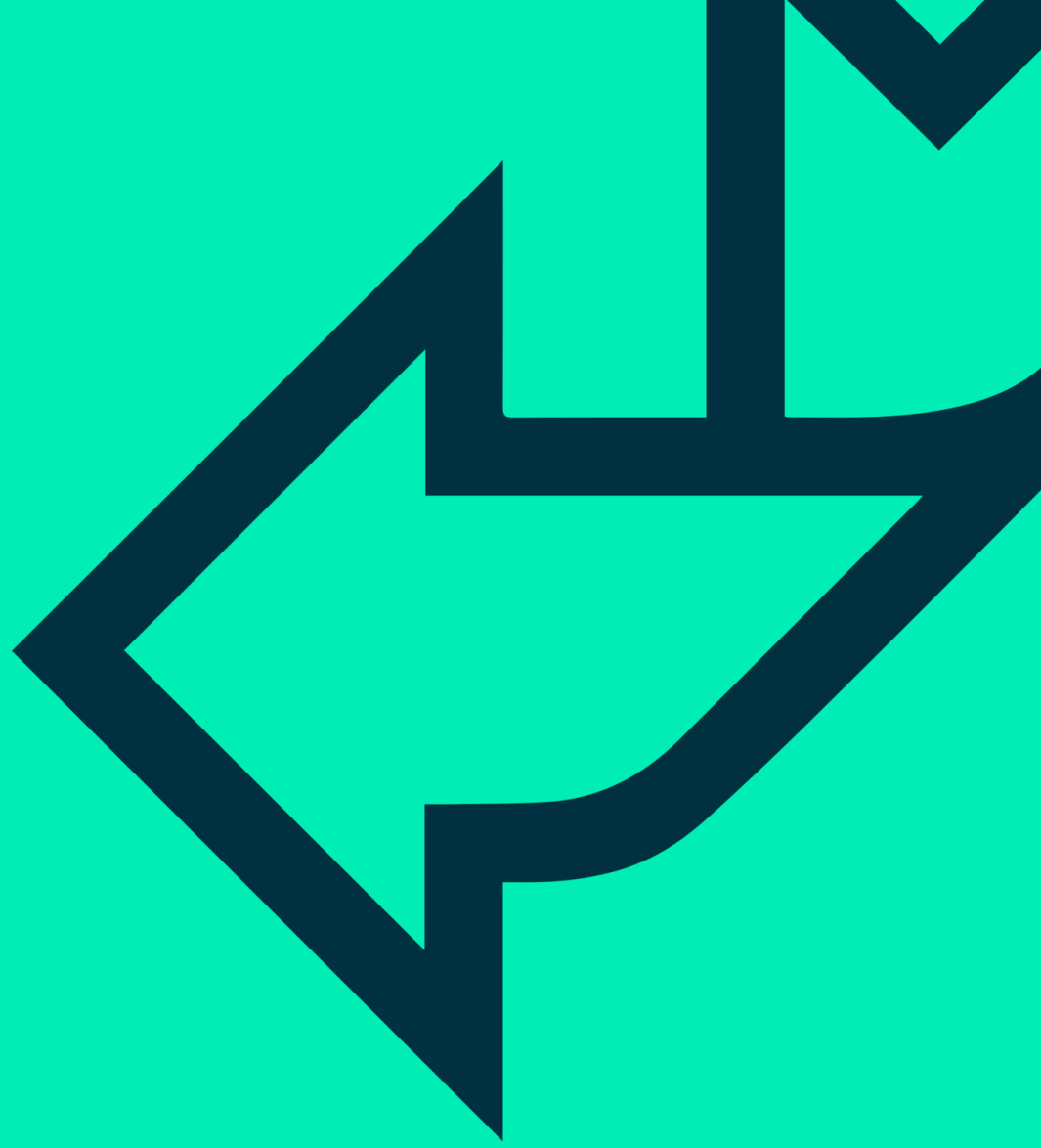
POWERING POTENTIAL

SQL: Window Functions

We will cover

- **SQL window functions**
- **Clauses**
  - → Over
  - → Partitioning By
  - → Rows between
- **Aggregates**
- **Ranking**
- **Positional**
- **Distribution**



WINDOW FUNCTIONS

A **window function** performs a calculation across a set of table rows that are related to the current row.

- This is comparable to the type of calculation that can be done with an aggregate function and the GROUP BY clause.
- 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.

