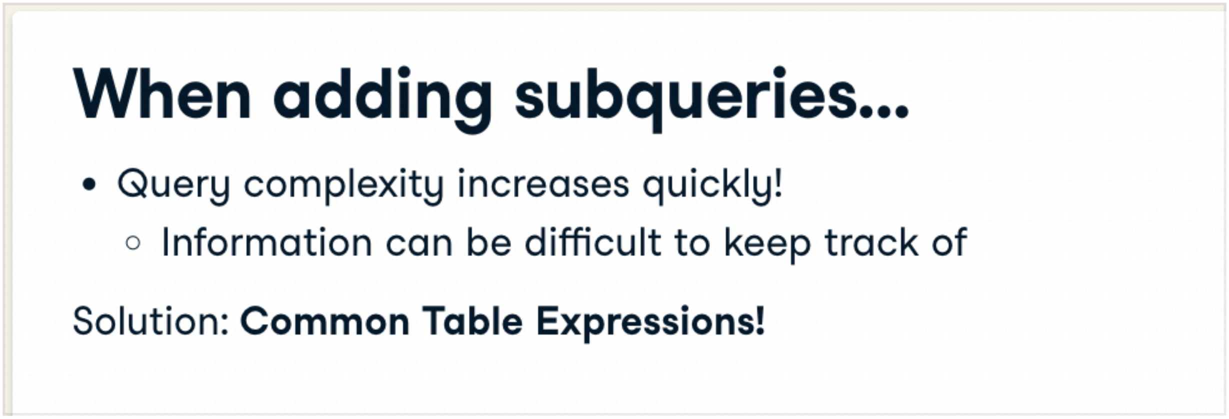
Common Table Expressions

Intermediate SQL - 2

Common table expressions are declared using the WITH Statement

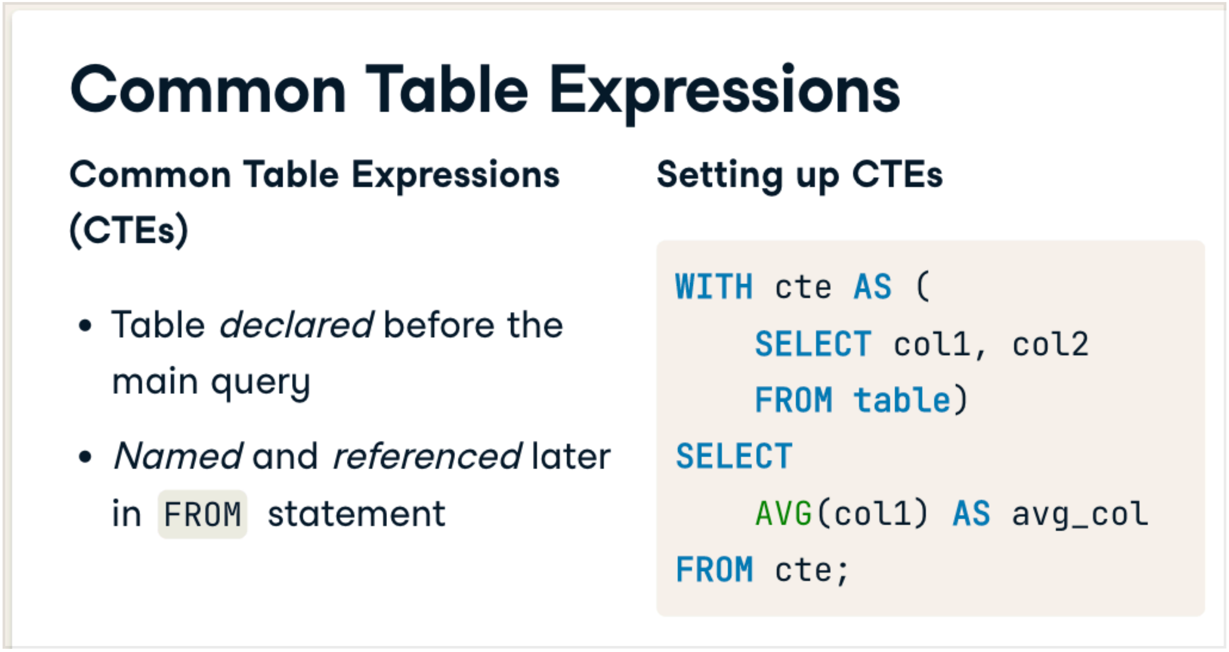
- We declare the common table expression before the query and then use it as we would use a table in the from statement.



When adding subqueries...

- Query complexity increases quickly!
  - Information can be difficult to keep track of

Solution: **Common Table Expressions!**



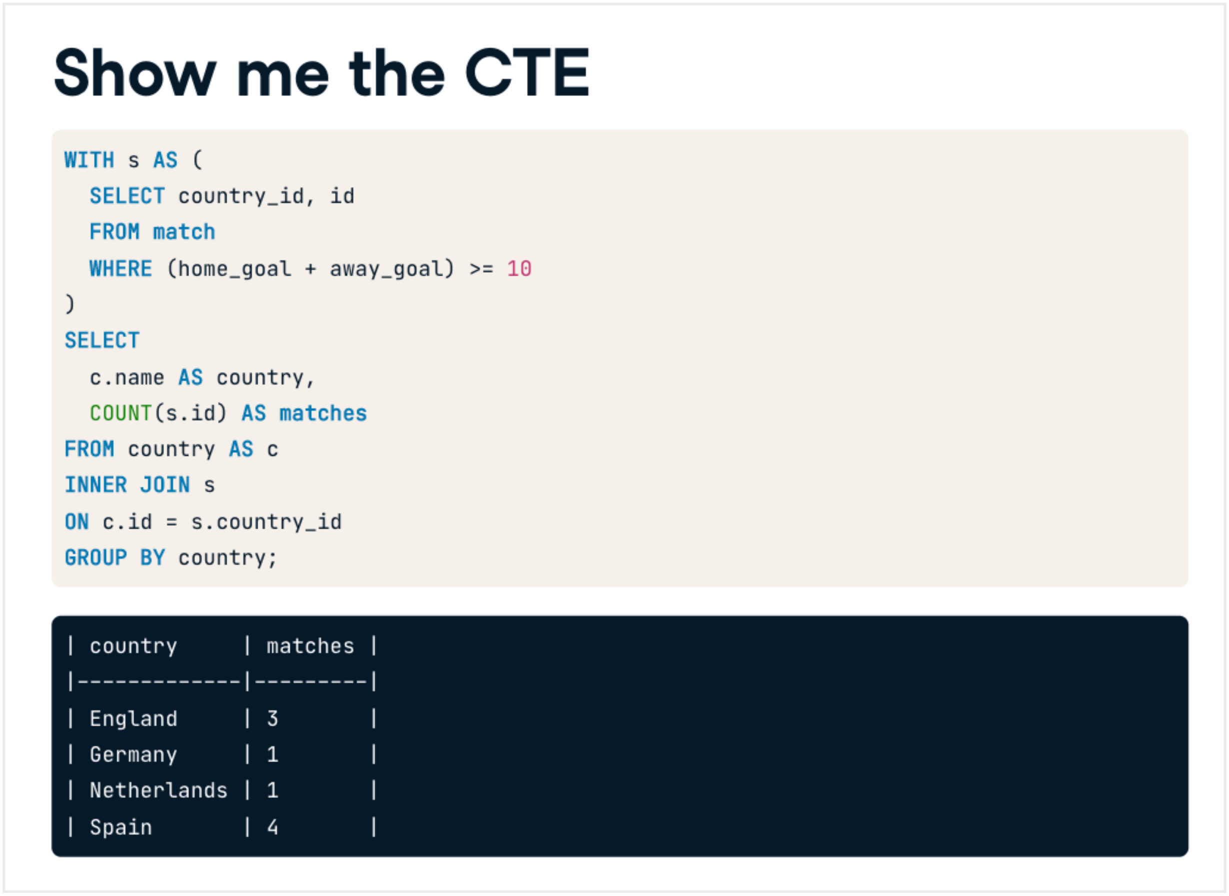
Common Table Expressions

Common Table Expressions (CTEs)

- Table *declared* before the main query
- *Named* and *referenced* later in `FROM` statement

Setting up CTEs

```
WITH cte AS (
    SELECT col1, col2
    FROM table)
SELECT
    AVG(col1) AS avg_col
FROM cte;
```

