Common Table Expressions and Window Functions

Contents

1 Learning Objectives

- Understand the use of common table expressions to simplify complex queries.
- Understand the basic concepts of window functions.
- Have seen some examples of window definitions using `ORDER BY`, `GROUP BY` and their combination.
- Be aware that more complex window definitions are possible.

Duration - 120 minutes

2 Introduction

In this lesson we will look at two more recent additions to the `SQL` standard: **common table expressions (CTEs)** and **window functions**.

We will use the `omni_pool` database, so re-establish your connection to it in `DBeaver` if it has lapsed.

3 Common table expressions (CTEs)

Common table expressions (CTEs) are a useful addition made available in `ANSI SQL` in the `SQL:1999` standard.

Think of them as a 'temporary table' that you define and is then available to be used in a subsequent query. They can help you avoid writing complex queries involving subqueries, particularly where you might end up using the subquery more than once. They can also perform better than queries with more complex structure.

Let's see a simple example where a CTE can make a query more readable. We are going to try to solve the following problem:

"Add a column for each employee showing the ratio of their salary to the average salary of their team."

Our first thought for this might be "we'll need a table of the average salary of each team". Team `name` is stored in the `teams` table, whereas employee `salary` is in the `employees` table, so it seems we will need a join. Let's write the query:

```
SELECT
    t.id,
    t.name,
    AVG(e.salary) AS avg_salary
FROM employees AS e INNER JOIN teams AS t
ON e.team_id = t.id
GROUP BY t.id
```

id	name	avg_salary
8	Data Team 2	59545.49
7	Data Team 1	62983.22
10	Corporate	60988.89
9	Data Escalate	59944.80
1	Audit Team 1	57203.59
5	Audit Escalate	59210.88
4	Risk Team 2	56816.13
2	Audit Team 2	63909.17
6	Risk Escalate	56514.34
3	Risk Team 1	63054.65

Note:
10 of 10 rows

Now we will need to join the `employees` table to the one that we just made. Based on what we've covered up until now, it looks like we will need to use the query above as a subquery:

```
SELECT
    e.first_name,
    e.last_name,
    team_avgs.name AS team_name,
    e.salary,
    e.salary / team_avgs.avg_salary AS salary_over_team_avg
FROM employees AS e INNER JOIN
```

```
(
    SELECT
        t.id,
        t.name,
        AVG(e.salary) AS avg_salary
    FROM employees AS e INNER JOIN teams AS t
    ON e.team_id = t.id
    GROUP BY t.id
) AS team_avgs
ON e.team_id = team_avgs.id
ORDER BY e.team_id
```

first_name	last_name	team_name	salary	salary_over_team_avg
Giovanni	Maddrell	Audit Team 1	34932	0.6106610
Jilli	Ashbee	Audit Team 1	42848	0.7490439
	Dodshun	Audit Team 1	20898	0.3653267
Melina	Simpole	Audit Team 1	83585	1.4611846
Maryjane	Reynalds	Audit Team 1	63299	1.1065565
Symon	Bolletti	Audit Team 1	28861	0.5045313
Uriah	Balhatchet	Audit Team 1	62266	1.0884982
Cyndie	Jeaycock	Audit Team 1	87674	1.5326661
Efrem	Manifould	Audit Team 1	53829	0.9410074
Elsa	Smetoun	Audit Team 1	38077	0.6656401
Goddard	Maseres	Audit Team 1	21979	0.3842242
Daisey	Lunny	Audit Team 1	74019	1.2939573
Rudie	Carbert	Audit Team 1		
Sigismondo	Skipworth	Audit Team 1		
Bentlee	Toy	Audit Team 1	40355	0.7054628
Jolee	Hamer	Audit Team 1	62412	1.0910505
Leigha	Megroff	Audit Team 1	88729	1.5511090
Domeniga	Ravenscroft	Audit Team 1	64603	1.1293523
Jermayne	Learmouth	Audit Team 1	69538	1.2156231
	Devonside	Audit Team 1	53596	0.9369343

Note:
20 of 1000 rows

This is quite difficult to read and understand. Let's try writing this instead using a CTE:

```
WITH team_avgs(id, name, avg_salary) AS (
    SELECT
        t.id,
        t.name,
        AVG(e.salary)
    FROM employees AS e INNER JOIN teams AS t
    ON e.team_id = t.id
    GROUP BY t.id
)
SELECT
    e.first_name,
    e.last_name,
```

```
    team_avgs.name AS team_name,
    e.salary,
    e.salary / team_avgs.avg_salary AS salary_over_team_avg
FROM employees AS e INNER JOIN team_avgs
ON e.team_id = team_avgs.id
ORDER BY e.team_id
```

first_name	last_name	team_name	salary	salary_over_team_avg
Giovanni	Maddrell	Audit Team 1	34932	0.6106610
Jilli	Ashbee	Audit Team 1	42848	0.7490439
	Dodshun	Audit Team 1	20898	0.3653267
Melina	Simpole	Audit Team 1	83585	1.4611846
Maryjane	Reynalds	Audit Team 1	63299	1.1065565
Symon	Bolletti	Audit Team 1	28861	0.5045313
Uriah	Balhatchet	Audit Team 1	62266	1.0884982
Cyndie	Jeaycock	Audit Team 1	87674	1.5326661
Efrem	Manifould	Audit Team 1	53829	0.9410074
Elsa	Smetoun	Audit Team 1	38077	0.6656401
Goddard	Maseres	Audit Team 1	21979	0.3842242
Daisey	Lunny	Audit Team 1	74019	1.2939573
Rudie	Carbert	Audit Team 1		
Sigismondo	Skipworth	Audit Team 1		
Bentlee	Toy	Audit Team 1	40355	0.7054628
Jolee	Hamer	Audit Team 1	62412	1.0910505
Leigha	Megroff	Audit Team 1	88729	1.5511090
Domeniga	Ravenscroft	Audit Team 1	64603	1.1293523
Jermayne	Learmouth	Audit Team 1	69538	1.2156231
	Devonside	Audit Team 1	53596	0.9369343

Note:
20 of 1000 rows

So, we name the CTE `team_avgs` after the `WITH` keyword, and then define it as a query enclosed in `AS ( )`. You see that we can also specify the names of the columns returned by the CTE, although this is not required (default columns names will be provided).