WHEN GROUP BY IS NOT ENOUGH

Let's create a database and import flat file DeptEmployees.csv. We obtain the following table.

Results | Messages

	EmployeeID	EmployeeName	Department	Salary
1	1001	John Smith	Sales	2000
2	1002	Mary Higgins	Sales	1800
3	1003	Peter Cook	Sales	2500
4	1004	Barbara Jenkins	Sales	1500
5	1005	Stephen Newton	IT	1700
6	1006	George Edwards	IT	2200
7	1007	Lilian Humphries	Sales	1500
8	1008	Martin Elliot	Accounts	1400
9	1009	Elizabeth Jones	Accounts	1000
10	1010	Robert Watson	IT	2000

Let's assume we are asked to produce a list containing each employee, their department, their salary, and the average salary for their department.

WHEN GROUP BY IS NOT ENOUGH

TASK: produce a list containing each employee, their department, their salary, and the average salary for their department.

Attempt 1:

```
SELECT EmployeeID, EmployeeName, Department,
    Salary, AVG(Salary) as AvDepartment
FROM DeptEmployees
GROUP BY Department
```

This will not work (can you explain why?)

Attempt 2:

```
SELECT Department, AVG(Salary) as AvDepartment
FROM DeptEmployees
GROUP BY Department
```

Results | Messages

	Department	AvDepartment
1	Accounts	1200
2	IT	1966
3	Sales	1860

This works but the detail is missing – we get only one row per department.

WHEN GROUP BY IS NOT ENOUGH

TASK: produce a list containing each employee, their department, their salary, and the average salary for their department.

Attempt 3:

```
SELECT EmployeeID, EmployeeName, Department, Salary, AVG(Salary)
    OVER() as AvCompany
FROM DeptEmployees
```

Results | Messages

	EmployeeID	EmployeeName	Department	Salary	AvCompany
1	1001	John Smith	Sales	2000	1760
2	1002	Mary Higgins	Sales	1800	1760
3	1003	Peter Cook	Sales	2500	1760
4	1004	Barbara Jenkins	Sales	1500	1760
5	1005	Stephen Newton	IT	1700	1760
6	1006	George Edwards	IT	2200	1760
7	1007	Lilian Humphries	Sales	1500	1760
8	1008	Martin Elliot	Accounts	1400	1760
9	1009	Elizabeth Jones	Accounts	1000	1760
10	1010	Robert Watson	IT	2000	1760

This is more like it!

WINDOW FUNCTION SYNTAX

```
SELECT EmployeeID, EmployeeName, Department, Salary, AVG(Salary)
    OVER() as AvCompany
FROM DeptEmployees
```

The OVER clause designates a window function.
In this case the window function is performed over the entire set of rows – we have the average salary for the whole company.

To narrow the window from the entire data set to individual groups we use PARTITION BY. This will help us finally solve our problem and display the average salary of the department where they work for each employee.

WINDOW FUNCTION

TASK: produce a list containing each employee, their department, their salary, and the average salary for their department.

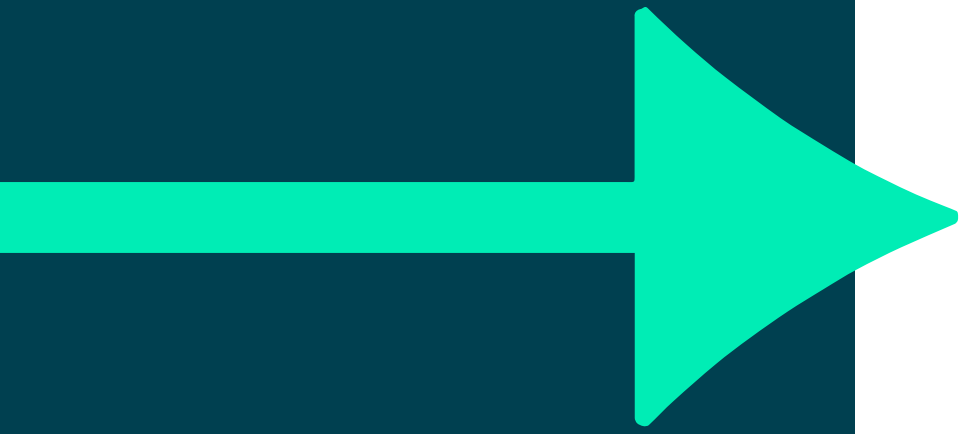
Solution:

```
SELECT EmployeeID, EmployeeName, Department, Salary, AVG(Salary)
    OVER(PARTITION BY Department) as AvDepartment
FROM DeptEmployees
```