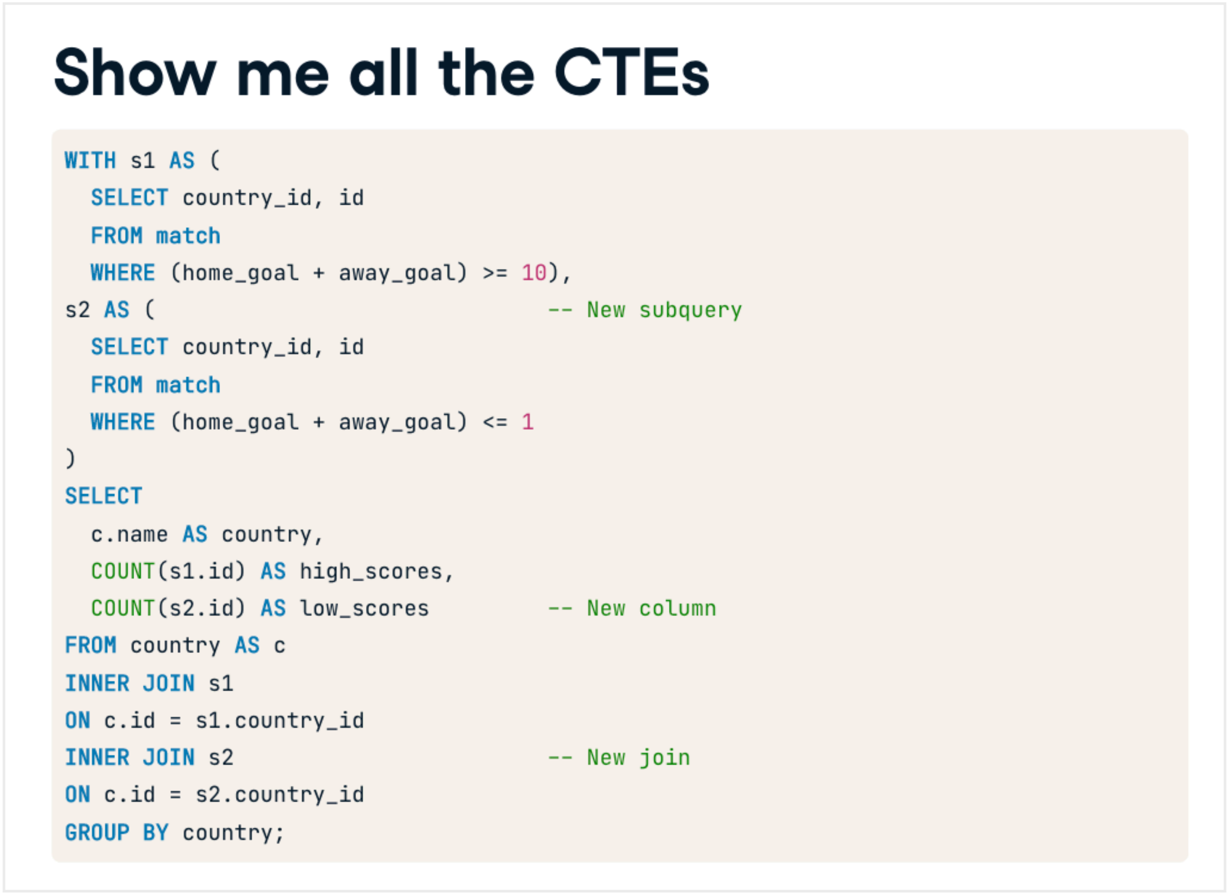


Show me the CTE

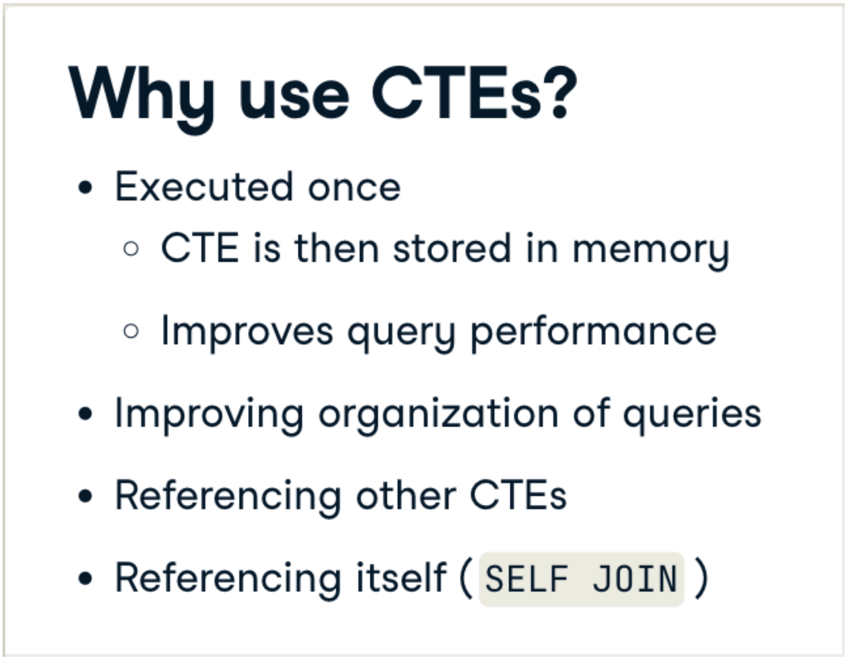
```sql
WITH s AS (
  SELECT country_id, id
  FROM match
  WHERE (home_goal + away_goal) >= 10
)
SELECT
  c.name AS country,
  COUNT(s.id) AS matches
FROM country AS c
INNER JOIN s
ON c.id = s.country_id
GROUP BY country;
```

```
| country     | matches |
|-------------|---------|
| England     | 3       |
| Germany     | 1       |
| Netherlands | 1       |
| Spain       | 4       |
```

We can list all the CTEs one after another with a comma departing them
Note: There is only one WITH keyword before all the CTEs


Show me all the CTEs

```
WITH s1 AS (
  SELECT country_id, id
  FROM match
  WHERE (home_goal + away_goal) >= 10),
s2 AS (                                  -- New subquery
  SELECT country_id, id
  FROM match
  WHERE (home_goal + away_goal) <= 1
)
SELECT
  c.name AS country,
  COUNT(s1.id) AS high_scores,
  COUNT(s2.id) AS low_scores             -- New column
FROM country AS c
INNER JOIN s1
ON c.id = s1.country_id
INNER JOIN s2                            -- New join
ON c.id = s2.country_id
GROUP BY country;
```


Why use CTEs?

- Executed once
  - CTE is then stored in memory
  - Improves query performance
- Improving organization of queries
- Referencing other CTEs
- Referencing itself (`SELF JOIN`)

Deciding on techniques to use

Differentiating Techniques

Joins

- Combine 2+ tables
  - Simple operations/aggregations

Correlated Subqueries

- Match subqueries & tables
  - Avoid limits of joins
  - **High processing time**

Multiple/Nested Subqueries

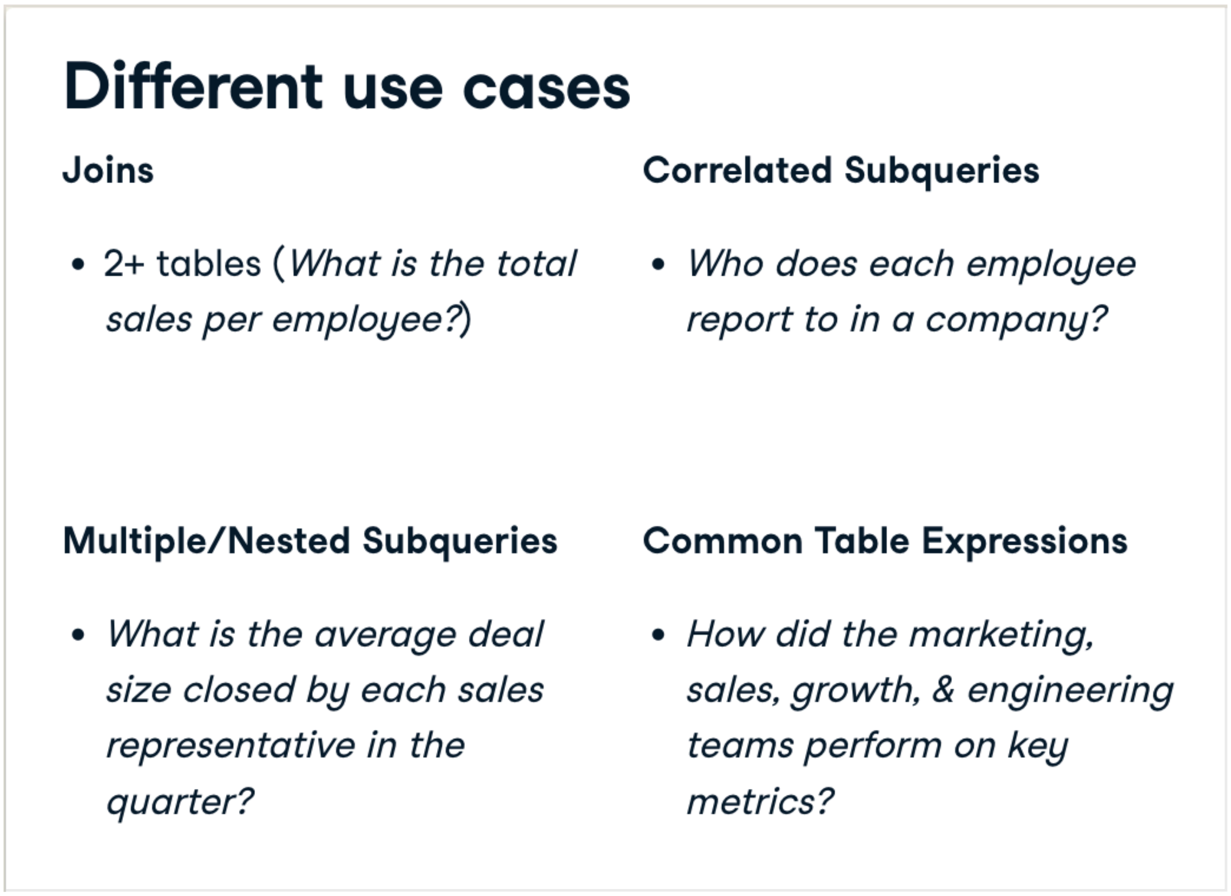
- Multi-step transformations
  - Improve accuracy and reproducibility

Common Table Expressions

- Organize subqueries sequentially
- Can reference other CTEs

So which do I use?

