Imperial College
Business School

Imperial means
Intelligent Business

Lecture 5
GROUP BY
and window functions

Dr Fintan Nagle
f.nagle@imperial.ac.uk

Reading

Video lectures:

3.3.1 - Theory of GROUP BY.mp4
3.3.2 - Making GROUP By queries.mp4
3.4 - Making ORDER BY queries.mp4

6.7 - Understanding window functions.mp4
6.8 - Using window functions.mp4

Postgres documentation on window functions:
https://www.postgresql.org/docs/10/static/tutorial-window.html

GROUP BY

GROUP BY

So far, we have looked at queries on individual rows. We have filtered out some rows using WHERE, and joined rows to other rows – but rows were treated individually.

GROUP BY allows us to put rows in groups and then perform operations on the whole group.

The most common operations are **aggregate functions** – functions like **MIN**, **MAX**, **MEAN** and **SUM**.

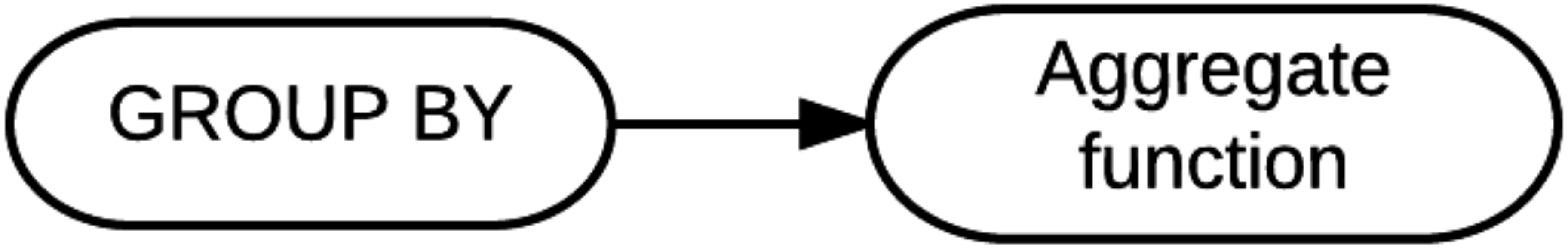
Aggregate functions take a group of rows and produce **one number**.

Aggregate functions

We can use aggregate functions on an **entire table** very simply:

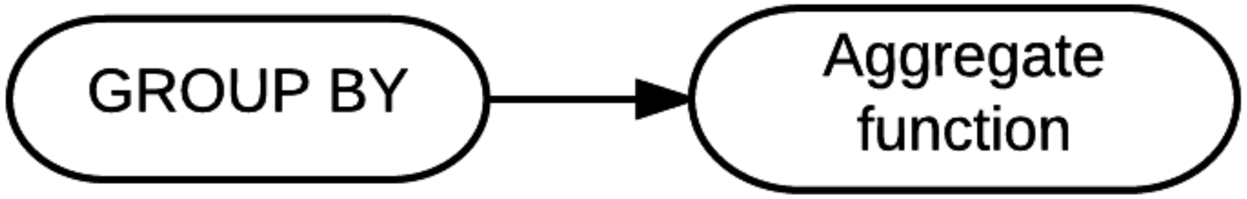
```
SELECT COUNT(*)
FROM film

SELECT MAX(price)
FROM products

SELECT AVG(salary)
FROM employees
```