Finally, let's `ROUND()` the number of decimal places reported for the ratio:

```
WITH team_avgs(id, name, avg_salary) AS (
    SELECT
        t.id,
        t.name,
        AVG(e.salary)
    FROM employees AS e INNER JOIN teams AS t
    ON e.team_id = t.id
    GROUP BY t.id
)
SELECT
    e.first_name,
    e.last_name,
    team_avgs.name AS team_name,
    e.salary,
```

```
    ROUND(e.salary / team_avgs.avg_salary, 3) AS salary_over_team_avg
FROM employees AS e INNER JOIN team_avgs
ON e.team_id = team_avgs.id
ORDER BY e.team_id
```

first_name	last_name	team_name	salary	salary_over_team_avg
Giovanni	Maddrell	Audit Team 1	34932	0.611
Jilli	Ashbee	Audit Team 1	42848	0.749
	Dodshun	Audit Team 1	20898	0.365
Melina	Simpole	Audit Team 1	83585	1.461
Maryjane	Reynalds	Audit Team 1	63299	1.107
Symon	Bolletti	Audit Team 1	28861	0.505
Uriah	Balhatchet	Audit Team 1	62266	1.088
Cyndie	Jeaycock	Audit Team 1	87674	1.533
Efrem	Manifould	Audit Team 1	53829	0.941
Elsa	Smetoun	Audit Team 1	38077	0.666
Goddard	Maseres	Audit Team 1	21979	0.384
Daisey	Lunny	Audit Team 1	74019	1.294
Rudie	Carbert	Audit Team 1		
Sigismondo	Skipworth	Audit Team 1		
Bentlee	Toy	Audit Team 1	40355	0.705
Jolee	Hamer	Audit Team 1	62412	1.091
Leigha	Megroff	Audit Team 1	88729	1.551
Domeniga	Ravenscroft	Audit Team 1	64603	1.129
Jermayne	Learmouth	Audit Team 1	69538	1.216
	Devonside	Audit Team 1	53596	0.937

Note:
20 of 1000 rows

Now, how could we change this query to instead report a `salary_over_country_avg` for each employee?

Task - 5 mins

- Write a query for the average `salary` of employees in each country (we want just two columns in this query: `country` and `avg_salary`).
- Use that query as a CTE (call it `country_avgs`).
- Use the `country_avgs` CTE to calculate the `salary_over_country_avg` column in the main query.
- [**Extension**] If you have time, can you figure out the syntax to define **multiple CTEs**, so you can write a query providing both `salary_over_team_avg` and `salary_over_country_avg` columns?

Solution

Here is the query for the average `salary` in each country:

```
SELECT
    country,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY country
```

country	avg_salary
Bangladesh	42137.50
Indonesia	60637.89
Mayotte	89297.00
Cameroon	37713.50
Luxembourg	59098.00
Czech Republic	60142.00
Sweden	54956.81
Uganda	81734.00
Dominican Republic	58528.50
Ireland	75419.00
Macedonia	50462.00
American Samoa	89142.00
Laos	45048.33
Uzbekistan	51862.00
Finland	59092.00
Portugal	52526.92
Colombia	65900.10
Albania	57805.00
Ukraine	59413.08
Cuba	50781.67

Note:
20 of 130 rows

Now let's use that as a CTE to calculate `salary_over_country_avg` for each employee

```
WITH country_avgs(country, avg_salary) AS (
  SELECT
      country,
      AVG(salary)
  FROM employees
  GROUP BY country
)
SELECT
    e.first_name,
    e.last_name,
    country_avgs.country AS country,
    e.salary,
    ROUND(e.salary / country_avgs.avg_salary, 3)
      AS salary_over_country_avg
FROM employees AS e INNER JOIN country_avgs
ON e.country = country_avgs.country
ORDER BY e.country
```

first_name	last_name	country	salary	salary_over_country_avg
Bentlee	Toy	Afghanistan	40355	0.638
Olenolin	Wegman	Afghanistan	75435	1.193
Vance	Ratlee	Afghanistan	89578	1.416
Trixi	Pickvance	Afghanistan	82880	1.310
Abeu	Pawden	Afghanistan	58366	0.923
Beale	Raynard	Afghanistan	32849	0.519
Judy	D'Emanuele	Albania	61785	1.069
Ida	Limeburn	Albania	59084	1.022
Carmencita	Janks	Albania	26677	0.461
Francine	Brobak	Albania	94357	1.632
Marika	Loxdale	Albania	47122	0.815
Saundra	Fearnyhough	American Samoa	89142	1.000
Cornie	Palluschek	Angola	65340	1.000
Janaya	Giacopazzi	Antigua and Barbuda	66154	1.000
Lisabeth	Grason	Argentina	81355	1.257
Dagmar	Alf	Argentina	65572	1.013
Ray	Safont	Argentina	24238	0.374
Lindsey	Humpherson	Argentina	90241	1.394
Tam	Tsar	Argentina	37676	0.582
Jerrold	Hardern	Argentina	48722	0.753