- Depends on your database/question
- The technique that best allows you to:
  - Use and reuse your queries
  - Generate clear and accurate results



Different use cases

Joins

- 2+ tables (*What is the total sales per employee?*)

Correlated Subqueries

- *Who does each employee report to in a company?*

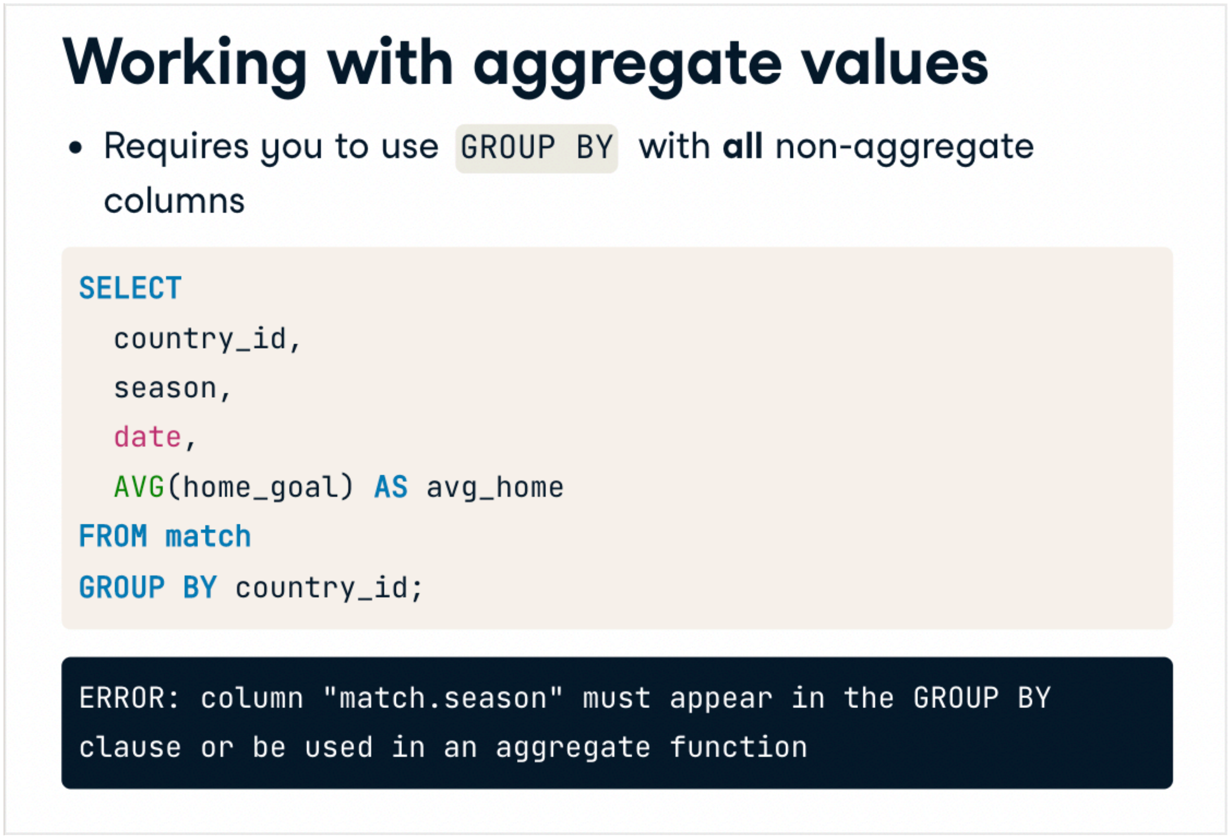
Multiple/Nested Subqueries

- *What is the average deal size closed by each sales representative in the quarter?*

Common Table Expressions

- *How did the marketing, sales, growth, & engineering teams perform on key metrics?*

WINDOW FUNCTIONS

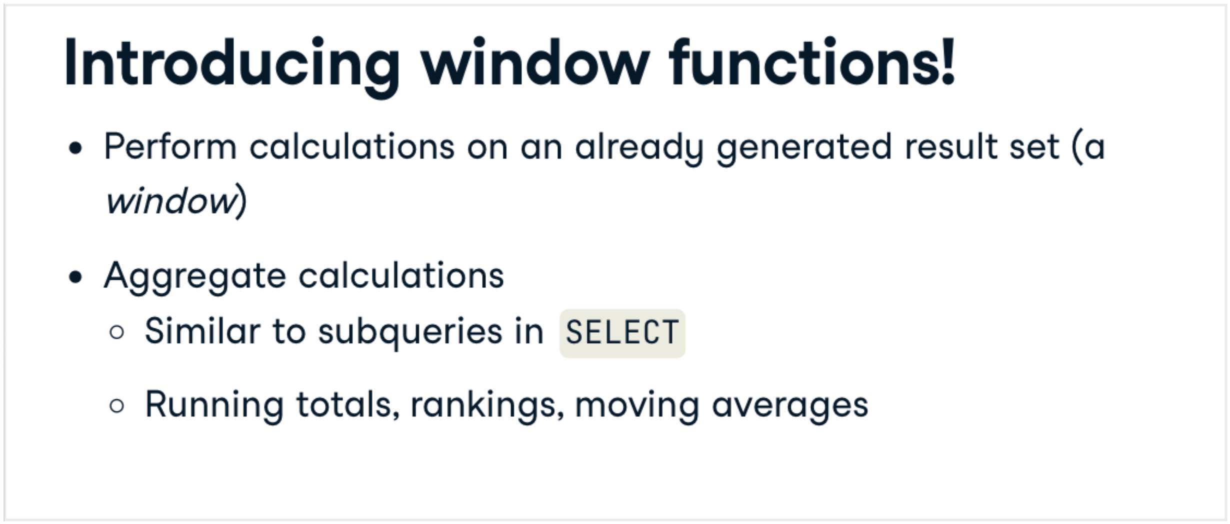


Working with aggregate values

- Requires you to use `GROUP BY` with **all** non-aggregate columns

```
SELECT
  country_id,
  season,
  date,
  AVG(home_goal) AS avg_home
FROM match
GROUP BY country_id;
```

```
ERROR: column "match.season" must appear in the GROUP BY
clause or be used in an aggregate function
```

