
These windows can be time, categories, or more

The general concept is:

- decide on what we want to do: **SUM**, **RANK**, etc;
- decide on **group(s)** to **partition (group)** on: state, user id, etc;
- decide on **feature(s)** to order by: time, amount, etc;

Group by squashes rows. Window functions keep individual rows!





Notice that while this is similar to group-by, this is fundamentally a different concept!

Window Functions

Window functions are one of the more complicated parts of SQL but very powerful

Let's talk about their overall syntax first:

PARTITION BY = group by, this is the category we are interested in measuring

ORDER BY = the sorting order, very important! this will determine the order of data

```
SELECT *,

COUNT(customer_id)  OVER (PARTITION BY country
ORDER BY age) AS running_total

FROM Customers;
```

Window Functions

We want the **total customers by country**, as we **order by customer_id**

this gives us a **running total of customers** over as we move forward in the **age** column

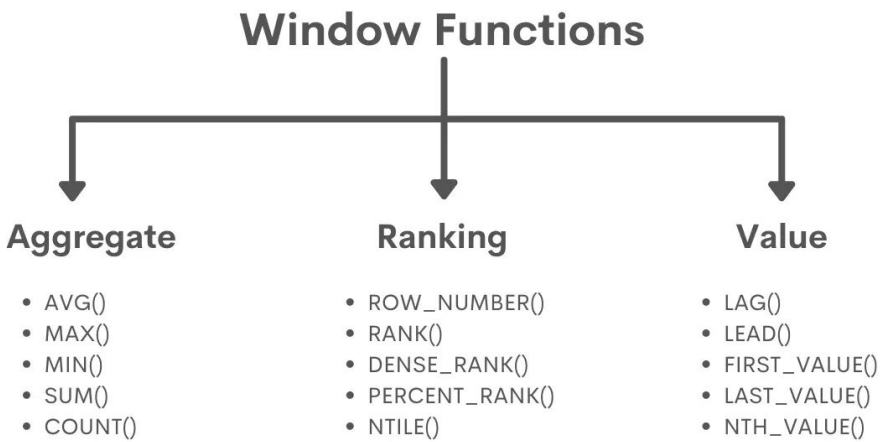
Note that we usually do this over a **date** column to calculate a "running total."

```
SELECT *,

COUNT(customer_id)  OVER (PARTITION BY country
ORDER BY age) AS running_total

FROM Customers;
```

Example Window Functions



ORDER BY vs PARTITION BY

We can treat **ORDER BY and PARTITION BY** as our “windows”

The **partition window** says “I am interested in the change by *category*”

The **order by window** says “I am interested in the change by *order*”

ORDER BY vs PARTITION BY

The partition window says “I am interested in the change by *category*”

- It works similar to a group by function
- by itself, it is not super powerful but we combine this with order by

The order by window says “I am interested in the change by *order*”

- do this calculator in a specific way
- by start date/time is common as well as by value amount

Ranking Window Functions

Ranking Functions

The power of window functions also lies in **ranking functions**

Things like **ROW_NUMBER, RANK, DENSE_RANK**, and so on
These functions allow us to know how our data is distributed, attach ranks to them, and use them in creative ways

We can now rank our data by our *partition* in a particular *order* which can let us answer questions like
“What is the longest ride by station?”

Different Ranking Functions

We'll cover the main functions:

ROW_NUMBER()
RANK()
DENSE_RANK()
NTILE()

```sql
SELECT *,
    ROW_NUMBER() OVER (ORDER BY age DESC) AS ROW_NUMBER
FROM Customers
;
```

Output

customer_id	first_name	last_name	age	country	ROW_NUMBER
1	John	Doe	31	USA	1
5	Betty	Doe	28	UAE	2
4	John	Reinhardt	25	UK	3
2	Robert	Luna	22	USA	4
3	David	Robinson	22	UK	5

Different Ranking Functions

We'll cover the main functions:

ROW_NUMBER()

- every time has **a unique row** number with no gaps or duplicates

RANK()

DENSE_RANK()

NTILE()

```sql
SELECT *,
    ROW_NUMBER() OVER (ORDER BY age DESC) AS ROW_NUMBER
FROM Customers
;
```

Output

customer_id	first_name	last_name	age	country	ROW_NUMBER
1	John	Doe	31	USA	1
5	Betty	Doe	28	UAE	2
4	John	Reinhardt	25	UK	3
2	Robert	Luna	22	USA	4
3	David	Robinson	22	UK	5

Different Ranking Functions

We'll cover the main functions:

ROW_NUMBER()

RANK()

- Items are ranked and we are allowing ties
- we will skip numbers in the event of ties

DENSE_RANK()

NTILE()

```sql
SELECT *,
    RANK() OVER (ORDER BY amount DESC) AS ROW_NUMBER
FROM Orders
;
```

Output

order_id	item	amount	customer_id	ROW_NUMBER
3	Monitor	12000	3	1
1	Keyboard	400	4	2
4	Keyboard	400	1	2
2	Mouse	300	4	4
5	Mousepad	250	2	5

Different Ranking Functions

We'll cover the main functions:

ROW_NUMBER()

RANK()

DENSE_RANK()

- similar to rank where we allow for ties
- we **do *not* skip numbers** however, even in the event of a tie
- once tie ends, we go to the next number

NTILE()

```
SELECT *,
    DENSE_RANK() OVER (ORDER BY amount DESC) AS ROW_NUMBER
FROM Orders
;
```

Output

order_id	item	amount	customer_id	ROW_NUMBER
3	Monitor	12000	3	1
1	Keyboard	400	4	2
4	Keyboard	400	1	2
2	Mouse	300	4	3
5	Mousepad	250	2	4

Different Ranking Functions

We'll cover the main functions:
ROW_NUMBER()
RANK()
DENSE_RANK()
NTILE(n)

- will divide the data into n **even groups**
- will provide rank based on order of aggregation (1 is always first so choose ORDER BY DESC/ASC appropriately)

```
SELECT *,
    NTILE(3) OVER (ORDER BY amount DESC) AS ROW_NUMBER
FROM Orders
;
```

Output

order_id	item	amount	customer_id	ROW_NUMBER
3	Monitor	12000	3	1
1	Keyboard	400	4	1
4	Keyboard	400	1	2
2	Mouse	300	4	2
5	Mousepad	250	2	3

Different Ranking Functions

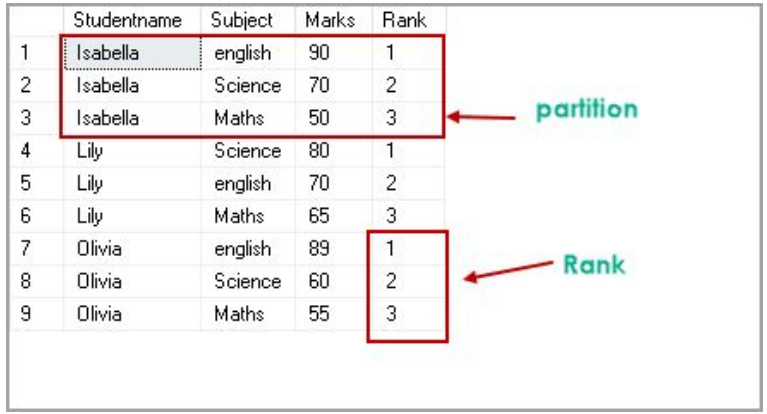
Notice that when you do a rank combined with a PARTITION it will rank ***each individual partition***

So it is important to properly define your partitions

Generally, you will not ORDER BY a group that you are partitioning by

```
SELECT Studentname,
       Subject,
       Marks,
       RANK() OVER(PARTITION BY Studentname ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Studentname,
       Rank;
```

