# Lecture 6: Views, CTEs, window functions, indexing

## Contents



**DSCI 513**

Databases & Data Retrieval

## Lecture outline

- Views
- Common Table Expressions (CTEs)
- Window functions
- indexing and performance

Skip to main content

# Views

In the last lecture, we tried to write a query for the following example:

---

**Example:** Which countries speak at least one language that is not spoken an any other country in their continent?

---

This query involves data both from the `country` table as well and the `countrylanguage` table.

On the other hand, we also had to use an `EXISTS` subquery. This was why we first created an intermediate query to first store the results from joining the two tables, and then add the subquery part.

While that solution worked, it might not always be ideal to store the data in another table (why?).

Although the separation of country data from language data makes sense from a logical point of view, it might happen that country + language data are the ones that are accessed the most by our users. Also it might be the case that our users only need names, continents, and population data from the `country` table, and language and percentage from the `countrylanguage` table. In this situation, we can create a **view** of the data that we have in our database.

A view is a **named** result of a `SELECT` query that behaves just like a table; i.e. you can query it just like you query any other table in your database.

Skip to main content

# How to create and delete a view

```
CREATE VIEW view_name AS
    select_statment
;
```

```
DROP VIEW [IF EXISTS] view_name;
```

# Properties of a view

- A view acts as if it is a table, while it's actually only a way of looking at the data differently. This is why they're sometimes called **virtual tables**. As opposed to the temporary table approach, there is no duplication of data.
- A view is always **current and dynamic**. Whenever we access the view, the query that generates it is re-run and we will see an up-to-date version of the data. This may not be true with a temporary table.
- **Views persist.** You might argue that a temporary table only occupies space in the duration of a session, but it's not certain if that's truly an advantage. While it's good that the taken space is released after closing the connection, each time a user needs that particular query, the table also needs to be recreated.
- Views can **hide unnecessary details, or add details that are computed on demand**. Especially in complex databases, not every database user needs to, or wants to have access to all details of the dataset. Can you think of an example?
- With views we can also **manage what different users can see**. For example, in the database of a company not every department needs (or should) have access to all financial information of employees. But they may need their account and SIN numbers to be able to pay them for the job the do for that department. This can be easily managed

Skip to main content

- Views do not support constraints.

## Materialized views: the best of both worlds

Sometimes the query underlying a view might be too slow to compute on the fly in a simple view. In this situation, there is a way to physically store a view. In other words, views can be **materialized**.

A materialized view stores the results of the view query on the physical disk, instead of recomputing the entire query, which is why materialized views are much faster to use.

Here is the syntax for creating and dropping materialized views:

```
CREATE MATERIALIZED VIEW my_mat_view AS
    select_statement
;

DROP MATERIALIZED VIEW my_mat_view;
```

Like almost anything in this world, you won't gain anything unless you lose something else. Same is true with materialized views. The **trade-offs** you need to be aware of when using materialized views:

- Just like a temporary table, a materialized view occupies disk space
- The data in a materialized view needs to be refreshed if the data in the related tables are modified.

Skip to main content

Materialized views can be refreshed using the following syntax:

```
REFRESH MATERIALIZED VIEW [CONCURRENTLY] my_mat_view;
```

**Note:** If the keyword `CONCURRENTLY` is used, the view will be usable during the refresh process, but there's no guarantee that the data will be up to date.

**Takeaway:** Using a materialized view is justified if the query takes **a lot of time but little space**. A simple query, on the other hand, is best suited to a query that takes **a considerable amount of space, but relatively little time**.

---

**Question:** Can you think of two advantages of a temporary table over a simple view?

---

```
%load_ext sql
%config SqlMagic.displaylimit = 30
```

```
import json
import urllib.parse

with open('data/credentials.json') as f:
    login = json.load(f)
```

Skip to main content

```
password = urllib.parse.quote(login['password'])
host = login['host']
port = login['port']
```

```
%sql postgresql://{username}:{password}@{host}/imdb
```

```
'Connected: postgres@imdb'
```

In this example here, I have intentionally written a slow query (involving a correlated subquery) to find movies whose number of votes is higher than the average number of votes of all movies belonging to the same genre. Running this query will take some time:

```
%%sql

SELECT
    *
FROM
    movies m
JOIN
    movie_genres mg
ON
    m.id = mg.movie_id
WHERE
    rating > 8
    AND
    nvotes > (
        SELECT
            AVG(m2.nvotes)
        FROM
            movies m2
        JOIN
            movie_genres mg2
        ON
            m2.id = mg2.movie_id
        WHERE
            mg.genre = mg2.genre
            AND
            m.id <> m2.id
    )
;
```

```
 * postgresql://postgres:***@localhost/imdb
795 rows affected.
```