Note:
20 of 1000 rows

Extension

```
WITH
team_avgs(id, name, avg_salary) AS (
    SELECT
        t.id,
        t.name,
        AVG(e.salary)
    FROM employees AS e INNER JOIN teams AS t
    ON e.team_id = t.id
    GROUP BY t.id
),
country_avgs(country, avg_salary) AS (
  SELECT
      country,
      AVG(salary)
  FROM employees
  GROUP BY country
)
SELECT
    e.first_name,
    e.last_name,
    country_avgs.country AS country,
```

```
    team_avgs.name AS team_name,
    e.salary,
    ROUND(e.salary / country_avgs.avg_salary, 3)
      AS salary_over_country_avg,
    ROUND(e.salary / team_avgs.avg_salary, 3)
      AS salary_over_team_avg
FROM employees AS e INNER JOIN country_avgs
ON e.country = country_avgs.country
INNER JOIN team_avgs
ON e.team_id = team_avgs.id
ORDER BY e.country, e.team_id
```

first_name	last_name	country	team_name	salary	salary_over_country_avg	salary_over_team_avg
Bentlee	Toy	Afghanistan	Audit Team 1	40355	0.638	0.705
Olenolin	Wegman	Afghanistan	Risk Team 2	75435	1.193	1.328
Abeu	Pawden	Afghanistan	Risk Escalate	58366	0.923	1.033
Vance	Ratlee	Afghanistan	Data Team 1	89578	1.416	1.422
Trixi	Pickvance	Afghanistan	Data Team 2	82880	1.310	1.392
Beale	Raynard	Afghanistan	Data Team 2	32849	0.519	0.552
Carmencita	Janks	Albania	Audit Team 1	26677	0.461	0.466
Ida	Limeburn	Albania	Risk Team 2	59084	1.022	1.040
Judy	D'Emanuele	Albania	Data Team 1	61785	1.069	0.981
Marika	Loxdale	Albania	Data Team 2	47122	0.815	0.791
Francine	Brobak	Albania	Data Escalate	94357	1.632	1.574
Saundra	Fearnyhough	American Samoa	Data Team 2	89142	1.000	1.497
Cornie	Palluschek	Angola	Risk Team 2	65340	1.000	1.150
Janaya	Giacopazzi	Antigua and Barbuda	Audit Escalate	66154	1.000	1.117
Viv	Stanluck	Argentina	Audit Team 2	92153	1.424	1.442
Lucais	Kenshole	Argentina	Risk Team 1	50684	0.783	0.804
Lisabeth	Grason	Argentina	Risk Team 1	81355	1.257	1.290
Kelly	Guirau	Argentina	Risk Team 2	80074	1.237	1.409
Garrik	Strodder	Argentina	Risk Team 2	88600	1.369	1.559
Agathe	Durban	Argentina	Audit Escalate	98929	1.528	1.671

Note:
20 of 1000 rows

4 Window functions

So far you've seen **aggregate** functions applied mainly to columns of **grouped rows**:

```
SELECT
    department,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

department	avg_salary
Marketing	61132.48
Training	59852.68
Research and Development	58450.00
Business Development	61175.28
Sales	62600.53
Product Management	55723.27
Support	59573.90
Legal	56503.95
Accounting	60557.16
Human Resources	60200.30
Services	63063.45
Engineering	61575.59

Note:
12 of 12 rows

and perhaps to **whole tables**:

```
SELECT
  AVG(salary) AS avg_salary
FROM employees
```

avg_salary
59929.57

Note:
1 of 1 rows

When we apply aggregates to groups, you've seen that we lose access to the contents of individual rows. It's as if all we can extract from each row group are:

- values shared by all rows in the group (typically the value of the `GROUP BY` column for those rows)
- the results of aggregate functions run on each group of rows (`AVG()`, `SUM()`, `COUNT()` etc.)

But what if we want to apply aggregate functions (or other functions, as we'll see) to row groups *while retaining access to individual rows?* This is where **window functions** come to the rescue. These were added to `ANSI SQL` with the publication of the `SQL:2003` standard.

4.1 What is a window function?

A window function is defined by the addition of the `OVER` keyword after a function in the `SELECT` part of a query.

```
SELECT
  function_name OVER (window_definition)
FROM ...
```

A `window_definition` in parentheses must immediately follow `OVER`: this defines **a window of rows relative to the current row** over which the function is applied. If we leave the parentheses empty, then we get the default window which corresponds to **the whole table**. Let's see an example:

```
SELECT
  first_name,
  last_name,
  salary,
  SUM(salary) OVER () AS sum_salary
FROM employees;
```

first_name	last_name	salary	sum_salary
Ibbie	Roscrigg	97667	56034147
Sylas	Smallcomb	48556	56034147
Osmund	Kittel	51200	56034147
Feodora	Dumingos	60460	56034147
Peter	de Vaen	32060	56034147
Livy	Coneau	88481	56034147
Misti	Toll	81088	56034147
	Spawforth	61847	56034147
Seymour	Crumbleholme	48322	56034147
Idaline	Pawfoot	56593	56034147
Thorstein	Garr	39926	56034147
Jessa	Orman	30138	56034147
Harlen	Cottu	75705	56034147
Minette	Scamadin	84127	56034147
Augusta	Andresen	42571	56034147
Janie	Bourgaize	76831	56034147
Feliks	Balhatchet	81815	56034147
Robby	Harragin	70830	56034147
Paula	Bloggett		56034147
Fina	Klimt		56034147

Note:
20 of 1000 rows

This idea of the *current row* is important for understanding window functions. Imagine the database system 'building up' the table returned by a query *row-by-row*. The current row is the row being worked on by the system at that point in time.

Task - 2 mins Examine the results of the query above.

- What do you think it is doing?
- Could we obtain the same output with the `SQL` you have learned up until this point? [**Hint**: this is more tricky - think of using a *subquery*].

Solution

The query adds a column `sum_salary` containing the sum of `salary` over the whole table. This happens because the default window of rows for a window function (defined by the `OVER ()` part) is the whole table. We could do the same thing with a subquery as follows:

```
SELECT
  first_name,
  last_name,
  salary,
    (SELECT SUM(salary) FROM employees) AS sum_salary
FROM employees;
```

first_name	last_name	salary	sum_salary
Ibbie	Roscrigg	97667	56034147
Sylas	Smallcomb	48556	56034147
Osmund	Kittel	51200	56034147
Feodora	Dumingos	60460	56034147
Peter	de Vaen	32060	56034147
Livy	Coneau	88481	56034147
Misti	Toll	81088	56034147
	Spawforth	61847	56034147
Seymour	Crumbleholme	48322	56034147
Idaline	Pawfoot	56593	56034147
Thorstein	Garr	39926	56034147
Jessa	Orman	30138	56034147
Harlen	Cottu	75705	56034147
Minette	Scamadin	84127	56034147
Augusta	Andresen	42571	56034147
Janie	Bourgaize	76831	56034147
Feliks	Balhatchet	81815	56034147
Robby	Harragin	70830	56034147
Paula	Bloggett		56034147
Fina	Klimt		56034147

Note:
20 of 1000 rows

This is generally true: window functions let you straightforwardly obtain that might otherwise require a complex subquery!

So far, so what!? Well, we haven't yet really used the capabilities of the `OVER ()` window definition. We'll examine two keywords we can use between the parentheses: `ORDER BY` and `PARTITION BY`.

4.2 ORDER BY

Let's see an example of using `ORDER BY` in the window definition.

Get a table of employees' names, salary and start date ordered by start date, together with a running total of salaries by start date.

```
SELECT
  id,
    first_name,
    last_name,
    salary,
  start_date,
  SUM(salary) OVER (ORDER BY start_date ASC NULLS LAST) AS running_total_salary
FROM employees;
```

id	first_name	last_name	salary	start_date	running_total_salary
859	Corny	Yearn	99798	1990-01-25	99798
38	Bettye	Moxted	38648	1990-02-12	138446
975	Leonie	Haggleton	87628	1990-02-25	226074
979	Tam	Tsar	37676	1990-03-17	263750
677	Jaquelyn	Viggers	69490	1990-03-18	333240
367	Noelani	Gass	32916	1990-03-19	366156
201	Pascal	Wickling		1990-03-20	366156
84	Brena	MacPhail	99119	1990-03-24	465275
5	Feodora	Dumingos	60460	1990-03-28	525735
273	Gwynne	Robertot		1990-04-06	525735
821	Jeana	McEnery	22403	1990-04-12	548138
253	Zoe	Hews	47529	1990-06-05	595667
37	Franky	Idell		1990-06-06	595667
370	Efrem	Manifould	53829	1990-06-07	714864
459	Torrence	Garrick	65368	1990-06-07	714864
230	Kayle	O'Fogarty	40551	1990-07-22	755415
697	Kris	Lepope	54795	1990-07-31	810210
901	Garland	Belfit	36575	1990-08-04	846785
29	Odessa	Defew		1990-08-07	846785
675	Stephie	Kuna	72001	1990-08-26	918786

Note:
20 of 1000 rows

`ORDER BY` makes the window for the **current row** run from the start of the table up to that row (including any rows with the **same value** of ordering variable as the current row).

Task - 2 mins Look at the table produced by the query above.

- What is the window for the 'Jaquelyn Viggers' row?

- Have a look at the 'Efrem Manifould' and 'Torrence Garrick' rows. Why do these rows have the **same value** of `running_total_salary`? What are the windows for these two rows? [**Hint** - think about the phrase above: 'including any rows with the same value of ordering variable as the current row']

Solution

	first_name	last_name	salary	start_date	running_total_salary
1	Corny	Yearn	99,798	1990-01-25	99,798
2	Bettye	Moxted	38,648	1990-02-12	138,446
3	Leonie	Haggleton	87,628	1990-02-25	226,074
4	Tam	Tsar	37,676	1990-03-17	263,750
5	Jaquelyn	Viggers	69,490	1990-03-18	Jaquelyn Viggers 333,240
6	Noelani	Gass	32,916	1990-03-19	366,156
7	Pascal	Wickling	[NULL]	1990-03-20	366,156
8	Brena	MacPhail	99,119	1990-03-24	465,275
9	Feodora	Dumingos	60,460	1990-03-28	525,735
10	Gwynne	Robertot	[NULL]	1990-04-06	525,735
11	Jeana	McEnery	22,403	1990-04-12	548,138
12	Zoe	Hews	47,529	1990-06-05	595,667
13	Franky	Idell	[NULL]	1990-06-06	595,667
14	Efrem	Manifould	53,829	1990-06-07	Efrem Manifould 714,864
15	Torrence	Garrick	65,368	1990-06-07	Torrence Garrick 714,864
16	Kayle	O'Fogarty	40,551	1990-07-22	755,415

Figure 1: Windows for the 'Jaquelyn Viggers', 'Efrem Manifould' and 'Torrence Garrick' rows

The 'Efrem Manifould' and 'Torrence Garrick' rows have the same `running_total_salary` as these employees had the same `start_date` (the ordering variable). So, both rows have the same window, and hence the same `running_total_salary`.

4.2.1 `RANK()`, `DENSE_RANK()` and `ROW_NUMBER()`

Ranking is one of the main reasons `SQL` developers discover and then learn to use window functions. There are three 'ranking' window functions it is useful to know about: `RANK()`, `DENSE_RANK()` and `ROW_NUMBER()`. Which you use depends upon precisely what you are trying to achieve, as they each do subtly different things. Let's say we're trying to answer the following question:

Rank employees in order by their start date with the corporation.

Which 'rank' you use depends upon what the intended application of the ranking values. Let's see what `RANK()` does:

```
SELECT
  id,
    first_name,
    last_name,
    start_date,
  RANK() OVER (ORDER BY start_date ASC NULLS LAST) AS start_rank
FROM employees;
```

id	first_name	last_name	start_date	start_rank
859	Corny	Yearn	1990-01-25	1
38	Bettye	Moxted	1990-02-12	2
975	Leonie	Haggleton	1990-02-25	3
979	Tam	Tsar	1990-03-17	4
677	Jaquelyn	Viggers	1990-03-18	5
367	Noelani	Gass	1990-03-19	6
201	Pascal	Wickling	1990-03-20	7
84	Brena	MacPhail	1990-03-24	8
5	Feodora	Dumingos	1990-03-28	9
273	Gwynne	Robertot	1990-04-06	10
821	Jeana	McEnery	1990-04-12	11
253	Zoe	Hews	1990-06-05	12
37	Franky	Idell	1990-06-06	13
370	Efrem	Manifould	1990-06-07	14
459	Torrence	Garrick	1990-06-07	14
230	Kayle	O'Fogarty	1990-07-22	16
697	Kris	Lepope	1990-07-31	17
901	Garland	Belfit	1990-08-04	18
29	Odessa	Defew	1990-08-07	19
675	Stephie	Kuna	1990-08-26	20

Note:
20 of 1000 rows

Task - 5 mins Add two extra columns to the query above, defined very similarly, but using the `DENSE_RANK()` and `ROW_NUMBER()` functions. Compare all three 'ranking' columns. How do they differ?

Solution

```
SELECT
  id,
    first_name,
    last_name,
    start_date,
  RANK() OVER (ORDER BY start_date ASC NULLS LAST) AS start_rank,
  DENSE_RANK() OVER (ORDER BY start_date ASC NULLS LAST) AS start_dense_rank,
  ROW_NUMBER() OVER (ORDER BY start_date ASC NULLS LAST) AS start_row_num
FROM employees;
```

id	first_name	last_name	start_date	start_rank	start_dense_rank	start_row_num
859	Corny	Yearn	1990-01-25	1	1	1
38	Bettye	Moxted	1990-02-12	2	2	2
975	Leonie	Haggleton	1990-02-25	3	3	3
979	Tam	Tsar	1990-03-17	4	4	4
677	Jaquelyn	Viggers	1990-03-18	5	5	5
367	Noelani	Gass	1990-03-19	6	6	6
201	Pascal	Wickling	1990-03-20	7	7	7
84	Brena	MacPhail	1990-03-24	8	8	8
5	Feodora	Dumingos	1990-03-28	9	9	9
273	Gwynne	Robertot	1990-04-06	10	10	10
821	Jeana	McEnery	1990-04-12	11	11	11
253	Zoe	Hews	1990-06-05	12	12	12
37	Franky	Idell	1990-06-06	13	13	13
370	Efrem	Manifould	1990-06-07	14	14	14
459	Torrence	Garrick	1990-06-07	14	14	15
230	Kayle	O'Fogarty	1990-07-22	16	15	16
697	Kris	Lepope	1990-07-31	17	16	17
901	Garland	Belfit	1990-08-04	18	17	18
29	Odessa	Defew	1990-08-07	19	18	19
675	Stephie	Kuna	1990-08-26	20	19	20

Note:
20 of 1000 rows

- `RANK()` labels each value with it's ranked position in the column, but the labels can have gaps. So, for example, 'Efrem Manifould' and 'Torrence Garrick' both joined OmniCorp on the 7th June 1990 and are joint `RANK()` 14, and 'Kayle O'Fogarty' is next at `RANK()` 16, being the 16th employee to join. So 15 is missing.
- `DENSE_RANK()` doesn't allow for gaps in the rank labels, so 'Efrem Manifould' and 'Torrence Garrick' are both `DENSE_RANK()` 14, as before, but now 'Kayle O'Fogarty' is `DENSE_RANK()` 15.
- `ROW_NUMBER()` just ranks rows by the order they occur in the table. So, while 'Efrem Manifould' gets `ROW_NUMBER()` 14 and 'Torrence Garrick', 15, it might just as easily have been the other way around. For true ranking purposes, it makes sense to choose one of `RANK()` or `DENSE_RANK()`, as these have no ambiguities in what they output.

4.2.2 `NTILE()`

Let's see an example of another function to solve the following problem:

Split the salaries of employees into four groups corresponding to the 'quartiles' of salary (so, for example, group 1 will contain the lowest 25%), group 2 the next lowest 25%, and so on).

The `NTILE()` window function is designed for this task! Those of you from a more 'statistical' background might recognise this as the 'nth-quantile' function. We pass the number of groups into the function, so for our case we want `NTILE(4)`. We tell the window function which column to order rows on, the direction of ordering etc. We'll also order the rows by `id` to make the effect of `NTILE()` more clear:

```sql
SELECT
  id,
    first_name,
    last_name,
    salary,
    -- let's put NULLs in the first group
  NTILE(4) OVER (ORDER BY salary ASC NULLS FIRST) AS salary_group
FROM employees
ORDER BY id;
```

id	first_name	last_name	salary	salary_group
1	Ibbie	Roscrigg	97667	4
2	Sylas	Smallcomb	48556	2
3	Saleem	Adame	91749	4
4	Osmund	Kittel	51200	2
5	Feodora	Dumingos	60460	3
6	Peter	de Vaen	32060	1
7	Livy	Coneau	88481	4
8	Misti	Toll	81088	4
9		Spawforth	61847	3
10	Seymour	Crumbleholme	48322	2
11	Idaline	Pawfoot	56593	2
12	Thorstein	Garr	39926	2
13	Jessa	Orman	30138	1
14	Harlen	Cottu	75705	3
15	Minette	Scamadin	84127	4
16	Augusta	Andresen	42571	2
17	Janie	Bourgaize	76831	3
18	Feliks	Balhatchet	81815	4
19	Robby	Harragin	70830	3
20	Paula	Bloggett		1

Note:
20 of 1000 rows

You can see that `ORDER BY id` has overriden the `ORDER BY salary` passed in as the window definition. This works because, if you remember back to the recap yesterday, `SELECT` comes before `ORDER BY` in the order of execution of `SQL` queries:

Order of execution	Keyword
...	...
5	`SELECT` (& window functions)
6	`ORDER BY`
...	...

Task - 2 mins If the `NTILE()` function works as expected, we should expect each `salary_group` to contain 250 rows. Let's check this!

Have a look at the following query, which uses a CTE to group the salaries first. Can you see how to complete it to count the number of rows in each `salary_group`?