	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Isabella	Science	70	2
3	Isabella	Maths	50	3
4	Lily	Science	80	1
5	Lily	english	70	2
6	Lily	Maths	65	3
7	Olivia	english	89	1
8	Olivia	Science	60	2
9	Olivia	Maths	55	3



LAG/LEAD Window Function

LAG and LEAD

LAG and LEAD are some of the more complicated window functions

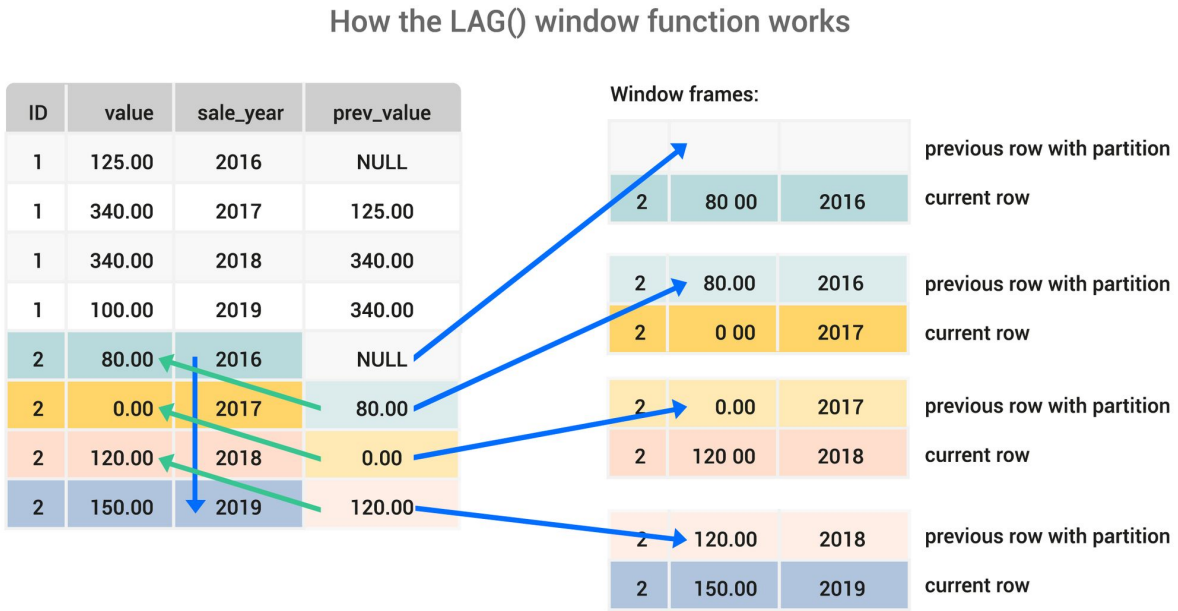
The idea is we are now interested in getting the data to interact with each other

Maybe we have data on a week-by-week basis and we want to calculate change
We would do a **LAG function** where we subtract our **current row with the row prior**

Let's first talk about how LAG/LEAD functions look then how we might implement

LAG and LEAD

```
SELECT ID, value, sale_year,
  LAG(value)
    OVER(
      PARTITION BY ID
      ORDER BY year
    ) AS prev_value
FROM sales;
```



LAG and LEAD

```
SELECT
   toy_name, month,
sale_value,
   LEAD(sale_value)
OVER(PARTITION BY toy_name
ORDER BY month)
     AS next_month_value
FROM toys_sale;
```

toy_name	month	sale_value	next_month_value
ball	3	45123	42000
ball	4	42000	20300
ball	5	20300	NULL
kite	3	6890	7600
kite	4	7600	9120
kite	5	9120	NULL
puzzle	5	67000	NULL
robot	3	23455	12345
robot	4	12345	23000
robot	5	23000	NULL