Skip to main content

| id | title | orig_title | start_year | end_year | runtime | rating | nv |
|---|---|---|---|---|---|---|---|
| 10042876 | Rashomon | Rashômon | 1950 | None | 88 | 8.2 | 13 |
| 10042876 | Rashomon | Rashômon | 1950 | None | 88 | 8.2 | 13 |
| 10042876 | Rashomon | Rashômon | 1950 | None | 88 | 8.2 | 13 |
| 10043014 | Sunset Blvd. | None | 1950 | None | 110 | 8.4 | 18 |
| 10043014 | Sunset Blvd. | None | 1950 | None | 110 | 8.4 | 18 |
| 10044741 | Ikiru | None | 1952 | None | 143 | 8.3 | 6 |
| 10044837 | Limelight | None | 1952 | None | 137 | 8.1 | 1 |
| 10045152 | Singin' in the Rain | None | 1952 | None | 103 | 8.3 | 20 |
| 10045152 | Singin' in the Rain | None | 1952 | None | 103 | 8.3 | 20 |
| 10045152 | Singin' in the Rain | None | 1952 | None | 103 | 8.3 | 20 |
| 10046268 | The Wages of Fear | Le salaire de la peur | 1953 | None | 131 | 8.1 | 4 |
| 10046268 | The Wages of Fear | Le salaire de la peur | 1953 | None | 131 | 8.1 | 4 |
| 10046478 | Ugetsu | Ugetsu monogatari | 1953 | None | 96 | 8.3 | 1 |
| 10046912 | Dial M for Murder | None | 1954 | None | 105 | 8.2 | 14 |
| 10046912 | Dial M for Murder | None | 1954 | None | 105 | 8.2 | 14 |
| 10047296 | On the Waterfront | None | 1954 | None | 108 | 8.1 | 13 |
| 10047296 | On the Waterfront | None | 1954 | None | 108 | 8.1 | 13 |
| 10047296 | On the Waterfront | None | 1954 | None | 108 | 8.1 | 13 |
| 10047396 | Rear Window | None | 1954 | None | 112 | 8.5 | 40 |
| 10047396 | Rear Window | None | 1954 | None | 112 | 8.5 | 40 |

Skip to main content

| id | title | orig_title | start_year | end_year | runtime | rating | nv |
|---|---|---|---|---|---|---|---|
| 10050083 | 12 Angry Men | None | 1957 | None | 96 | 8.9 | 61 |
| 10050613 | Throne of Blood | Kumonosu-jô | 1957 | None | 110 | 8.1 | 4( |
| 10050613 | Throne of Blood | Kumonosu-jô | 1957 | None | 110 | 8.1 | 4( |
| 10050783 | The Nights of Cabiria | Le notti di Cabiria | 1957 | None | 110 | 8.1 | 3! |
| 10050825 | Paths of Glory | None | 1957 | None | 88 | 8.4 | 16( |
| 10050825 | Paths of Glory | None | 1957 | None | 88 | 8.4 | 16( |
| 10050986 | Wild Strawberries | Smultronstället | 1957 | None | 91 | 8.2 | 8! |
| 10050986 | Wild Strawberries | Smultronstället | 1957 | None | 91 | 8.2 | 8! |
| 10051036 | Sweet Smell of Success | None | 1957 | None | 96 | 8.1 | 2. |
| 10051036 | Sweet Smell of Success | None | 1957 | None | 96 | 8.1 | 2. |

*795 rows, truncated to displaylimit of 30*

Let's create a simple view for this query:

```
%%sql

CREATE VIEW
    simple_view
AS
    SELECT
        *
    FROM
        movies m
    JOIN
        movie_genres mg
    ON
        m.id = mg.movie_id
    WHERE
        rating > 8
        AND
```

Skip to main content

```
            AVG(m2.nvotes)
        FROM
            movies m2
        JOIN
            movie_genres mg2
        ON
            m2.id = mg2.movie_id
        WHERE
            mg.genre = mg2.genre
            AND
            m.id <> m2.id
    )
;
```

```
 * postgresql://postgres:***@localhost/imdb
Done.
```

```
[]
```

Retrieving data from this view will be as slow as running the original query:

```
%%timeit -r 1 -n 1
%%sql

SELECT * FROM simple_view;
```

```
 * postgresql://postgres:***@localhost/imdb
795 rows affected.
13.6 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

The result of this query takes little space (only 795 rows), but it takes a long time to compute, which makes it a good candidate for a materialized view. Note that creating the materialized view will be slow the first time:

```
%%timeit -r 1 -n 1
%%sql
```

Skip to main content

```
AS
    SELECT
        *
    FROM
        movies m
    JOIN
        movie_genres mg
    ON
        m.id = mg.movie_id
    WHERE
        rating > 8
        AND
        nvotes > (
            SELECT
                AVG(m2.nvotes)
            FROM
                movies m2
            JOIN
                movie_genres mg2
            ON
                m2.id = mg2.movie_id
            WHERE
                mg.genre = mg2.genre
                AND
                m.id <> m2.id
        )
;
```

```
 * postgresql://postgres:***@localhost/imdb
795 rows affected.
13.4 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

But retrieving data from the materialized view should be almost instantaneous:

```
%%timeit -r 1 -n 1
%%sql

SELECT * FROM mat_view;
```

```
 * postgresql://postgres:***@localhost/imdb
795 rows affected.
3.55 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Let's not forget to clean up:

Skip to main content

```
DROP VIEW simple_view;
DROP MATERIALIZED VIEW mat_view;
```

```
 * postgresql://postgres:***@localhost/imdb
Done.
Done.
```

```
[]
```

# Common Table Expressions (CTEs)

It's not uncommon for our queries to become overly complex. For example, consider the following question:

---

**Example:** Which countries speak at least one language that is not spoken an any other country in their continent?

---

A query to answer the above question involves joining tables first to have both country and language data in the same place, and then use a subquery.

To that end, we can create a temporary table first and then use the subquery on that table:

```
DROP TABLE IF EXISTS ccl;

CREATE TEMPORARY TABLE ccl AS (
    SELECT
        co.name, co.continent, cl.language
    FROM
        country co
    JOIN
        countrylanguage cl
    ON
        co.code = cl.countrycode
)
```

Skip to main content

```
SELECT
    t1.*
FROM
    ccl t1
WHERE
    NOT EXISTS (
        SELECT
            *
        FROM
            ccl t2
        WHERE
            t1.name <> t2.name
            AND
            t1.continent = t2.continent
            AND
            t1.language = t2.language
    )
;
```

For this situation, we may not want to create a **temporary table** because:

- copying the data of two entire tables might create a disk space problem during a session
- if the data is modified in anyways before another subsequent query, it won't be reflected in the temporary table
- the temporary table has to be created each time a connection to the database is made

Creating a **view** looks like a nice alternative because it doesn't pose any of the above problems, but:

- The purpose of a view is to provide a more or less permanent way of accessing the data or hiding details. It doesn't make sense to create a view that is intended just for a very specific query and useless otherwise. For example, do we really want a view in our database that shows countries and languages just in Asia and Europe, or is it just for the purpose of a special query?
- Creating views requires special access privileges, which as a user we might not have!

Skip to main content

Standard SQL has an elegant way of addressing this issue: **Common Table Expressions**

A common table expression is a temporary intermediate result set that we can use within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. The syntax for a CTE is:

```
WITH
    expression_name [(column_names, ...)]
AS (
    query
)
query
;
```

Let's take a look at a few examples:

```
%sql postgresql://{username}:{password}@{host}:{port}/world
```

```
'Connected: postgres@world'
```

The following CTE creates an intermediate result set from the `country` and `countrylanguage` tables:

```
%%sql

WITH my_cte AS (
    SELECT
        name, population, language
    FROM
        country co
    JOIN
        countrylanguage cl
    ON
        cl.countrycode = co.code
```

Skip to main content

```
SELECT * FROM my_cte
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/world
984 rows affected.
```

Skip to main content

| name | population | language |
|---|---|---|
| Afghanistan | 22720000 | Pashto |
| Netherlands | 15864000 | Dutch |
| Netherlands Antilles | 217000 | Papiamento |
| Albania | 3401200 | Albaniana |
| Algeria | 31471000 | Arabic |
| American Samoa | 68000 | Samoan |
| Andorra | 78000 | Spanish |
| Angola | 12878000 | Ovimbundu |
| Anguilla | 8000 | English |
| Antigua and Barbuda | 68000 | Creole English |
| United Arab Emirates | 2441000 | Arabic |
| Argentina | 37032000 | Spanish |
| Armenia | 3520000 | Armenian |
| Aruba | 103000 | Papiamento |
| Australia | 18886000 | English |
| Azerbaijan | 7734000 | Azerbaijani |
| Bahamas | 307000 | Creole English |
| Bahrain | 617000 | Arabic |
| Bangladesh | 129155000 | Bengali |
| Barbados | 270000 | Bajan |
| Belgium | 10239000 | Dutch |
| Belize | 241000 | English |
| Benin | 6097000 | Fon |
| Bermuda | 65000 | English |
| Bhutan | 2124000 | Dzongkha |
| Bolivia | 8329000 | Spanish |
| Bosnia and Herzegovina | 3972000 | Serbo-Croatian |

Skip to main content

| name | population | language |
|---|---|---|
| Botswana | 1622000 | Tswana |
| Brazil | 170115000 | Portuguese |
| United Kingdom | 59623400 | English |

*984 rows, truncated to displaylimit of 30*

Let's say we want to count the number of official languages in each country. We can write this query as:

```
%%sql

WITH my_cte AS (
    SELECT
        co.name, cl.language
    FROM
        country co
    JOIN
        countrylanguage cl
    ON
        cl.countrycode = co.code
    WHERE
        cl.isofficial = TRUE
)
SELECT
    name, COUNT(language)
FROM
    my_cte
GROUP BY
    name
ORDER BY
    COUNT(language) DESC
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/world
190 rows affected.
```

Skip to main content

| name | count |
|---|---|
| Switzerland | 4 |
| South Africa | 4 |
| Singapore | 3 |
| Peru | 3 |
| Bolivia | 3 |
| Vanuatu | 3 |
| Belgium | 3 |
| Luxembourg | 3 |
| Seychelles | 2 |
| Cyprus | 2 |
| Netherlands Antilles | 2 |
| Romania | 2 |
| Kyrgyzstan | 2 |
| Burundi | 2 |
| Afghanistan | 2 |
| Guam | 2 |
| Togo | 2 |
| Israel | 2 |
| Faroe Islands | 2 |
| Nauru | 2 |
| Canada | 2 |
| Palau | 2 |
| Tonga | 2 |
| Madagascar | 2 |
| Greenland | 2 |
| Tuvalu | 2 |
| Samoa | 2 |

Skip to main content

| name | count |
|---|---|
| Malta | 2 |
| Rwanda | 2 |
| Sri Lanka | 2 |

*190 rows, truncated to displaylimit of 30*

We can also rewrite this query from the last lecture using a CTE:

**Question:** Which countries speak at least one language that is not spoken in any other country on their continent?

```
%%sql

WITH ccl AS (
    SELECT
        co.name, co.continent, cl.language
    FROM
        country co
    JOIN
        countrylanguage cl
    ON
        co.code = cl.countrycode
)
SELECT
    t1.*
FROM
    ccl t1
WHERE
    NOT EXISTS (
        SELECT
            *
        FROM
            ccl t2
        WHERE
            t1.name <> t2.name
            AND
            t1.continent = t2.continent
            AND
            t1.language = t2.language
    )
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/world
362 rows affected.
```

Skip to main content

| name | continent | language |
|---|---|---|
| Angola | Africa | Ovimbundu |
| Barbados | North America | Bajan |
| Benin | Africa | Fon |
| Bhutan | Asia | Dzongkha |
| Burundi | Africa | Kirundi |
| Ethiopia | Africa | Oromo |
| Falkland Islands | South America | English |
| Fiji Islands | Oceania | Fijian |
| Philippines | Asia | Pilipino |
| Faroe Islands | Europe | Faroese |
| Georgia | Asia | Georgiana |
| Greenland | North America | Greenlandic |
| Haiti | North America | Haiti Creole |
| Indonesia | Asia | Javanese |
| Iceland | Europe | Icelandic |
| Japan | Asia | Japanese |
| Kenya | Africa | Kikuyu |
| Central African Republic | Africa | Gbaya |
| Kyrgyzstan | Asia | Kirgiz |
| Congo, The Democratic Republic of the | Africa | Luba |
| Cocos (Keeling) Islands | Oceania | Malay |
| Cyprus | Asia | Greek |
| Latvia | Europe | Latvian |
| Lesotho | Africa | Sotho |
| Luxembourg | Europe | Luxembourgish |
| Malawi | Africa | Chichewa |
| Maldives | Asia | Dhivehi |

Skip to main content

| name | continent | language |
|---|---|---|
| Mali | Africa | Bambara |
| Malta | Europe | Maltese |
| Marshall Islands | Oceania | Marshallese |

*362 rows, truncated to displaylimit of 30*

---

**Question:** Can you think of two advantages of a temporary table over CTEs?

---

Besides simplifying long and complex queries and increase readability, we can do things with CTEs that are impossible otherwise. One example is with grouping and aggregation.

Remember that when aggregate functions are used, only columns can be in the `SELECT` clause that are either aggregated (used inside an aggregate function) or appear also in the `GROUP BY` clause. Now, consider the following example:

---

**Example:** List city names, their country and region where they are located. Also in each row (i.e. for each city)

- list the number of official languages and
- number of cities with population over 1,000,000

in their respective countries.

---

Skip to main content

We can obviously join the three tables `country`, `city`, and `countrylanguage` and do grouping, but we won't be able to list city names for the reason just discussed. But we can actually do this using a CTE:

```
%%sql

WITH
    ccl (code, country, region, "num_cities_over_1_mil", "num_off_lang")
AS (
    SELECT
        co.code, co.name, co.region,
        COUNT(DISTINCT ci.name),
        COUNT(DISTINCT cl.language)
    FROM
        country co
    JOIN
        city ci
    ON
        co.code = ci.countrycode
    JOIN
        countrylanguage cl
    ON
        co.code = cl.countrycode
    WHERE
        ci.population > 1000000
    GROUP BY
        co.code, co.name, co.region
)
SELECT
    ci.name, ccl.*
FROM
    ccl
JOIN
    city ci
ON
    ccl.code = ci.countrycode
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/world
3671 rows affected.
```

Skip to main content

| name | code | country | region | num_cities_over_1_mil | num_off_lang |
|---|---|---|---|---|---|
| Kabul | AFG | Afghanistan | Southern and Central Asia | 1 | 5 |
| Qandahar | AFG | Afghanistan | Southern and Central Asia | 1 | 5 |
| Herat | AFG | Afghanistan | Southern and Central Asia | 1 | 5 |
| Mazar-e-Sharif | AFG | Afghanistan | Southern and Central Asia | 1 | 5 |
| Alger | DZA | Algeria | Northern Africa | 1 | 2 |
| Oran | DZA | Algeria | Northern Africa | 1 | 2 |
| Constantine | DZA | Algeria | Northern Africa | 1 | 2 |
| Annaba | DZA | Algeria | Northern Africa | 1 | 2 |
| Batna | DZA | Algeria | Northern Africa | 1 | 2 |
| Sétif | DZA | Algeria | Northern Africa | 1 | 2 |
| Sidi Bel Abbès | DZA | Algeria | Northern Africa | 1 | 2 |
| Skikda | DZA | Algeria | Northern Africa | 1 | 2 |
| Biskra | DZA | Algeria | Northern Africa | 1 | 2 |
| Blida (el-Boulaida) | DZA | Algeria | Northern Africa | 1 | 2 |
| Béjaïa | DZA | Algeria | Northern | 1 | 2 |

Skip to main content

| name | code | country | region | num_cities_over_1_mil | num_off_lang |
|---|---|---|---|---|---|
| Mostaganem | DZA | Algeria | Northern Africa | 1 | 2 |
| Tébessa | DZA | Algeria | Northern Africa | 1 | 2 |
| Tlemcen (Tilimsen) | DZA | Algeria | Northern Africa | 1 | 2 |
| Béchar | DZA | Algeria | Northern Africa | 1 | 2 |
| Tiaret | DZA | Algeria | Northern Africa | 1 | 2 |
| Ech-Chleff (el-Asnam) | DZA | Algeria | Northern Africa | 1 | 2 |
| Ghardaïa | DZA | Algeria | Northern Africa | 1 | 2 |
| Luanda | AGO | Angola | Central Africa | 1 | 9 |
| Huambo | AGO | Angola | Central Africa | 1 | 9 |
| Lobito | AGO | Angola | Central Africa | 1 | 9 |
| Benguela | AGO | Angola | Central Africa | 1 | 9 |
| Namibe | AGO | Angola | Central Africa | 1 | 9 |
| Buenos Aires | ARG | Argentina | South America | 3 | 3 |
| La Matanza | ARG | Argentina | South America | 3 | 3 |
| Córdoba | ARG | Argentina | South America | 3 | 3 |

*3671 rows, truncated to displaylimit of 30*

Skip to main content

# (OPTIONAL) Data-modifying CTEs

CTEs can also be used with `INSERT`, `DELETE`, and `UPDATE` to catch the rows modified by these statements via the `RETURNING` statement and use them for another purpose.

Let's connect to the `mds` database we created earlier and reload it with the original data:

```
%sql postgresql://{username}:{password}@{host}:{port}/mds
```

```
'Connected: postgres@mds'
```

```
%%sql

DROP TABLE IF EXISTS
    instructor,
    instructor_course,
    course_cohort
;

CREATE TABLE instructor (
    id INTEGER PRIMARY KEY,
    name TEXT,
    email TEXT,
    phone VARCHAR(12),
    department VARCHAR(50)
    )
;