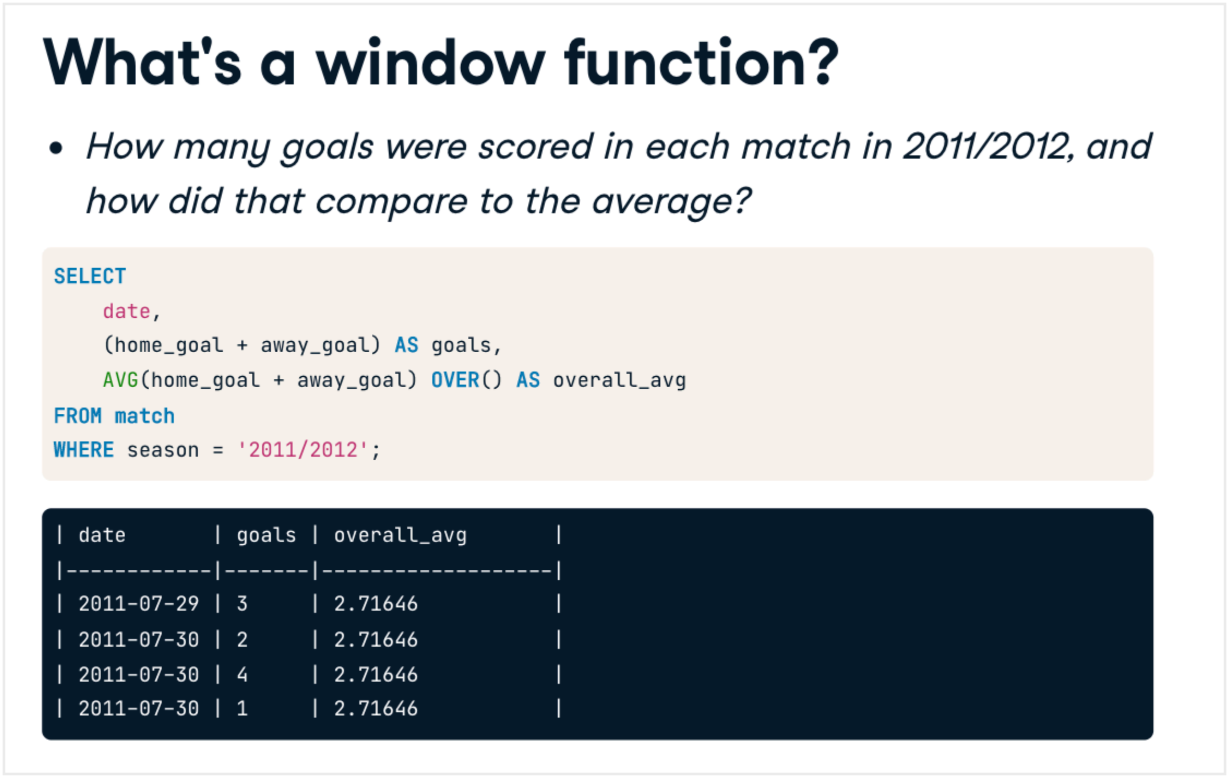


Introducing window functions!

- Perform calculations on an already generated result set (a *window*)
- Aggregate calculations
  - Similar to subqueries in `SELECT`
  - Running totals, rankings, moving averages

OVER()

- Used to calculate a result over the whole table/dataframe



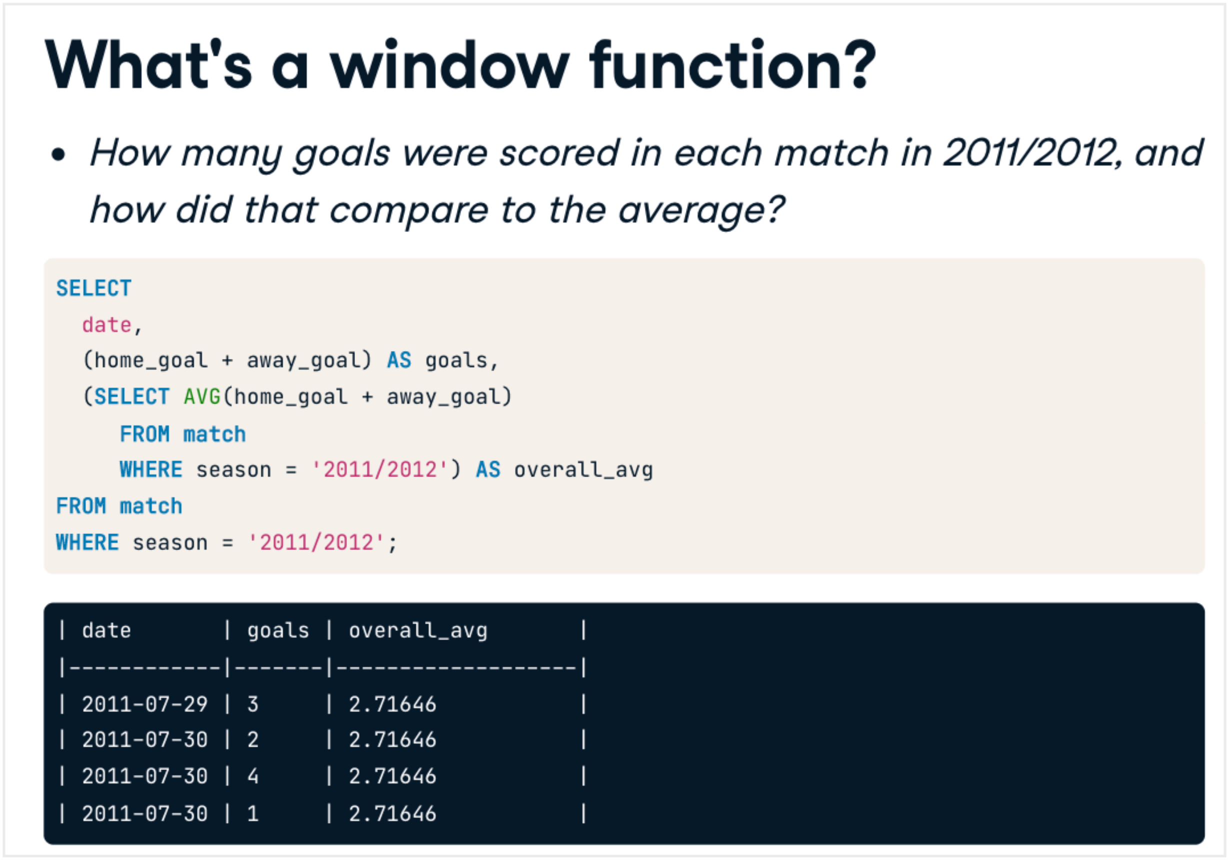
What's a window function?

- *How many goals were scored in each match in 2011/2012, and how did that compare to the average?*

```
SELECT
    date,
    (home_goal + away_goal) AS goals,
    AVG(home_goal + away_goal) OVER() AS overall_avg
FROM match
WHERE season = '2011/2012';
```

```
| date       | goals | overall_avg       |
|------------|-------|-------------------|
| 2011-07-29 | 3     | 2.71646           |
| 2011-07-30 | 2     | 2.71646           |
| 2011-07-30 | 4     | 2.71646           |
| 2011-07-30 | 1     | 2.71646           |
```

- Simpler to use than subqueries and faster than subqueries
- When a subquery was used we would have computed the same result for all the rows, and by using the over function we can make the calculation such that it is calculated only once.



What's a window function?

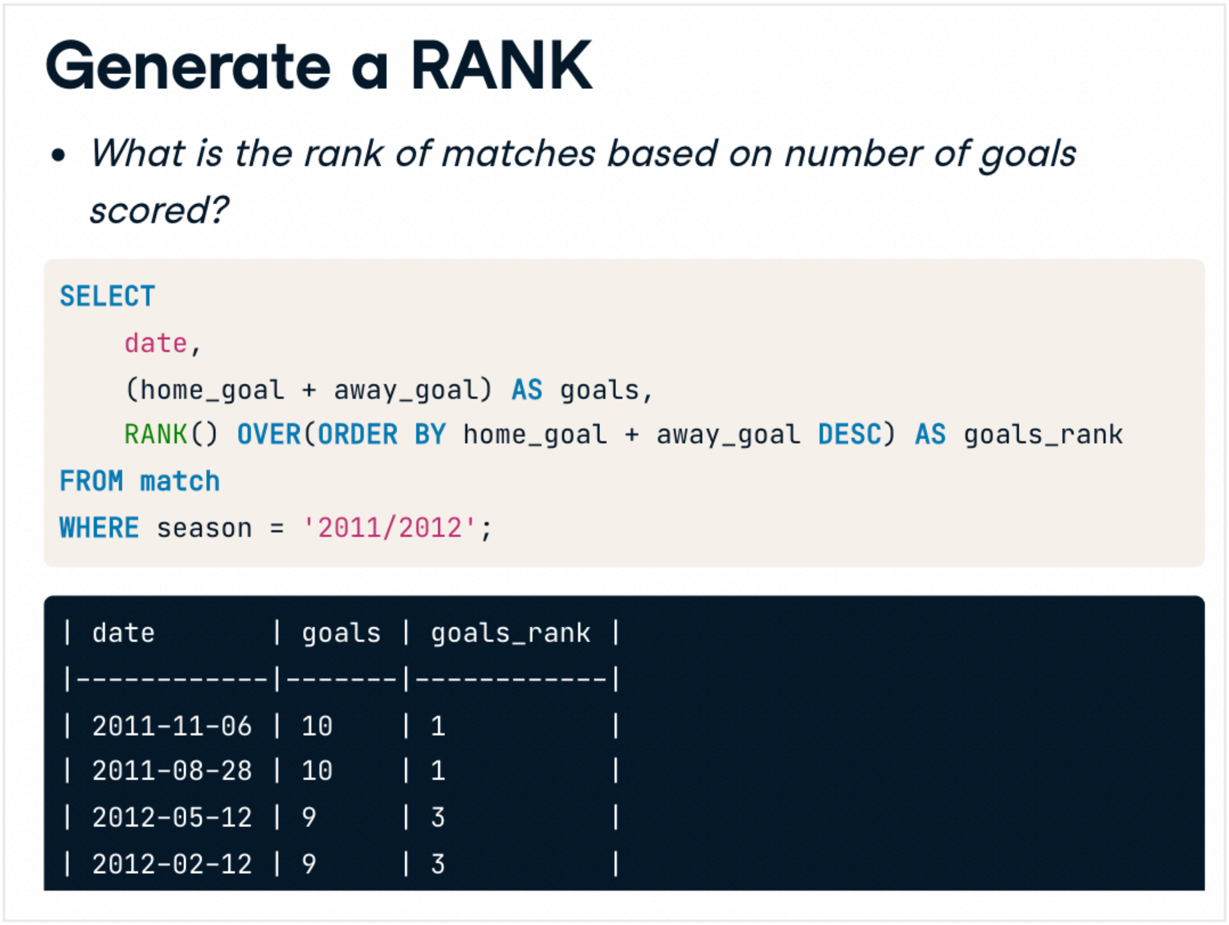
- *How many goals were scored in each match in 2011/2012, and how did that compare to the average?*

```
SELECT
  date,
  (home_goal + away_goal) AS goals,
  (SELECT AVG(home_goal + away_goal)
     FROM match
     WHERE season = '2011/2012') AS overall_avg
FROM match
WHERE season = '2011/2012';
```

```
| date       | goals | overall_avg       |
|------------|-------|-------------------|
| 2011-07-29 | 3     | 2.71646           |
| 2011-07-30 | 2     | 2.71646           |
| 2011-07-30 | 4     | 2.71646           |
| 2011-07-30 | 1     | 2.71646           |
```