```
WITH grouped_salaries(salary_group) AS (
  SELECT
    NTILE(4) OVER (ORDER BY salary ASC NULLS FIRST)
  FROM employees
)
SELECT
  ...,
  COUNT(*) AS num_in_group
FROM ...
...
```

Solution

```
WITH grouped_salaries(salary_group) AS (
  SELECT
    NTILE(4) OVER (ORDER BY salary ASC NULLS FIRST)
  FROM employees
)
SELECT
  salary_group,
  COUNT(*) AS num_in_group
FROM grouped_salaries
GROUP BY salary_group
```

salary_group	num_in_group
3	250
4	250
2	250
1	250

Note:
4 of 4 rows

4.3 PARTITION BY

The `PARTITION BY` keyword defines the window for the current row as **the current row plus any other rows having the same value(s) in the column(s) specified after `PARTITION BY`**. The easiest way to understand this is probably with a few examples:

"Show for each employee the number of other employees who are members of the same department as them."

```
SELECT
  id,
  first_name,
  last_name,
  department,
  COUNT(*) OVER (PARTITION BY department) - 1 AS num_other_employees_in_department
FROM employees
ORDER BY id;
```

id	first_name	last_name	department	num_other_employees_in_department
1	Ibbie	Roscrigg	Legal	101
2	Sylas	Smallcomb	Training	80
3	Saleem	Adame	Services	72
4	Osmund	Kittel	Legal	101
5	Feodora	Dumingos	Engineering	86
6	Peter	de Vaen	Product Management	78
7	Livy	Coneau	Marketing	83
8	Misti	Toll	Support	80
9		Spawforth	Product Management	78
10	Seymour	Crumbleholme	Training	80
11	Idaline	Pawfoot	Marketing	83
12	Thorstein	Garr	Legal	101
13	Jessa	Orman	Marketing	83
14	Harlen	Cottu	Engineering	86
15	Minette	Scamadin	Accounting	71
16	Augusta	Andresen	Sales	79
17	Janie	Bourgaize	Services	72
18	Feliks	Balhatchet	Business Development	76
19	Robby	Harragin	Legal	101
20	Paula	Bloggett	Research and Development	93

Note:
20 of 1000 rows

"Show for each employee the number of employees who started in the same month as them."

This is a little bit trickier. Extracting just the month from the employees' start dates isn't specific enough, we also need to extract year too, and `PARTITION BY` the distinct pairings of month and year:

```
SELECT
  id,
  first_name,
  last_name,
  start_date,
  TO_CHAR(start_date, 'Month') || ' ' || TO_CHAR(start_date, 'yyyy') AS month,
  COUNT(*) OVER (
    PARTITION BY EXTRACT(MONTH FROM start_date), EXTRACT(YEAR FROM start_date)
  ) AS num_that_month
FROM employees
ORDER BY id;
```

id	first_name	last_name	start_date	month	num_that_month
1	Ibbie	Roscrigg	2014-12-25	December 2014	2
2	Sylas	Smallcomb	1991-08-01	August 1991	6
3	Saleem	Adame	1999-10-09	October 1999	3
4	Osmund	Kittel	2007-09-06	September 2007	4
5	Feodora	Dumingos	1990-03-28	March 1990	6
6	Peter	de Vaen	1992-06-30	June 1992	5
7	Livy	Coneau	2018-11-11	November 2018	2
8	Misti	Toll	2008-07-06	July 2008	1
9		Spawforth	2013-01-18	January 2013	4
10	Seymour	Crumbleholme	2015-09-05	September 2015	4
11	Idaline	Pawfoot	2017-09-29	September 2017	4
12	Thorstein	Garr	2012-03-23	March 2012	4
13	Jessa	Orman	2008-10-22	October 2008	4
14	Harlen	Cottu	2012-12-18	December 2012	3
15	Minette	Scamadin	1990-10-08	October 1990	4
16	Augusta	Andresen	2018-12-07	December 2018	2
17	Janie	Bourgaize	2005-09-07	September 2005	2
18	Feliks	Balhatchet	2008-03-01	March 2008	6
19	Robby	Harragin	1998-07-20	July 1998	2
20	Paula	Bloggett	2009-08-06	August 2009	4

Note:
20 of 1000 rows

Task - 5 mins Use a window function to answer the following problem:

"Get a table of employee id, first and last name, grade and salary, together with two new columns showing the maximum salary for employees of their grade, and the minimum salary for employees of their grade."

Solution