Putting things into groups



```
COUNT
SUM
AVG
MAX
MIN
```

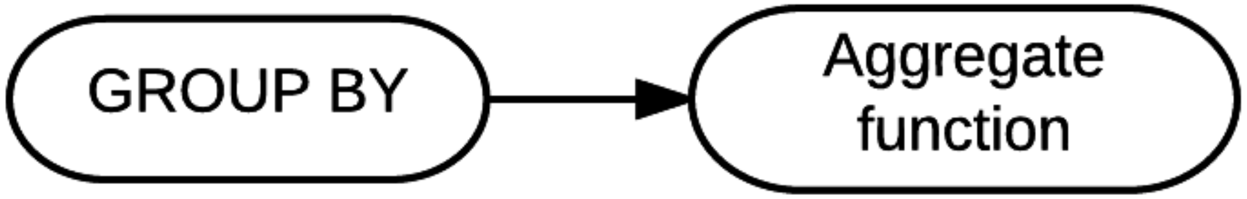
Putting things into groups



dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

Putting things into groups

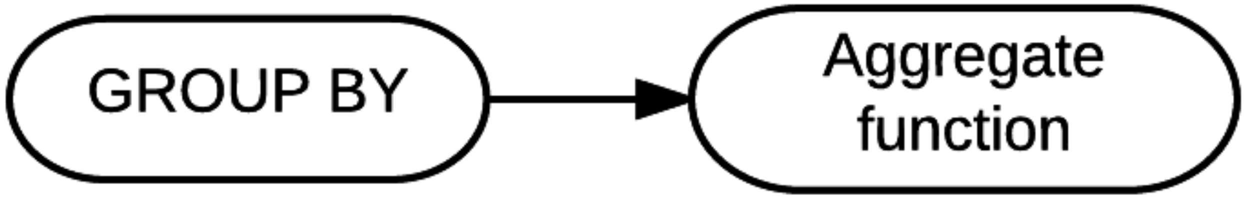


dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```
SELECT role, COUNT(*)
FROM dogs
GROUP BY role
```

Putting things into groups



dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```sql
SELECT role, COUNT(*)
FROM dogs
GROUP BY role
```

working

Artemis, Roger, Buddy, Shep

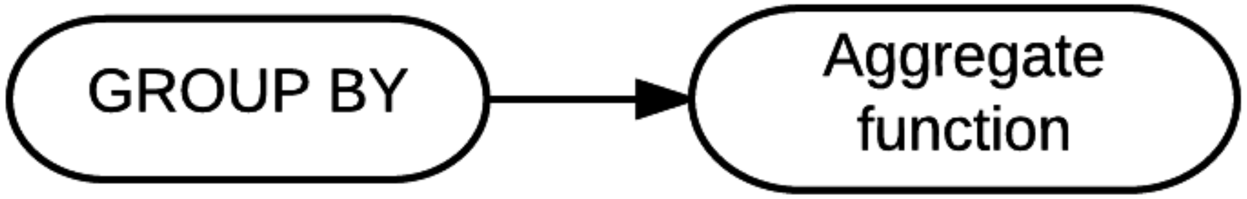
companion

Fido, Rover

show

Daisy

Putting things into groups



dogs

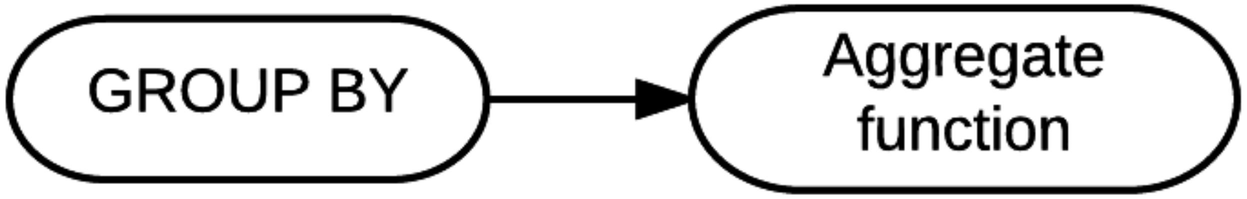
name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```
SELECT role, COUNT(*)
FROM dogs
GROUP BY role
```

result

role	
working	4
companion	2
show	1

Putting things into groups



GROUP BY

Aggregate function

dogs

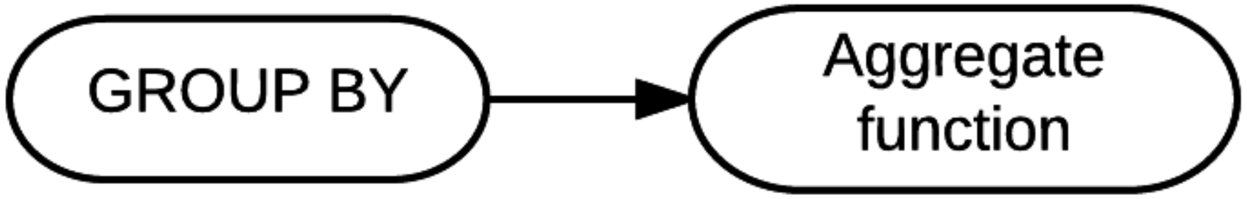
name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```sql
SELECT role, COUNT(*) AS dog_count
FROM dogs
GROUP BY role
```

result

role	dog_count
working	4
companion	2
show	1

Putting things into groups



dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

why COUNT(*)?

```
SELECT role, COUNT(*) AS dog_count
FROM dogs
GROUP BY role
```