RANK()

– Rank over the total goal in descending order. We execute this at the last once all the other parts of the query have been executed.



Generate a RANK

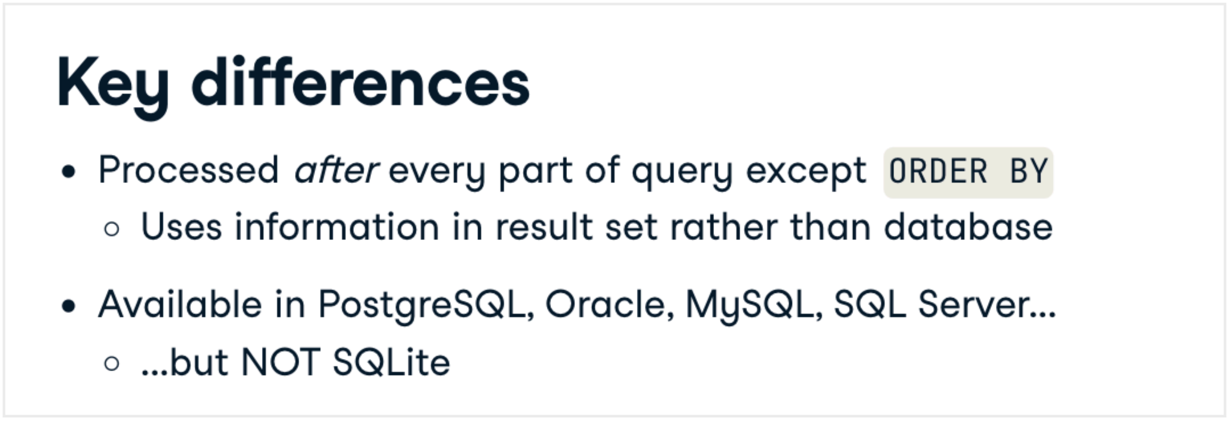
- *What is the rank of matches based on number of goals scored?*

```
SELECT
    date,
    (home_goal + away_goal) AS goals,
    RANK() OVER(ORDER BY home_goal + away_goal DESC) AS goals_rank
FROM match
WHERE season = '2011/2012';
```

```
| date       | goals | goals_rank |
|------------|-------|------------|
| 2011-11-06 | 10    | 1          |
| 2011-08-28 | 10    | 1          |
| 2012-05-12 | 9     | 3          |
| 2012-02-12 | 9     | 3          |
```

- Rank is a function that returns a column with the rank of the input provided to it.

In the above query, we are first ordering the total goal(home + away) and then we are passing it over once using the over function, then this we pass into the Rank() function.
Window functions allow you to create a RANK of information according to any variable you want to use to sort your data. When setting this up, you will need to specify what column/calculation you want to use to calculate your rank. This is done by including an ORDER BY clause inside the OVER() clause.



Key differences

- Processed *after* every part of query except `ORDER BY`
  - Uses information in result set rather than database
- Available in PostgreSQL, Oracle, MySQL, SQL Server...
  - ...but NOT SQLite

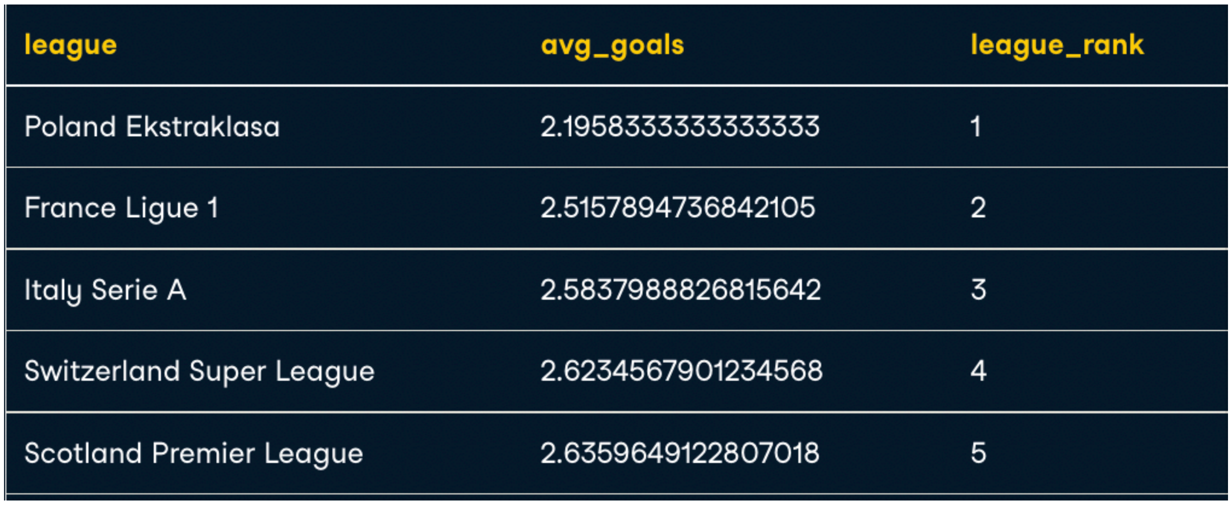
Example - OVER()
Use a window function to include the aggregate average in each row

```
SELECT
   -- Select the id, country name, season, home, and away goals
   m.id,
   c.name AS country,
   m.season,
   m.home_goal,
   m.away_goal,
   -- Use a window to include the aggregate average in each row
   AVG(m.home_goal + m.away_goal) OVER() AS overall_avg
FROM match AS m
LEFT JOIN country AS c ON m.country_id = c.id;
```

————————————————————————————

EXAMPLE - RANK()
Use a window function to Rank each league according to the average goals

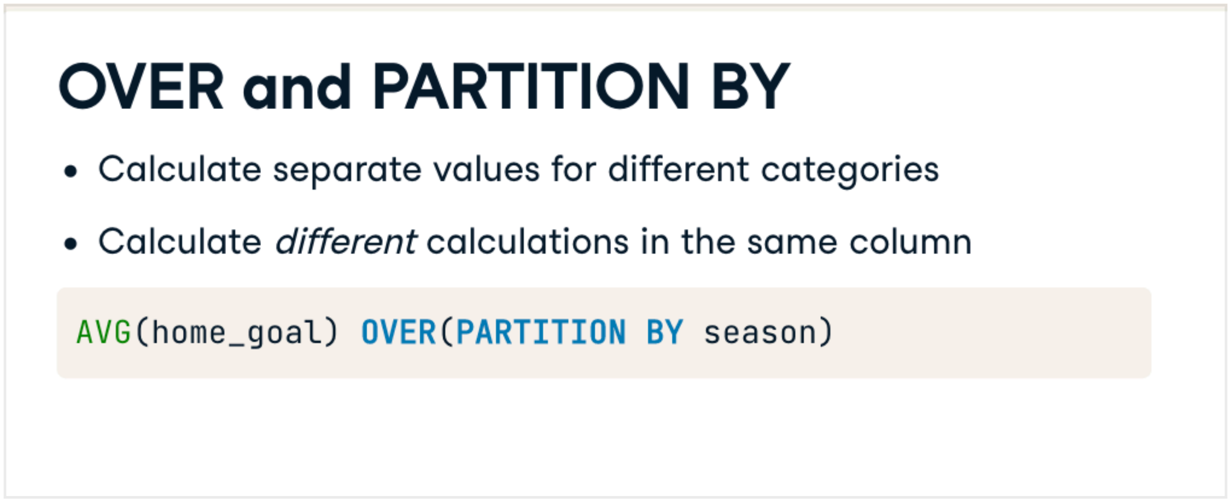
```
SELECT
   -- Select the league name and average goals scored
   l.name AS league,
   AVG(m.home_goal + m.away_goal) AS avg_goals,
   -- Rank each league according to the average goals
   RANK() OVER(ORDER BY AVG(m.home_goal + m.away_goal)) AS league_rank
FROM league AS l
LEFT JOIN match AS m
ON l.id = m.country_id
WHERE m.season = '2011/2012'
GROUP BY l.name
-- Order the query by the rank you created
ORDER BY league_rank;
```



league	avg_goals	league_rank
Poland Ekstraklasa	2.1958333333333333	1
France Ligue 1	2.5157894736842105	2
Italy Serie A	2.5837988826815642	3
Switzerland Super League	2.6234567901234568	4
Scotland Premier League	2.6359649122807018	5