```
SELECT
  id,
  first_name,
  last_name,
  grade,
  salary,
  MAX(salary) OVER (PARTITION BY grade) AS max_salary_that_grade,
  MIN(salary) OVER (PARTITION BY grade) AS min_salary_that_grade
FROM employees
ORDER BY id;
```

id	first_name	last_name	grade	salary	max_salary_that_grade	min_salary_that_grade
1	Ibbie	Roscrigg	0	97667	99889	20063
2	Sylas	Smallcomb	0	48556	99889	20063
3	Saleem	Adame	0	91749	99889	20063
4	Osmund	Kittel	0	51200	99889	20063
5	Feodora	Dumingos	0	60460	99889	20063
6	Peter	de Vaen	1	32060	99551	20532
7	Livy	Coneau	0	88481	99889	20063
8	Misti	Toll	0	81088	99889	20063
9		Spawforth	0	61847	99889	20063
10	Seymour	Crumbleholme	0	48322	99889	20063
11	Idaline	Pawfoot	0	56593	99889	20063
12	Thorstein	Garr	0	39926	99889	20063
13	Jessa	Orman	0	30138	99889	20063
14	Harlen	Cottu	0	75705	99889	20063
15	Minette	Scamadin	1	84127	99551	20532
16	Augusta	Andresen	0	42571	99889	20063
17	Janie	Bourgaize	1	76831	99551	20532
18	Feliks	Balhatchet	0	81815	99889	20063
19	Robby	Harragin	0	70830	99889	20063
20	Paula	Bloggett	0		99889	20063

Note:
20 of 1000 rows

Let's go back and see a more elegant solution to a problem we faced earlier (which we solved using a CTE):

"Add a column for each employee showing the ratio of their salary to the average salary of their team."

We now see that this can be solved with a window function using definition `OVER (PARTITION BY team_id)`:

```
SELECT
    e.first_name,
    e.last_name,
    t.name AS team_name,
    e.salary,
    ROUND(e.salary / AVG(e.salary) OVER (PARTITION BY e.team_id), 3)
      AS salary_ratio_team_average
FROM employees AS e INNER JOIN teams AS t
ON e.team_id = t.id
ORDER BY e.team_id
```

first_name	last_name	team_name	salary	salary_ratio_team_average
Giovanni	Maddrell	Audit Team 1	34932	0.611
Jilli	Ashbee	Audit Team 1	42848	0.749
	Dodshun	Audit Team 1	20898	0.365
Melina	Simpole	Audit Team 1	83585	1.461
Maryjane	Reynalds	Audit Team 1	63299	1.107
Symon	Bolletti	Audit Team 1	28861	0.505
Uriah	Balhatchet	Audit Team 1	62266	1.088
Cyndie	Jeaycock	Audit Team 1	87674	1.533
Efrem	Manifould	Audit Team 1	53829	0.941
Elsa	Smetoun	Audit Team 1	38077	0.666
Goddard	Maseres	Audit Team 1	21979	0.384
Daisey	Lunny	Audit Team 1	74019	1.294
Rudie	Carbert	Audit Team 1		
Sigismondo	Skipworth	Audit Team 1		
Bentlee	Toy	Audit Team 1	40355	0.705
Jolee	Hamer	Audit Team 1	62412	1.091
Leigha	Megroff	Audit Team 1	88729	1.551
Domeniga	Ravenscroft	Audit Team 1	64603	1.129
Jermayne	Learmouth	Audit Team 1	69538	1.216
	Devonside	Audit Team 1	53596	0.937

Note:
20 of 1000 rows

So here `PARTITION BY e. team_id` says 'make the window for the current row all rows with the **same** `team_id`'. We then calculate the `AVG()` salary over that window, and use this as divisor for the `salary` of the current row to get the required ratio.

Task - 2 mins As earlier, define another window function to add a column for each employee showing their salary as a ratio with their country's average.

Solution

```
SELECT
    e.first_name,
    e.last_name,
    t.name AS team_name,
    e.salary,
    ROUND(e.salary / AVG(e.salary) OVER (PARTITION BY e.team_id), 3)
      AS salary_ratio_team_average,
    ROUND(e.salary / AVG(e.salary) OVER (PARTITION BY e.country), 3)
      AS salary_ratio_country_average
FROM employees AS e INNER JOIN teams AS t
ON e.team_id = t.id
ORDER BY e.team_id
```

first_name	last_name	team_name	salary	salary_ratio_team_average	salary_ratio_country_average
Cole	Priter	Audit Team 1	61666	1.078	1.011
Taddeusz	Tonnesen	Audit Team 1	50961	0.891	0.836
Dale	Borthe	Audit Team 1	50005	0.874	0.764
Sutherland	Peachey	Audit Team 1	81381	1.423	1.243
Mitchell	Prydie	Audit Team 1	62576	1.094	1.026
Brier	Justham	Audit Team 1	86437	1.511	1.417
Jermayne	Learmouth	Audit Team 1	69538	1.216	1.140
Brucie	Ceschini	Audit Team 1	99634	1.742	1.381
Lida	Castagnier	Audit Team 1	79103	1.383	1.297
Verney	MacDermot	Audit Team 1	45640	0.798	0.748
Jessalin	Gobbet	Audit Team 1	99551	1.740	1.387
Nell	Harbidge	Audit Team 1	78350	1.370	1.206
Thorstein	Garr	Audit Team 1	39926	0.698	0.655
Claiborne	de Broke	Audit Team 1	50522	0.883	0.828
Melina	Simpole	Audit Team 1	83585	1.461	1.370
Carmon	Caldayrou	Audit Team 1	25867	0.452	0.516
Kayle	O'Fogarty	Audit Team 1	40551	0.709	0.809
Odessa	Defew	Audit Team 1			
Ellynn	Simonyi	Audit Team 1	78828	1.378	0.970
Symon	Bolletti	Audit Team 1	28861	0.505	0.508

Note:
20 of 1000 rows

4.4 `ORDER BY` and `PARTITION BY` together

Let's see an example of using an `ORDER BY` and `PARTITION BY` window definition in a query together!

"Get a table of employees showing the order in which they started work with the corporation split by department"