INSERT INTO
    instructor (id, name, email, phone, department)
VALUES
    (1, 'Mike', 'mike@mds.ubc.ca', '605-332-2343', 'Computer Science'),
    (2, 'Tiffany', 'tiff@mds.ubc.ca', '445-794-2233', 'Neuroscience'),
    (3, 'Arman', 'arman@mds.ubc.ca', '935-738-5796', 'Physics'),
    (4, 'Varada', 'varada@mds.ubc.ca', '243-924-4446', 'Computer Science'),
    (5, 'Quan', 'quan@mds.ubc.ca', '644-818-0254', 'Economics'),
    (6, 'Joel', 'joel@mds.ubc.ca', '773-432-7669', 'Biomedical Engineering'),
    (7, 'Florencia', 'flor@mds.ubc.ca', '773-926-2837', 'Biology'),
    (8, 'Alexi', 'alexiu@mds.ubc.ca', '421-888-4550', 'Statistics'),
    (15, 'Vincenzo', 'vincenzo@mds.ubc.ca', '776-543-1212', 'Statistics'),
    (19, 'Gittu', 'gittu@mds.ubc.ca', '776-334-1132', 'Biomedical Engineering'
    (16, 'Jessica', 'jessica@mds.ubc.ca', '211-990-1762', 'Computer Science')
;


CREATE TABLE instructor_course (
    id SERIAL PRIMARY KEY
```

Skip to main content

```
    enrollment INTEGER,
    begins DATE
    )
;

INSERT INTO
    instructor_course (instructor_id, course, enrollment, begins)
VALUES
    (8, 'Statistical Inference and Computation I', 125, '2021-10-01'),
    (8, 'Regression II', 102, '2022-02-05'),
    (1, 'Descriptive Statistics and Probability', 79, '2021-09-10'),
    (1, 'Algorithms and Data Structures', 25, '2021-10-01'),
    (3, 'Algorithms and Data Structures', 25, '2021-10-01'),
    (3, 'Python Programming', 133, '2021-09-07'),
    (3, 'Databases & Data Retrieval', 118, '2021-11-16'),
    (6, 'Visualization I', 155, '2021-10-01'),
    (6, 'Privacy, Ethics & Security', 148, '2022-03-01'),
    (2, 'Programming for Data Manipulation', 160, '2021-09-08'),
    (7, 'Data Science Workflows', 98, '2021-09-15'),
    (2, 'Data Science Workflows', 98, '2021-09-15'),
    (12, 'Web & Cloud Computing', 78, '2022-02-10'),
    (10, 'Introduction to Optimization', NULL, '2022-09-01'),
    (9, 'Parallel Computing', NULL, '2023-01-10'),
    (13, 'Natural Language Processing', NULL, '2023-09-10')
;

CREATE TABLE course_cohort (
    id INTEGER,
    cohort VARCHAR(7)
    )
;

INSERT INTO
    course_cohort (id, cohort)
VALUES
    (13, 'MDS-CL'),
    (8, 'MDS-CL'),
    (1, 'MDS-CL'),
    (3, 'MDS-CL'),
    (1, 'MDS-V'),
    (9, 'MDS-V'),
    (9, 'MDS-V'),
    (3, 'MDS-V')
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
Done.
Done.
11 rows affected.
Done.
```

Skip to main content

```
Done.
8 rows affected.
```

```
[]
```

```
%sql SELECT * FROM instructor;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
11 rows affected.
```

| id | name | email | phone | department |
|---|---|---|---|---|
| 1 | Mike | mike@mds.ubc.ca | 605-332-2343 | Computer Science |
| 2 | Tiffany | tiff@mds.ubc.ca | 445-794-2233 | Neuroscience |
| 3 | Arman | arman@mds.ubc.ca | 935-738-5796 | Physics |
| 4 | Varada | varada@mds.ubc.ca | 243-924-4446 | Computer Science |
| 5 | Quan | quan@mds.ubc.ca | 644-818-0254 | Economics |
| 6 | Joel | joel@mds.ubc.ca | 773-432-7669 | Biomedical Engineering |
| 7 | Florencia | flor@mds.ubc.ca | 773-926-2837 | Biology |
| 8 | Alexi | alexiu@mds.ubc.ca | 421-888-4550 | Statistics |
| 15 | Vincenzo | vincenzo@mds.ubc.ca | 776-543-1212 | Statistics |
| 19 | Gittu | gittu@mds.ubc.ca | 776-334-1132 | Biomedical Engineering |
| 16 | Jessica | jessica@mds.ubc.ca | 211-990-1762 | Computer Science |

```
%%sql

DROP TABLE IF EXISTS former_instructor;

CREATE TABLE
    former_instructor AS (
    SELECT
        *
    FROM
        instructor
    WHERE
```

Skip to main content

```
)
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
Done.
0 rows affected.
```

```
[]
```

```
%sql SELECT * FROM former_instructor;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
0 rows affected.
```

| id | name | email | phone | department |
|---|---|---|---|---|

Here I use the `RETURNING` statement to retrieve the results of deleting two rows using a CTE, and insert the deleted rows into another table, namely, `former_instructor`:

```
%%sql

WITH deleted_rows AS (
    DELETE FROM
        instructor
    WHERE
        name IN ('Arman', 'Mike')
    RETURNING
        *
)
INSERT INTO
    former_instructor
SELECT
    *
FROM
    deleted_rows
;
```

```
   postgresql://postgres:***@localhost/imdb
```

Skip to main content

```
   postgresql://postgres:***@localhost:5432/world
2 rows affected.
```

```
[]
```

Let's check out the `former_instructor` table now:

```
%sql SELECT * FROM former_instructor;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
2 rows affected.
```

| id | name | email | phone | department |
|---|---|---|---|---|
| 1 | Mike | mike@mds.ubc.ca | 605-332-2343 | Computer Science |
| 3 | Arman | arman@mds.ubc.ca | 935-738-5796 | Physics |

The two deleted rows have been successfully moved to the `former_instructor` table!

# Window functions

We've been using aggregation and grouping quite a lot so far. But as you may recall, there are situations that aggregation and grouping alone cannot address, such as having aggregated and non-aggregated columns together.

We know that aggregation functions perform calculations on groups of rows, and return a single value for all rows that take part in that calculation. They also collapse the participant rows into one single row with the the common column(s) values between all the collapsed rows and the aggregated value. For example, in the `instructor` table of the `mds` database, we can group rows by `department`, and apply `COUNT()` to find the number of instructors in each department, but we no longer see individual instructor rows. Instead, we will see a

Skip to main content

Window functions act in a similar fashion: they perform operations on groups of rows that are related in some way to the current row (like `GROUP BY` that bases the relatedness on common values in grouped columns), but these functions **do not collapse rows**! This is the most important difference between aggregation and window functions

Let's take a look at an example first. The following query finds the maximum population of countries in each continent:

```
%sql postgresql://{username}:{password}@{host}:{port}/world
```

```
'Connected: postgres@world'
```

```
%%sql

SELECT
    continent,
    MAX(population)
FROM
    country
GROUP BY
    continent
;
```

```
   postgresql://postgres:***@localhost/imdb
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
7 rows affected.
```

| continent | max |
|---|---|
| Asia | 1277558000 |
| South America | 170115000 |
| North America | 278357000 |
| Oceania | 18886000 |
| Antarctica | 0 |
| Africa | 111506000 |
| Europe | 146934000 |

Skip to main content

However, what if we want each country to be listed along with its own population, as well as the maximum population in their respective continent? This is not possible with aggregation functions alone.

In order to list country names, their population, and the maximum population in their continent we can use `MAX()` as a window function:

```
%%sql

SELECT
    name,
    population,
    continent,
    MAX(population)
        OVER (PARTITION BY continent)
FROM
    country
;
```

```
   postgresql://postgres:***@localhost/imdb
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

Skip to main content

| name | population | continent | max |
|---|---|---|---|
| Algeria | 31471000 | Africa | 111506000 |
| Western Sahara | 293000 | Africa | 111506000 |
| Madagascar | 15942000 | Africa | 111506000 |
| Uganda | 21778000 | Africa | 111506000 |
| Malawi | 10925000 | Africa | 111506000 |
| Mali | 11234000 | Africa | 111506000 |
| Morocco | 28351000 | Africa | 111506000 |
| Côte d Ivoire | 14786000 | Africa | 111506000 |
| Mauritania | 2670000 | Africa | 111506000 |
| Mauritius | 1158000 | Africa | 111506000 |
| Mayotte | 149000 | Africa | 111506000 |
| Nigeria | 111506000 | Africa | 111506000 |
| Niger | 10730000 | Africa | 111506000 |
| Mozambique | 19680000 | Africa | 111506000 |
| Namibia | 1726000 | Africa | 111506000 |
| Botswana | 1622000 | Africa | 111506000 |
| Tunisia | 9586000 | Africa | 111506000 |
| Angola | 12878000 | Africa | 111506000 |
| Burkina Faso | 11937000 | Africa | 111506000 |
| Burundi | 6695000 | Africa | 111506000 |
| Chad | 7651000 | Africa | 111506000 |
| Togo | 4629000 | Africa | 111506000 |
| Tanzania | 33517000 | Africa | 111506000 |
| Djibouti | 638000 | Africa | 111506000 |
| Egypt | 68470000 | Africa | 111506000 |
| Swaziland | 1008000 | Africa | 111506000 |
| Eritrea | 3850000 | Africa | 111506000 |

Skip to main content

| name | population | continent | max |
|---|---|---|---|
| South Africa | 40377000 | Africa | 111506000 |
| Ethiopia | 62565000 | Africa | 111506000 |
| Sudan | 29490000 | Africa | 111506000 |