OVER AND PARTITION BY

- The PARTITION BY clause allows you to calculate separate "windows" based on columns you want to divide your results. For example, you can create a single column that calculates an overall average of goals scored for each season



OVER and PARTITION BY

- Calculate separate values for different categories
- Calculate *different* calculations in the same column

```
AVG(home_goal) OVER(PARTITION BY season)
```

Partition your data

- *How many goals were scored in each match, and how did that compare to the overall average?*

```
SELECT
    date,
    (home_goal + away_goal) AS goals,
    AVG(home_goal + away_goal) OVER() AS overall_avg
FROM match;
```

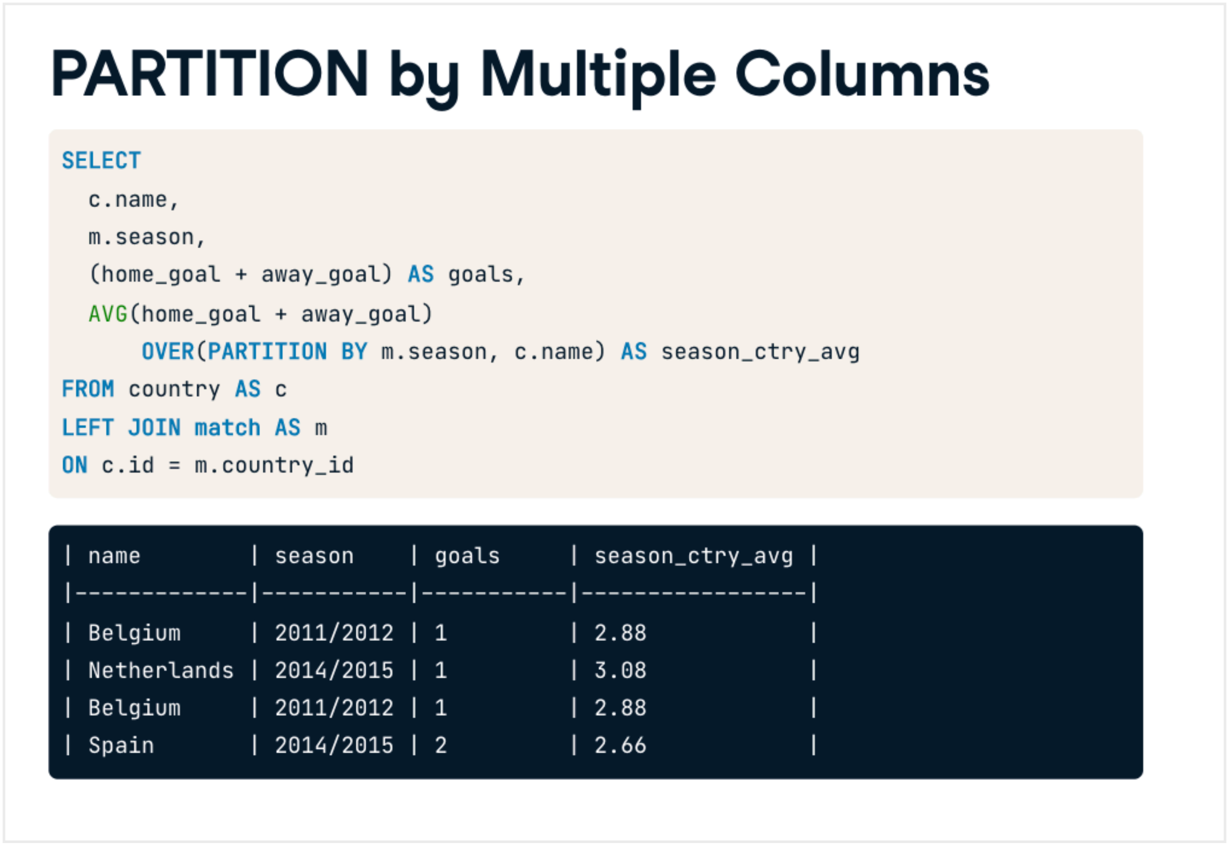
```
| date       | goals | overall_avg |
|------------|-------|-------------|
| 2011-12-17 | 3     | 2.73210     |
| 2012-05-01 | 2     | 2.73210     |
| 2012-11-27 | 4     | 2.73210     |
| 2013-04-20 | 1     | 2.73210     |
| 2013-11-09 | 5     | 2.73210     |
```

Partition your data

- *How many goals were scored in each match, and how did that compare to the season's average?*

```
SELECT
    date,
    (home_goal + away_goal) AS goals,
    AVG(home_goal + away_goal) OVER(PARTITION BY season) AS season_avg
FROM match;
```

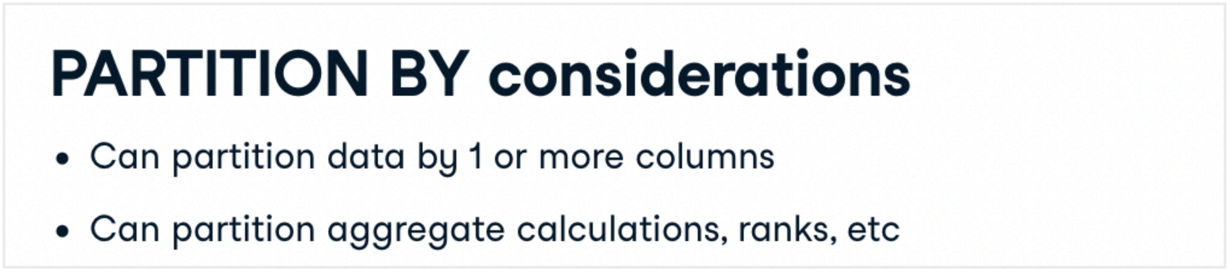
```
| date       | goals | season_avg  |
|------------|-------|-------------|
| 2011-12-17 | 3     | 2.71646     |
| 2012-05-01 | 2     | 2.71646     |
| 2012-11-27 | 4     | 2.77270     |
| 2013-04-20 | 1     | 2.77270     |
| 2013-11-09 | 5     | 2.76682     |
```



PARTITION by Multiple Columns

```
SELECT
  c.name,
  m.season,
  (home_goal + away_goal) AS goals,
  AVG(home_goal + away_goal)
      OVER(PARTITION BY m.season, c.name) AS season_ctry_avg
FROM country AS c
LEFT JOIN match AS m
ON c.id = m.country_id
```

```
| name        | season    | goals     | season_ctry_avg |
|-------------|-----------|-----------|-----------------|
| Belgium     | 2011/2012 | 1         | 2.88            |
| Netherlands | 2014/2015 | 1         | 3.08            |
| Belgium     | 2011/2012 | 1         | 2.88            |
| Spain       | 2014/2015 | 2         | 2.66            |
```



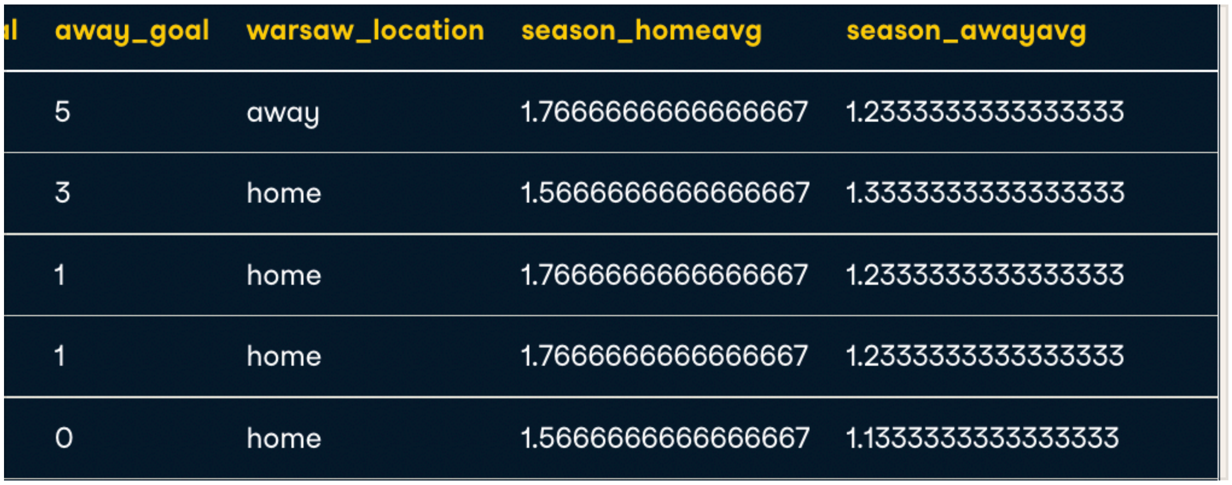
PARTITION BY considerations

- Can partition data by 1 or more columns
- Can partition aggregate calculations, ranks, etc