result

role	dog_count
working	4
companion	2
show	1

A note about COUNT

- **COUNT(*)** counts the number of rows, even if they have null cells

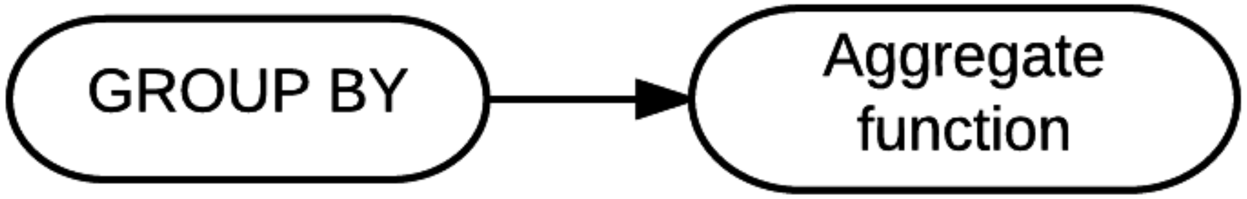
- **COUNT(column)** counts the number of rows where that column is not null

- **COUNT(DISTINCT column)** counts the number of distinct values in that column (nulls not counted)

- Usually, you either want **COUNT(*)** or **COUNT(DISTINCT column)**

- Whether you use **COUNT(*)** or **COUNT(column)** can be important in non-inner joins, where rows may have missing values.

Putting things into groups



dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

why COUNT(*)?

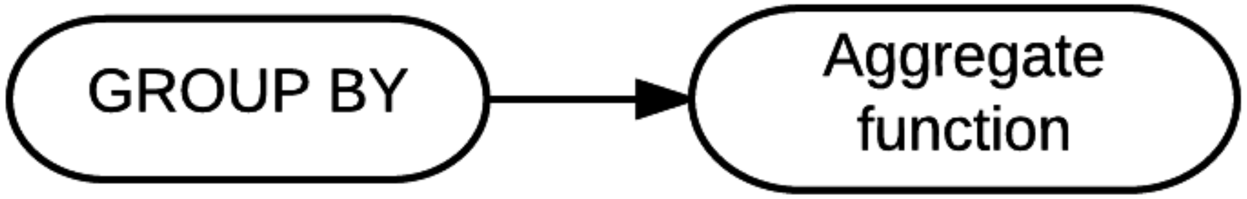
```
SELECT role, COUNT(*) AS dog_count
FROM dogs
GROUP BY role
```

because we're counting *rows*

result

role	dog_count
working	4
companion	2
show	1

Putting things into groups



dogs

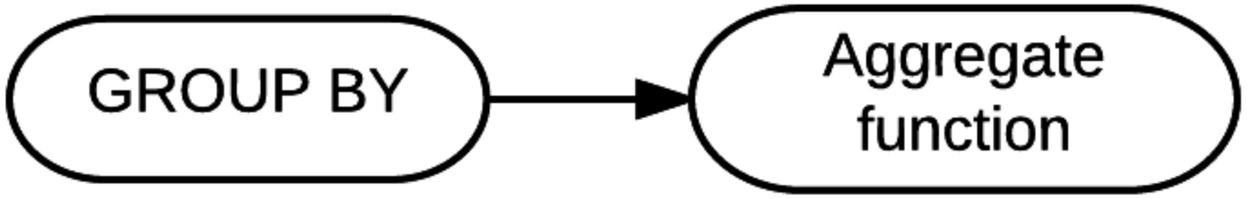
name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```
SELECT role, MAX(age) AS max_dog_age
FROM dogs
GROUP BY role
```

result

role	max_dog_age
working	10
companion	6
show	3

Putting things into groups



dogs

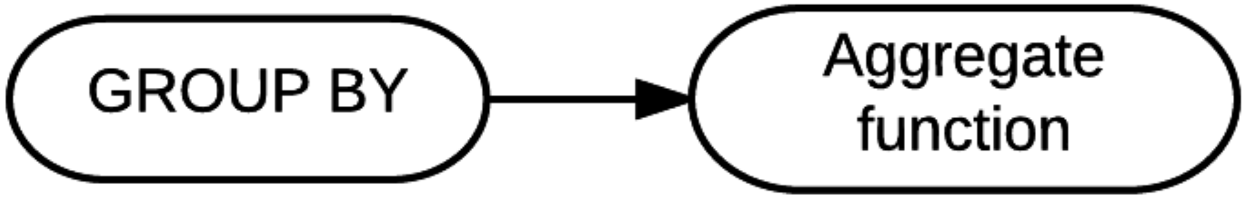
name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```sql
SELECT role, AVG(age) AS mean_dog_age
FROM dogs
GROUP BY role
```