Results | Messages

	EmployeeID	EmployeeName	Department	Salary	AvDepartment
1	1008	Martin Elliot	Accounts	1400	1200
2	1009	Elizabeth Jones	Accounts	1000	1200
3	1010	Robert Watson	IT	2000	1966
4	1005	Stephen Newton	IT	1700	1966
5	1006	George Edwards	IT	2200	1966
6	1007	Lilian Humphries	Sales	1500	1860
7	1001	John Smith	Sales	2000	1860
8	1002	Mary Higgins	Sales	1800	1860
9	1003	Peter Cook	Sales	2500	1860
10	1004	Barbara Jenkins	Sales	1500	1860

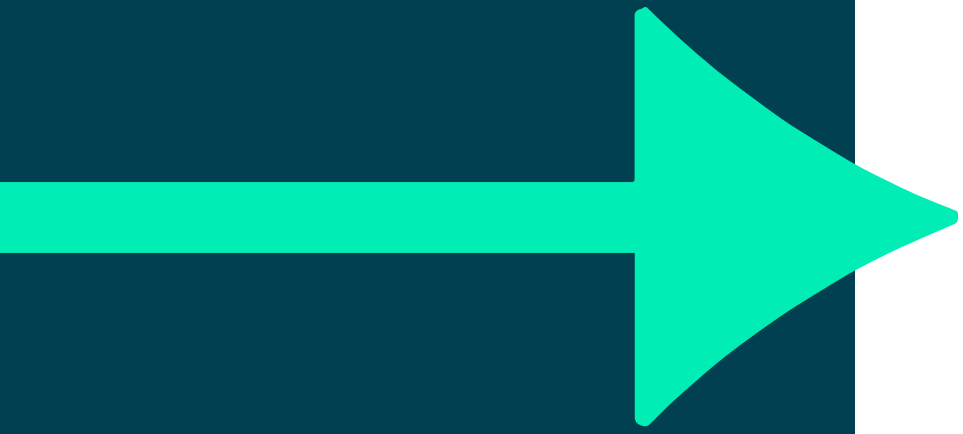
AGGREGATES IN WINDOW FUNCTIONS



Window Functions are applied to Aggregate Functions:

- Sum
- Avg
- Min
- Max
- Count
- Standard deviation and variance (stdev, stdevp, var, varp)

ROWS RELATED TO THE CURRENT ROW



It is often necessary to do aggregation using rows related to the current row, e.g. when calculating moving average.

Create table Revenue using script CreateRevenue.sql.

Example and data source:

https://stevestedman.com/2013/04/rows-and-range-preceding-and-following/

ROWS PRECEDING

ROWS PRECEDING specifies that the aggregate functions in the current partition in the OVER clause will consider the current row, and a specific number of rows before it.

```
-- sum of revenue over a trailing 3 year period
SELECT [Year], DepartmentID, Revenue,
 sum(Revenue) OVER (PARTITION by DepartmentID
 ORDER BY [YEAR]
 ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) as CurrentAndPrev3
FROM REVENUE
ORDER BY DepartmentID, [Year]
```

90 %

Results | Messages

	Year	DepartmentID	Revenue	CurrentAndPrev3
1	1998	1	10030	10030
2	1999	1	20000	30030
3	2000	1	40000	70030
4	2001	1	30000	100030
5	2002	1	90000	180000
6	2003	1	10300	170300
7	2004	1	10000	140300
8	2005	1	20000	130300
9	2006	1	40000	80300
10	2007	1	70000	140000
11	2008	1	50000	180000
12	2009	1	20000	180000
13	2010	1	30000	170000
14	2011	1	80000	180000
15	2012	1	10000	140000
16	1998	2	20000	20000

Default: unbounded preceding and current row

ROWS FOLLOWING

ROWS FOLLOWING specifies that the aggregate functions in the current partition in the OVER clause will consider the current row, and a specific number of rows after it.

```
-- ROWS FOLLOWING
SELECT [Year], DepartmentID, Revenue,
 sum(Revenue) OVER (PARTITION by DepartmentID
 ORDER BY [YEAR]
 ROWS BETWEEN CURRENT ROW AND 3 FOLLOWING) as CurrentAndNext3
FROM REVENUE
ORDER BY DepartmentID, [Year]
```