Example partition by

```
SELECT
    date,
    season,
    home_goal,
    away_goal,
    CASE WHEN hometeam_id = 8673 THEN 'home'
        ELSE 'away' END AS warsaw_location,
    -- Calculate the average goals scored partitioned by season
    AVG(home_goal) OVER(PARTITION BY season) AS season_homeavg,
    AVG(away_goal) OVER(PARTITION BY season) AS season_awayavg
FROM match
-- Filter the data set for Legia Warszawa matches only
WHERE
    hometeam_id = 8673
    OR awayteam_id = 8673
```

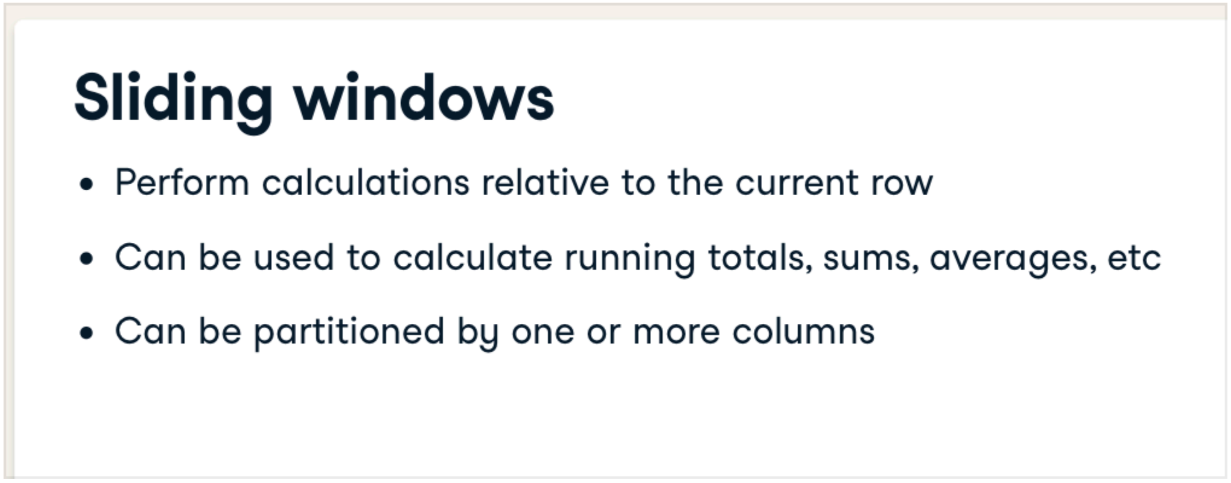
ORDER BY (home_goal + away_goal) DESC;



l	away_goal	warsaw_location	season_homeavg	season_awayavg
	5	away	1.7666666666666667	1.2333333333333333
	3	home	1.5666666666666667	1.3333333333333333
	1	home	1.7666666666666667	1.2333333333333333
	1	home	1.7666666666666667	1.2333333333333333
	0	home	1.5666666666666667	1.1333333333333333

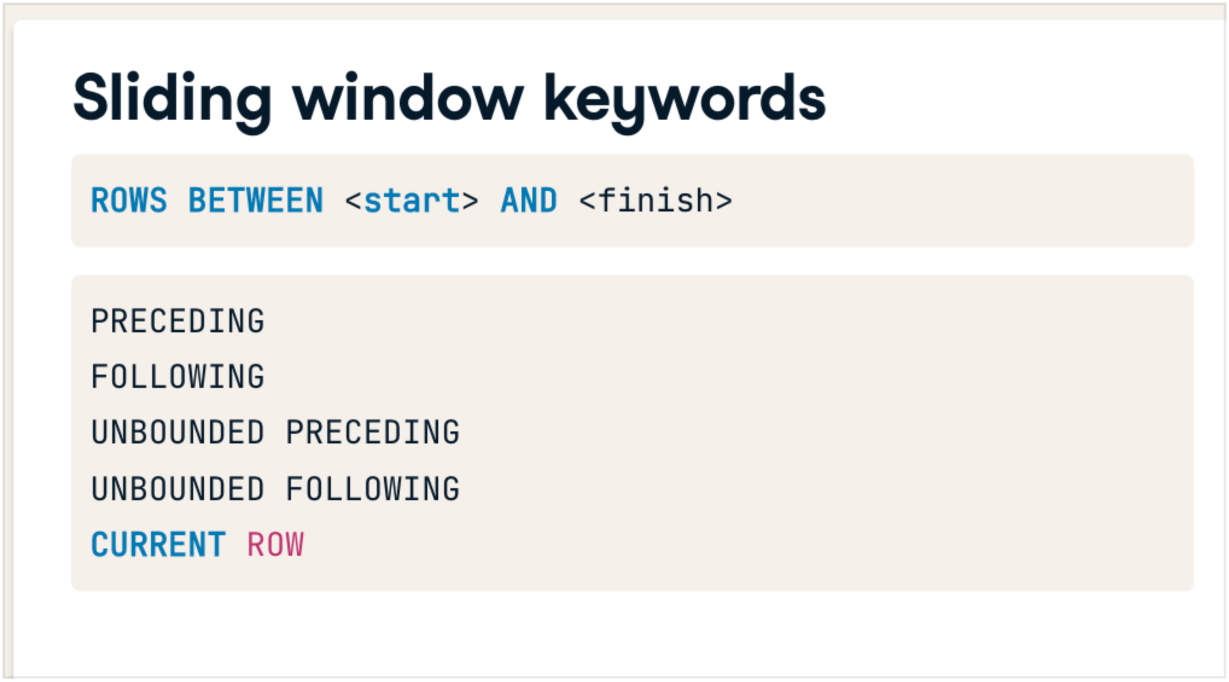
SLIDING WINDOWS

Sliding windows allow you to create running calculations between any two points in a window using functions such as PRECEDING, FOLLOWING, and CURRENT ROW. You can calculate running counts, sums, averages, and other aggregate functions between any two points you specify in the data set.



Sliding windows

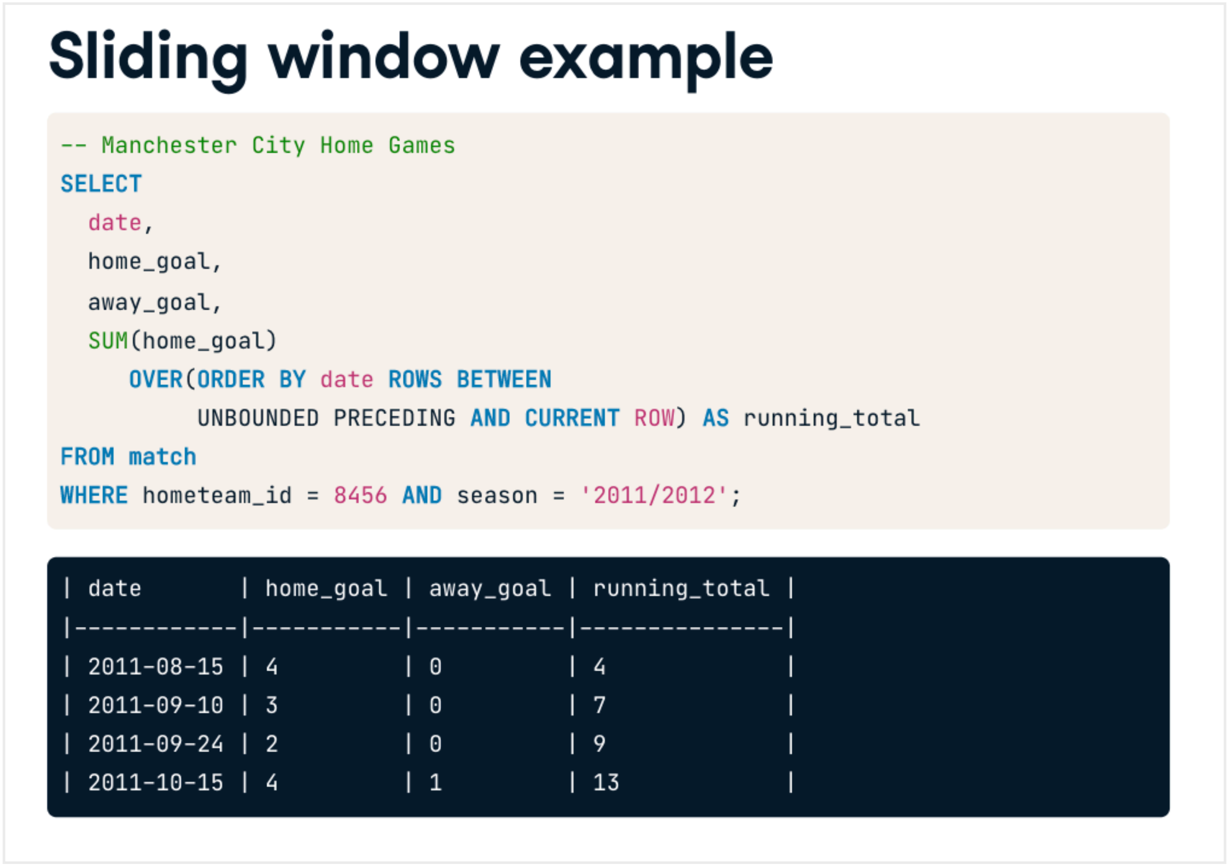
- Perform calculations relative to the current row
- Can be used to calculate running totals, sums, averages, etc
- Can be partitioned by one or more columns