result

role	mean_dog_age
working	5.5
companion	5
show	3

Putting things into groups



dogs

name	role	age
Artemis	working	2
Fido	companion	4
Rover	companion	6
Roger	working	3
Buddy	working	7
Daisy	show	3
Shep	working	10

```
SELECT role, SUM(age) AS total_dog_age
FROM dogs
GROUP BY role
```

result

role	total_dog_age
working	22
companion	10
show	3

HAVING

You can only use **HAVING** along with **GROUP BY**.

Starting with this query:

```
SELECT AVG(actor_1_facebook_likes) AS mean_likes
FROM movies
GROUP BY title_year
```

We can add a restriction with HAVING:

```
SELECT AVG(actor_1_facebook_likes) AS mean_likes
FROM movies
GROUP BY title_year
HAVING AVG(actor_1_facebook_likes) > 100
```

HAVING

We are not restricted to the columns (or aggregate function) used in **SELECT**:

```
SELECT AVG(actor_1_facebook_likes) AS mean_likes
FROM movies
GROUP BY title_year
HAVING MAX(actor_2_facebook_likes) > 100
```

However we can't use the new name – this will not run:

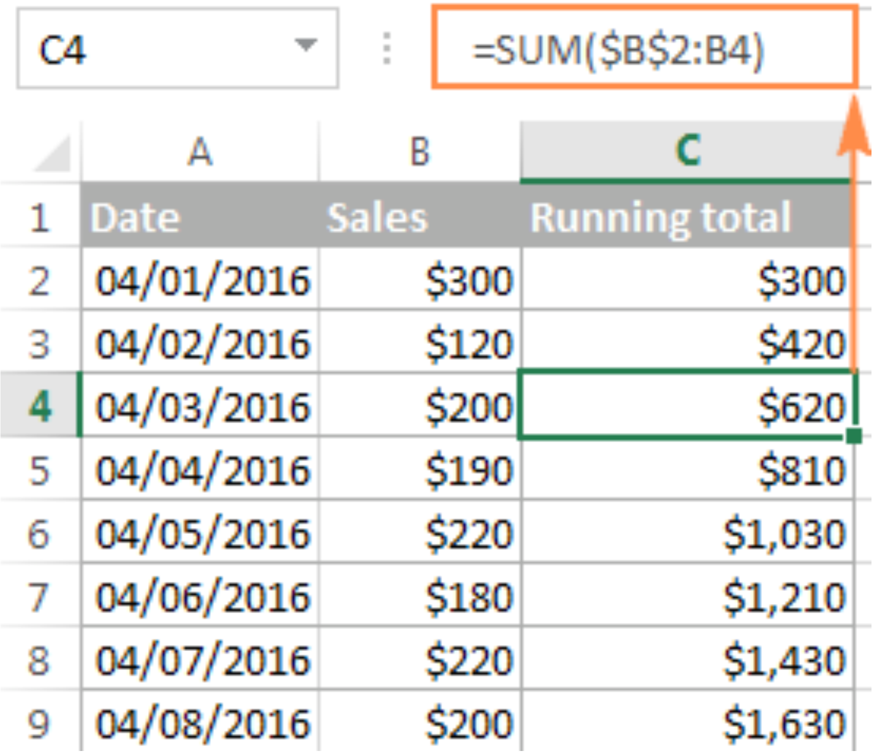
```
SELECT AVG(actor_1_facebook_likes) AS mean_likes
FROM movies
GROUP BY title_year
HAVING mean_likes > 100
```

We need to explicitly use the aggregate function.