90 %

Results | Messages

	Year	DepartmentID	Revenue	CurrentAndNext3
1	1998	1	10030	100030
2	1999	1	20000	180000
3	2000	1	40000	170300
4	2001	1	30000	140300
5	2002	1	90000	130300
6	2003	1	10300	80300
7	2004	1	10000	140000
8	2005	1	20000	180000
9	2006	1	40000	180000
10	2007	1	70000	170000
11	2008	1	50000	180000
12	2009	1	20000	140000
13	2010	1	30000	120000
14	2011	1	80000	90000
15	2012	1	10000	10000
16	1998	2	20000	150000

Default: current row and unbounded following

ROWS PRECEDING AND FOLLOWING

The following example illustrates smoothing the values using moving average.

```
--ROWS PRECEDING AND FOLLOWING
SELECT [Year], DepartmentID, Revenue,
sum(Revenue) OVER (PARTITION by DepartmentID
ORDER BY [YEAR]
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) as BeforeAndAfter
FROM REVENUE
ORDER BY DepartmentID, [Year]
```

90 %

Results | Messages

	Year	DepartmentID	Revenue	BeforeAndAfter
1	1998	1	10030	30030
2	1999	1	20000	70030
3	2000	1	40000	90000
4	2001	1	30000	160000
5	2002	1	90000	130300
6	2003	1	10300	110300
7	2004	1	10000	40300
8	2005	1	20000	70000
9	2006	1	40000	130000
10	2007	1	70000	160000
11	2008	1	50000	140000
12	2009	1	20000	100000
13	2010	1	30000	130000
14	2011	1	80000	120000
15	2012	1	10000	90000
16	1998	2	20000	80000

RANKING

Producing ranked list of values

Clauses

- Order by must be used.
- Partition by clause allowed.
- Rows between not allowed.

Functions available:

- Rank().
- Dense_Rank().
- Row_Number().
- Ntile(x).

EXAMPLES RANKING

Using the Northwind database, we will try out different ranking window functions.

```sql
-- rank within a category, by unit price
SELECT CategoryID, ProductID, UnitPrice,
    RANK() OVER(PARTITION BY CategoryID ORDER BY CategoryID, UnitPrice) AS NumRank
FROM Products

-- dense rank within a category, by unit price
SELECT CategoryID, ProductID, UnitPrice,
    DENSE_RANK() OVER(PARTITION BY CategoryID ORDER BY CategoryID, UnitPrice) AS NumRank
FROM Products
```

Observe the difference between the two rankings.

```sql
-- row number across the whole table
SELECT ROW_NUMBER() OVER(ORDER BY CategoryID, ProductID) AS NumRow, CategoryID, ProductID, UnitPrice
FROM Products

-- row number within the partitions
SELECT ROW_NUMBER() OVER(PARTITION BY CategoryID ORDER BY CategoryID, ProductID) AS NumRow,
    CategoryID, ProductID, UnitPrice
FROM Products

-- dividing the products within each category in groups
SELECT CategoryID, ProductID, UnitPrice,
    NTILE(3) OVER(PARTITION BY CategoryID ORDER BY CategoryID, UnitPrice) AS Bucket
FROM Products
```

POSITIONAL

Allows calculation of a value from another row in the dataset.

Clauses:

- Order by must be used.
- Partition by clause allowed.
- Rows between clause depends upon function.

Functions available:

- Lead
- Lag
- First_value
- Last_value

EXAMPLES POSITIONAL – LAG

Using the Northwind database, we will try out different positional window functions.

The **LAG** function allows access to a value stored in a different row above the current row. The row above may be adjacent or some number of rows above.

```
LAG(expression [,offset[,default_value]]) OVER(ORDER BY columns)
```

LAG takes three arguments: the name of the column or an expression from which the value is obtained, the number of rows to skip (offset) above, and the default value to be returned if the stored value obtained from the row above is empty. Only the first argument is required.

```
-- LAG - value in row before
SELECT CategoryID, ProductID, ProductName, UnitPrice,
    LAG(ProductID) OVER(PARTITION BY CategoryID ORDER BY UnitPrice DESC) AS PrevProd FROM Products

-- LAG - value 3 rows before
SELECT CategoryID, ProductID, ProductName, UnitPrice,
    LAG(ProductID,3) OVER(PARTITION BY CategoryID ORDER BY UnitPrice DESC) AS PrevProd FROM Products
```

EXAMPLES POSITIONAL – LEAD

Using the Northwind database, we will try out different positional window functions.