```
SELECT
    first_name,
    last_name,
    start_date,
    department,
    RANK() OVER (PARTITION BY department ORDER BY start_date ASC NULLS LAST)
      AS order_started_in_department
FROM employees
ORDER BY start_date
```

first_name	last_name	start_date	department	order_started_in_department
Corny	Yearn	1990-01-25	Engineering	1
Bettye	Moxted	1990-02-12	Engineering	2
Leonie	Haggleton	1990-02-25	Sales	1
Tam	Tsar	1990-03-17	Engineering	3
Jaquelyn	Viggers	1990-03-18	Support	1
Noelani	Gass	1990-03-19	Research and Development	1
Pascal	Wickling	1990-03-20	Engineering	4
Brena	MacPhail	1990-03-24	Marketing	1
Feodora	Dumingos	1990-03-28	Engineering	5
Gwynne	Robertot	1990-04-06	Business Development	1
Jeana	McEnery	1990-04-12	Sales	2
Zoe	Hews	1990-06-05	Marketing	2
Franky	Idell	1990-06-06	Legal	1
Torrence	Garrick	1990-06-07	Sales	3
Efrem	Manifould	1990-06-07	Legal	2
Kayle	O'Fogarty	1990-07-22	Support	2
Kris	Lepope	1990-07-31	Marketing	3
Garland	Belfit	1990-08-04	Human Resources	1
Odessa	Defew	1990-08-07	Research and Development	2
Stephie	Kuna	1990-08-26	Services	1
Rolfe	Camus	1990-08-30	Product Management	1
Claudio	Evemy	1990-09-06	Research and Development	3
Trude	Chastan	1990-09-12	Accounting	1
	Sprake	1990-09-19	Accounting	2
Minette	Scamadin	1990-10-08	Accounting	3
Jeno	Ikins	1990-10-10	Legal	3
	Klimsch	1990-10-14	Business Development	2
Cornelia	Cholerton	1990-10-25	Business Development	3
Orren	Orys	1990-11-17	Support	3
Sydel	Frowde	1990-11-20	Marketing	4
Emmalee	Elam	1990-11-22	Services	2
Wynn	Tellenbrook	1990-11-24	Product Management	2
Constantine	Mosley	1990-12-06	Human Resources	2
Dickie	Gosden	1990-12-17	Support	4
Felicle	McGeneay	1990-12-18	Legal	4
Troy	Vautre	1991-01-17	Product Management	3
Collie	Muckley	1991-02-07	Support	5
	Deathridge	1991-02-18	Sales	4
Valida	Fernely	1991-02-28	Product Management	4
Honoria	Dolphin	1991-03-01	Business Development	4

Note:
40 of 1000 rows

4.5 Extra options in window definition

There are more options in defining windows than we have had time to cover here! As just one example, let's see how we might code a **moving average** using a window function. This is a statistics concept: a moving average is just an average over a pre-defined number of rows that 'moves' through the data. So, for example,

a ten-row moving average of a given column would be, for the **current row**, an average over the column value on the current row, and the same column values on the previous nine rows.

To code this in `SQL`, we can specify a window by number of preceding rows! Let's do this for the `salary` column:

```
SELECT
    start_date,
    salary,
    AVG(salary) OVER (ORDER BY start_date ASC ROWS 9 PRECEDING)
      AS moving_avg_salary
FROM employees
```

start_date	salary	moving_avg_salary
1990-01-25	99798	99798.00
1990-02-12	38648	69223.00
1990-02-25	87628	75358.00
1990-03-17	37676	65937.50
1990-03-18	69490	66648.00
1990-03-19	32916	61026.00
1990-03-20		61026.00
1990-03-24	99119	66467.86
1990-03-28	60460	65716.88
1990-04-06		65716.88
1990-04-12	22403	56042.50
1990-06-05	47529	57152.62
1990-06-06		52799.00
1990-06-07	53829	55106.57
1990-06-07	65368	54517.71
1990-07-22	40551	55608.43
1990-07-31	54795	55506.75
1990-08-04	36575	47688.75
1990-08-07		45864.29
1990-08-26	72001	49131.38

Note:
20 of 1000 rows

This gives us our ten-row moving average (including the current row). Keywords `ROWS <num> PRECEDING` is one of the more advanced window definition options. In reality, you probably would not wish to calculate a moving average in this way as `start_date` is not spaced at uniform intervals (we could fix this by interpolation), but it is fine as a demonstration.

Another minor problem is that we shouldn't start to compute a moving average until we have at least nine preceding rows. We can fix this easily - let's use a CTE to keep the syntax clear:

```
WITH unfiltered(row_number, start_date, salary, moving_avg_salary) AS (
  SELECT
    ROW_NUMBER() OVER(ORDER BY start_date),
      start_date,
      salary,
      AVG(salary) OVER (ORDER BY start_date ASC ROWS 9 PRECEDING)
```

```
    FROM employees
)
SELECT
  start_date,
  salary,
  CASE
    WHEN row_number >= 10 THEN moving_avg_salary
    ELSE NULL
  END AS moving_avg_salary
FROM unfiltered
```

start_date	salary	moving_avg_salary
1990-01-25	99798	
1990-02-12	38648	
1990-02-25	87628	
1990-03-17	37676	
1990-03-18	69490	
1990-03-19	32916	
1990-03-20		
1990-03-24	99119	
1990-03-28	60460	
1990-04-06		65716.88
1990-04-12	22403	56042.50
1990-06-05	47529	57152.62
1990-06-06		52799.00
1990-06-07	53829	55106.57
1990-06-07	65368	54517.71
1990-07-22	40551	55608.43
1990-07-31	54795	55506.75
1990-08-04	36575	47688.75
1990-08-07		45864.29
1990-08-26	72001	49131.38

Note:
20 of 1000 rows

See this post and this post, both by Michal Konarski, for further window definition options.

5 Suggestions for further study

- Setting up databases in `PostgreSQL`.
- Using views and materialised views to write more efficient `SQL` queries.
- Adding indices to columns to speed up slow queries.
- Stored procedures and triggers.