Window functions

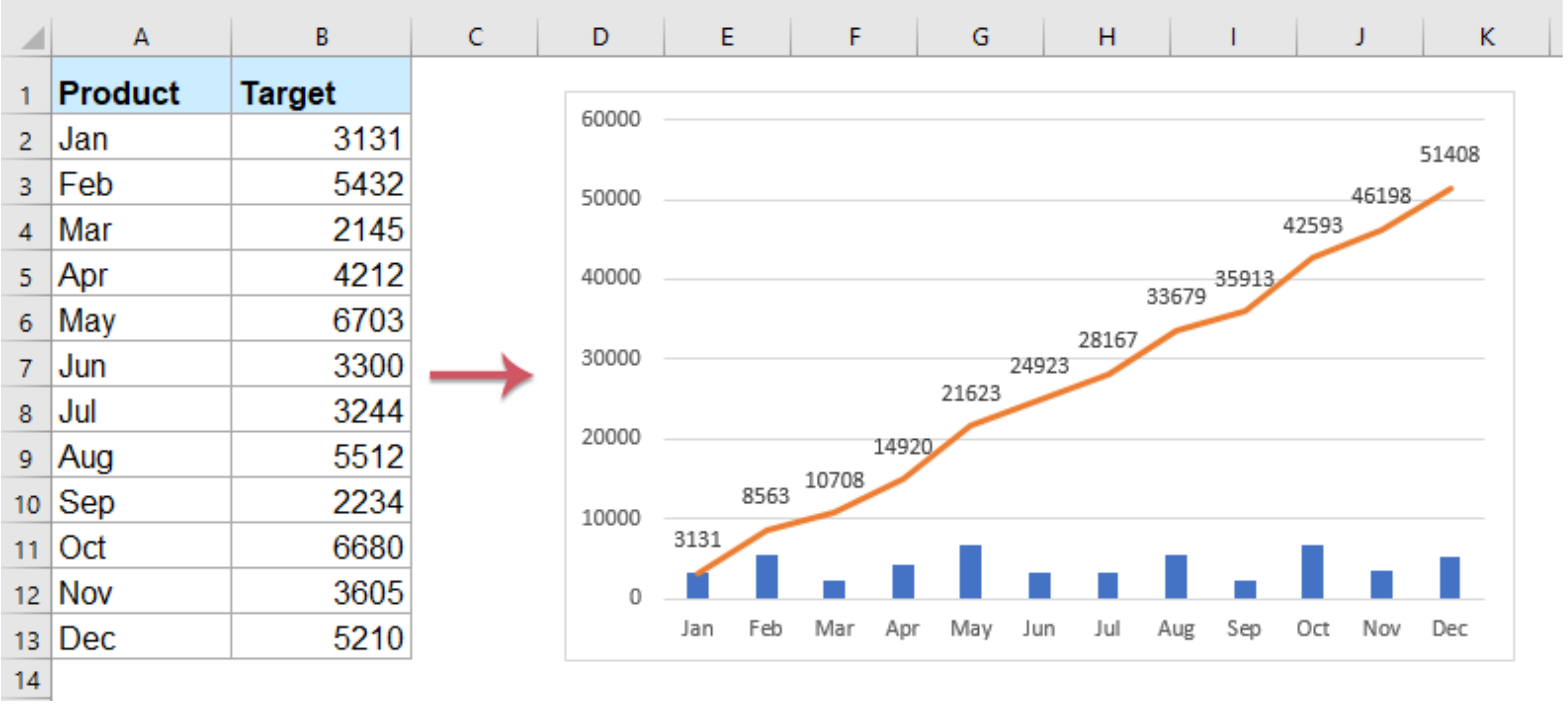
The running total (cumulative sum)

A running total (equivalently, cumulative sum)
 is a new column which adds up the figures in another column
as we work down the table.



C4 | =SUM(B2:B4)

	A	B	C
1	Date	Sales	Running total
2	04/01/2016	$300	$300
3	04/02/2016	$120	$420
4	04/03/2016	$200	$620
5	04/04/2016	$190	$810
6	04/05/2016	$220	$1,030
7	04/06/2016	$180	$1,210
8	04/07/2016	$220	$1,430
9	04/08/2016	$200	$1,630



	A	B
1	Product	Target
2	Jan	3131
3	Feb	5432
4	Mar	2145
5	Apr	4212
6	May	6703
7	Jun	3300
8	Jul	3244
9	Aug	5512
10	Sep	2234
11	Oct	6680
12	Nov	3605
13	Dec	5210
14		

Window functions

A window function adds a new column to our result table.

This column can either

- Work down the rows in a particular order,
  doing something as it goes along
  (this can be used to work out the running total)

- For each row, find a group of "related rows",
  and do something to that group.

Window functions

Window functions are always placed in the **SELECT** clause.

Setting up a window function is like asking to select another column;

the new column is defined (and constructed) by the window function.

The keyword **OVER** sets up a window function.

- The part before **OVER** is a function to be evaluated on each group.
- The part after **OVER** describes how the groups should be set up.