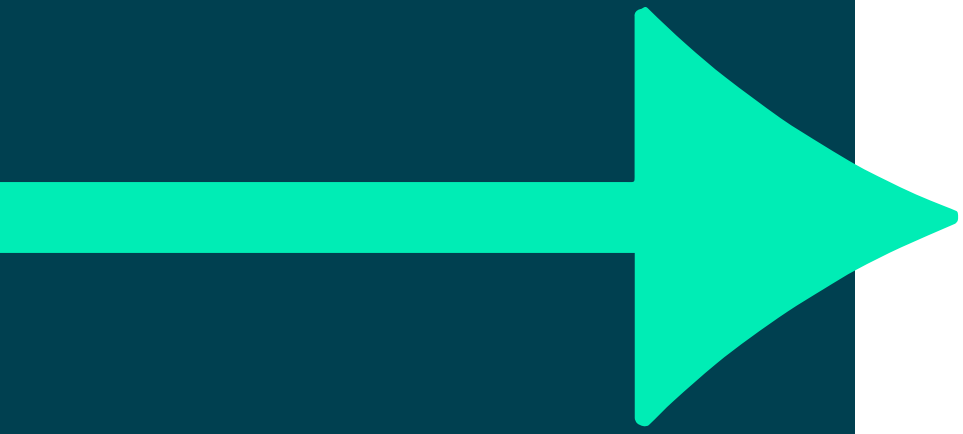
The **LEAD** function acts like LAG. The difference is that it accesses rows below.

```
LEAD(expression [,offset[,default_value]]) OVER(ORDER BY columns)
```

```
-- LEAD - value in row after
SELECT CategoryID, ProductID, ProductName, UnitPrice,
    LEAD(ProductID) OVER(PARTITION BY CategoryID ORDER BY UnitPrice DESC) AS NextProd FROM Products

-- LEAD - value 3 rows after
SELECT CategoryID, ProductID, ProductName, UnitPrice,
    LEAD(ProductID,3) OVER(PARTITION BY CategoryID ORDER BY UnitPrice DESC) AS NextProd FROM Products
```

EXAMPLES POSITIONAL – FIRST_VALUE, LAST_VALUE



Using the Northwind database, we will try out different positional window functions.

FIRST_VALUE and LAST_VALUE

```
-- first value
SELECT CategoryID, ProductID, ProductName, UnitPrice,
    FIRST_VALUE(ProductID) OVER(PARTITION BY CategoryID ORDER BY UnitPrice DESC) AS FirstProd FROM Products

-- last value
SELECT CategoryID, ProductID, ProductName, UnitPrice,
    LAST_VALUE(ProductID) OVER(PARTITION BY CategoryID ORDER BY UnitPrice DESC) AS FirstProd FROM Products
```

WINDOW FUNCTION – DISTRIBUTION

Using the Northwind database, we will try out different positional window functions.

PERCENTILE_CONT and PERCENTILE_DISC

```
-- percentiles
SELECT DISTINCT
    AVG(UnitPrice) OVER() AS Mean,
    PERCENTILE_CONT(0.25) WITHIN GROUP(ORDER BY UnitPrice ASC) OVER() AS Percentile_25,
    PERCENTILE_CONT(0.5)  WITHIN GROUP(ORDER BY UnitPrice ASC) OVER() AS Median,
    PERCENTILE_CONT(0.75) WITHIN GROUP(ORDER BY UnitPrice ASC) OVER() AS Percentile_75,
    PERCENTILE_DISC(0.5)  WITHIN GROUP(ORDER BY UnitPrice ASC) OVER() AS NextLowestToMedian
FROM Products
```

Results | Messages

Mean	Percentile_25	Median	Percentile_75	NextLowestToMedian
28.8663	13.25	19.5	33.25	19.50

WINDOW FUNCTION – DISTRIBUTION

Where are the obtained percentiles on the Box and Whisker plot?

Results | Messages

Mean	Percentile_25	Median	Percentile_75	NextLowestToMedian
28.8663	13.25	19.5	33.25	19.50