LAG and LEAD

When it comes to lag and lead, the limit is really your creativity

Just understand that it looks at periods before or periods after

There are a number of use cases like differences, averages, comparisons, etc;

```sql
SELECT *,
    LAG(amount) OVER (ORDER BY order_id) AS prev_order_amount
FROM Orders
;
```

Output

order_id	item	amount	customer_id	prev_order_amount
1	Keyboard	400	4	
2	Mouse	300	4	400
3	Monitor	12000	3	300
4	Keyboard	400	1	12000
5	Mousepad	250	2	400

For example, let's say we want to compare differences in order size across each order id. We could potentially implement the following. While we cannot perform additional arithmetic, **which SQL operation can we introduce to perform additional manipulation of this resultant table?**

```sql
SELECT r.order_id, r.amount - r.prev_order_amount
FROM (
  SELECT *,
      LAG(amount) OVER (ORDER BY order_id) AS prev_order_amount
  FROM Orders
) AS r
;
```

Output

order_id	r.amount - r.prev_order_amount
1	
2	-100
3	11700
4	-11600
5	-150

As we learned last week, we could introduce this query as a subquery of a another query to quickly calculate the difference between the current amount and the previous order amount column.

Window Function Nuance

ROWS PRECEDING AND ROWS FOLLOWING

Something to note is you may see something like “ROWS PRECEDING” and “ROWS FOLLOWING”

This is used to determine larger sliding windows

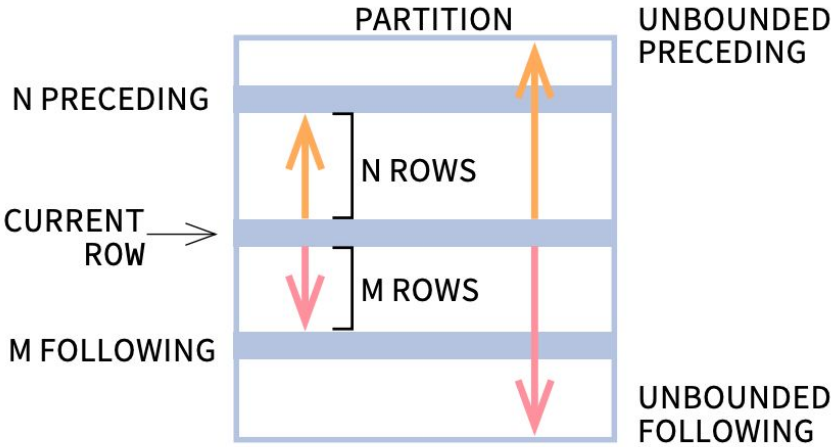
You may also see “ROWS BETWEEN UNBOUNDED PRECEDING AND 2 FOLLOWING” or “ROWS BETWEEN 2 PRECEDING AND UNBOUNDED FOLLOWING”

ROWS PRECEDING AND ROWS FOLLOWING

This simply determines our sliding window

This could be useful if we're doing things like moving averages

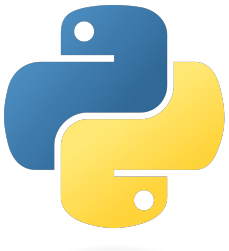
See how this impacts our partition:

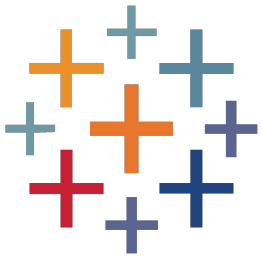


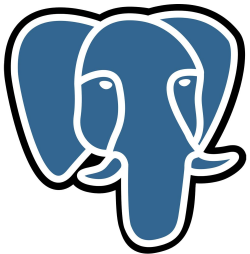
Window Functions - SQL COVID-19

SQL

To practice using LAG functions, check out the following [COVID-19 Case Study]







Tuesday

SQL + Python

- *How do we design a database?*
- *How do we use SQLite in the command line?*
- *How do we use SQLite in Python?*