Running total of price in date order:

SUM(price) **OVER** (ORDER BY date)

Show the average salary of everyone else in the same department:

AVG(salary) **OVER** (PARTITION BY department_name)

Window functions

Using a window function is like adding a new column.

```
SELECT order_id,
SUM(price) OVER (ORDER BY date)
FROM orders;
```

Window functions

Using a window function is like adding a new column.

```
SELECT depname, empno, salary,
avg(salary) OVER (PARTITION BY depname)
FROM empsalary;
```

Window functions

```
SELECT depname, empno, salary, avg(salary)
OVER (PARTITION BY depname)
FROM empsalary;
```

OVER: shows that we're applying a window function

PARTITION BY: sets up the groups the window function shall apply over.

This is very similar to how an aggregate function behaves – but the final number of rows doesn't go down, as the results for each group are copied rather than being unified.

Window functions

We can also have both PARTITION BY and ORDER BY in the OVER section:

SELECT orderid, employeeid, orderdate,

SUM(freight) **OVER** (PARTITION BY employeeid ORDER BY orderdate) from orders

ORDER BY orderdate

This produces a separate running total for each employee.

What happens if you remove the final **ORDER BY**?

Window functions

The window function works down the table in a particular order.

For each row, it finds a set of *related rows*.

For an **ORDER BY** window function, the group starts off as the first row, then gets bigger by one row each time we move down another row (running total). **We have to specify an order.**

For a **PARTITION BY** window function, there is no order. Each row's related group is *somehow related* to the current row; **we have to specify how they are related.**

Window functions

A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

[From the Postgres documentation]

https://www.postgresql.org/docs/9.1/tutorial-window.html

To remember about window functions

- One of the most powerful and useful features of SQL
- Need either a PARTITION BY, or an ORDER BY, or both
- The evaluation order of the window function is NOT the same as the display order of the final results!

Window functions

- Window functions are permitted only in the **SELECT** list and the **ORDER BY** clause of the query. They are forbidden elsewhere, such as in **GROUP BY**, **HAVING** and **WHERE** clauses. This is because they logically execute after the processing of those clauses.
- Also, window functions execute after regular aggregate functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

[From the Postgres documentation]

Window functions

A window function call always contains an OVER clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a regular function or aggregate function. The OVER clause determines exactly how the rows of the query are split up for processing by the window function. The PARTITION BY list within OVER specifies dividing the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

[From the Postgres documentation]

Window functions and COUNT DISTINCT

Distinct is not implemented for window functions.

Question: dvdrental database

Show Eleanor Hunt’s rental history, with cumulative total of how much she has paid.

```
SELECT first_name, last_name, rental_date, amount,
SUM(amount) OVER(ORDER BY rental_date)
AS cumulative_amount

FROM
customer INNER JOIN rental
ON customer.customer_id = rental.customer_id
INNER JOIN payment
ON rental.rental_id = payment.rental_id
WHERE first_name='Eleanor' AND last_name='Hunt'
```

Question: dvdrental database

NOTE: the window function can be processed in a different order than that in which the final rows are displayed!

Here they are both in date order:

```
SELECT first_name, last_name, rental_date, amount,
SUM(amount) OVER(ORDER BY rental_date)
AS cumulative_amount
FROM
customer INNER JOIN rental
ON customer.customer_id = rental.customer_id
INNER JOIN payment
ON rental.rental_id = payment.rental_id
WHERE first_name='Eleanor' AND last_name='Hunt’
ORDER BY rental_date
```

Question: dvdrental database

NOTE: the window function can be processed in a different order than that in which the final rows are displayed!

Here the window function goes in date order but the display is by amount:

```sql
SELECT first_name, last_name, rental_date, amount,
SUM(amount) OVER(ORDER BY rental_date)
AS cumulative_amount
FROM
customer INNER JOIN rental
ON customer.customer_id = rental.customer_id
INNER JOIN payment
ON rental.rental_id = payment.rental_id
WHERE first_name='Eleanor' AND last_name='Hunt’
ORDER BY amount
```

LEAD and LAG

You can access the next or previous rows with LEAD and LAG:

```
SELECT film_id, title, rental_rate,
LEAD(rental_rate) OVER (ORDER BY film_id) AS next_price,
LAG(rental_rate, 3) OVER (ORDER BY film_id) AS three_prices_ago
FROM film
```

LEAD(col): next row
LAG(col): previous row

LEAD(col, *n*): *n* rows ahead
LAG(col, *n*): *n* rows behind

These only refer to the execution order of the window function, not the order in which the results are displayed.