Sliding window keywords

```
ROWS BETWEEN <start> AND <finish>
```

```
PRECEDING
FOLLOWING
UNBOUNDED PRECEDING
UNBOUNDED FOLLOWING
CURRENT ROW
```

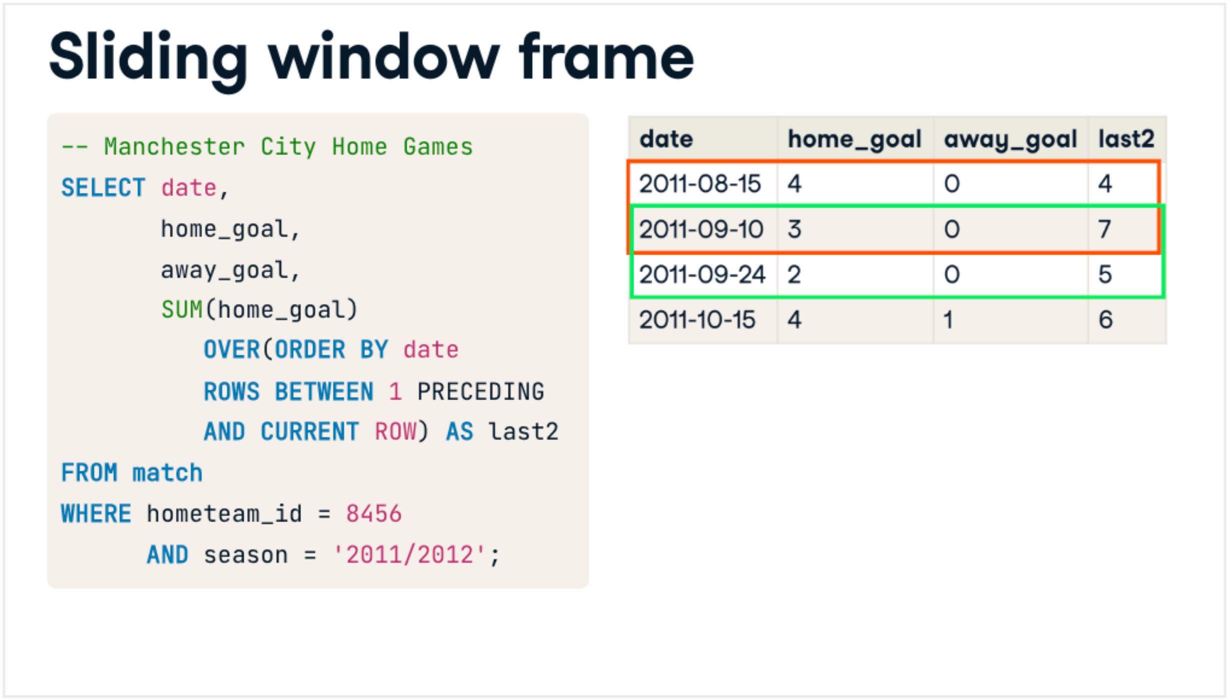
PRECEDING and Following - Are used to specify the number of rows to be included before or after the current row
UNBOUNDED PRECEDING/FOLLOWING - We want to include all the rows prior/ post to the current row


Sliding window example

```sql
-- Manchester City Home Games
SELECT
  date,
  home_goal,
  away_goal,
  SUM(home_goal)
     OVER(ORDER BY date ROWS BETWEEN
          UNBOUNDED PRECEDING AND CURRENT ROW) AS running_total
FROM match
WHERE hometeam_id = 8456 AND season = '2011/2012';
```

```
| date       | home_goal | away_goal | running_total |
|------------|-----------|-----------|---------------|
| 2011-08-15 | 4         | 0         | 4             |
| 2011-09-10 | 3         | 0         | 7             |
| 2011-09-24 | 2         | 0         | 9             |
| 2011-10-15 | 4         | 1         | 13            |
```

In the above query, we are calculating the running total goals of the team.



Sliding window frame

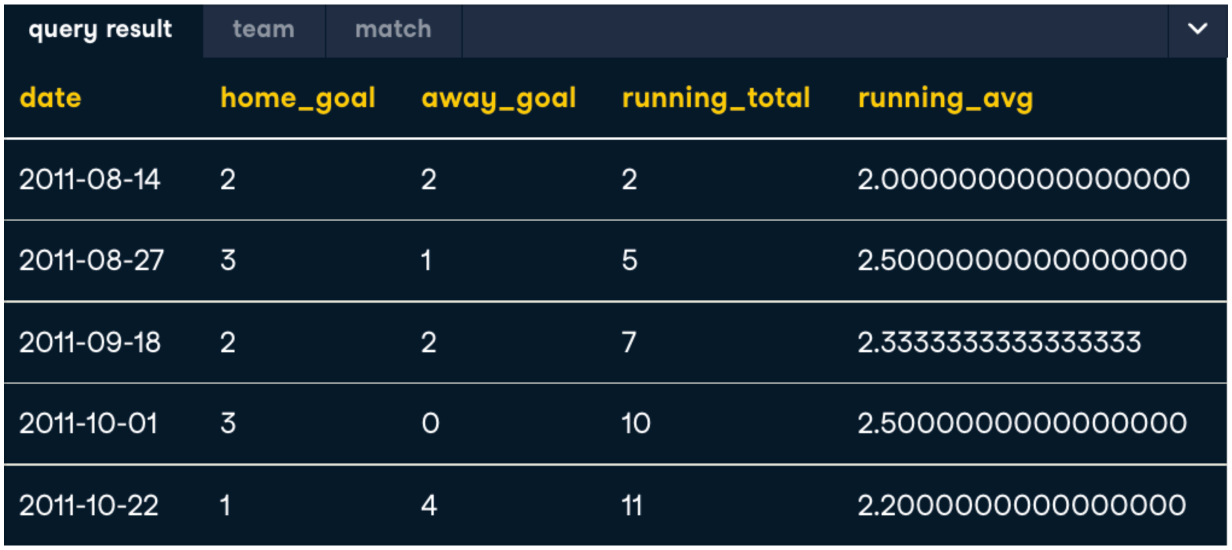
```
-- Manchester City Home Games
SELECT date,
       home_goal,
       away_goal,
       SUM(home_goal)
          OVER(ORDER BY date
          ROWS BETWEEN 1 PRECEDING
          AND CURRENT ROW) AS last2
FROM match
WHERE hometeam_id = 8456
      AND season = '2011/2012';
```

date	home_goal	away_goal	last2
2011-08-15	4	0	4
2011-09-10	3	0	7
2011-09-24	2	0	5
2011-10-15	4	1	6

- We have only used the goal count of 1 previous match (using date we have found the previous match)

Example - Sliding window - ROWS BETWEEN

```
SELECT
    date,
    home_goal,
    away_goal,
   -- Create a running total and running average of home goals
   SUM(home_goal) OVER(ORDER BY date
       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
running_total,
   AVG(home_goal) OVER(ORDER BY date
       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
running_avg
FROM match
WHERE
    hometeam_id = 9908
    AND season = '2011/2012';
```



date	home_goal	away_goal	running_total	running_avg
2011-08-14	2	2	2	2.0000000000000000
2011-08-27	3	1	5	2.5000000000000000
2011-09-18	2	2	7	2.3333333333333333
2011-10-01	3	0	10	2.5000000000000000
2011-10-22	1	4	11	2.2000000000000000