*239 rows, truncated to displaylimit of 30*

# Syntax

A lot of new keywords appeared all at once here, so let me explain them all in detail. The syntax of a window function is:

```
SELECT
    column,
    .
    .
    .
    window_function_name(expression) OVER (
        PARTITION BY column
        ORDER BY column
        frame_clause
    .
    .
    .
FROM
    table
;
```

- `window_function_name()` specifies which window function we want to use. I'll list the different types of window functions in a bit.
- `OVER` is what tells the SQL engine that we are going to use a window function
- `PARTITION BY` determines how or based on which column we want the window function to subdivide the rows into groups
- `ORDER BY` specifies how we want the rows in each subgroup to be ordered
- `frame_clause` gives us further control over which rows in each subgroup can participate

Skip to main content

> **Note:** Window functions in SQL are processed **after** `GROUP BY` and `HAVING`, and **before** `SELECT`. This is a crucial point to understand when using window functions in SQL.

> **Note:** Window functions are only allowed in the `SELECT` and `ORDER BY` clauses.

# Types of window functions

## Aggregate functions

These are the regular aggregate functions that we've learned about before, i.e. `AVG()`, `COUNT()`, `MAX()`, `MIN()`, and `SUM()`, which can also be used as window functions.

## Ranking functions

- `CUME_DIST()`: returns the cumulative distribution, i.e. the percentage of values less than or equal to the current value.

Skip to main content

- `NTILE()`: given a specified number of buckets, it tells us in which bucket each row goes among other rows in the partition.
- `PERCENT_RANK()`: similar to `CUME_DIST()`, but considers only the percentage of values less than (and not equal to) the current value.
- `DENSE_RANK()`: returns the rank of a row within a partition without jumps after duplicate ranks (e.g. 1, 2, 2, 3, ...)
- `RANK()`: returns the rank of row within a partition but with jumps after duplicates ranks (e.g. 1, 2, 2, 4, ...)
- `ROW_NUMBER()`: returns simply the number of a row in a partition, regardless of duplicate values

## Value functions

- `FIRST_VALUE()`: returns the first value of the ordered partition
- `LAST_VALUE()`: returns the last value of the ordered partition
- `LAG()`: returns value from the preceding row with a specified offset
- `LEAD()`: returns value from the following row with a specified offset

### Differences with subqueries

- Subqueries can operate on multiple tables, window functions are limited to the current table
- Subqueries are far more general (e.g. `WHERE` conditions including `EXITS`, etc.)
- Window functions are more readable
- Window functions offer functionalities such as `RANK()` or `LAG()` and `LEAD()` that are either very hard, if not impossible, to implement using subqueries
- Window functions are generally faster than subqueries when used for a same purpose

Skip to main content

**Example:** Using window functions, count the number of countries in which each language is officially spoken. Remove duplicates and sort the results in descending order by the count value.

```
%%sql

SELECT
    DISTINCT language,
    COUNT(*) OVER (PARTITION BY language) AS count
FROM
    countrylanguage
WHERE
    isofficial = TRUE
ORDER BY
    count DESC
```

```
   postgresql://postgres:***@localhost/imdb
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
102 rows affected.
```

Skip to main content

| language | count |
|---|---|
| English | 44 |
| Arabic | 22 |
| Spanish | 20 |
| French | 18 |
| German | 6 |
| Portuguese | 6 |
| Dutch | 4 |
| Italian | 4 |
| Malay | 4 |
| Danish | 3 |
| Russian | 3 |
| Serbo-Croatian | 3 |
| Aimará | 2 |
| Chinese | 2 |
| Greek | 2 |
| Ket ua | 2 |
| Korean | 2 |
| Norwegian | 2 |
| Romanian | 2 |
| Samoan | 2 |
| Swedish | 2 |
| Tamil | 2 |
| Turkish | 2 |
| Afrikaans | 1 |
| Albaniana | 1 |
| Armenian | 1 |
| Azerbaijani | 1 |

Skip to main content

| language | count |
|---|---|
| Belorussian | 1 |
| Bengali | 1 |
| Bislama | 1 |

*102 rows, truncated to displaylimit of 30*

**Question:** In the above query, is it possible to use a `WHERE` clause to limit the results to languages which are spoken in more than one country?

In the following example, we rank countries by life expectancy in each continent, but we also create a column to retrieve the name of the country whose ranking follows the one in the current row:

```
%%sql

SELECT
    name,
    continent,
    DENSE_RANK()
        OVER (PARTITION BY continent ORDER BY lifeexpectancy DESC),
    LEAD(name, 1)
        OVER (PARTITION BY continent ORDER BY lifeexpectancy DESC)
FROM
    country
WHERE
    population > 10000000
```

```
   postgresql://postgres:***@localhost/imdb
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
78 rows affected.
```

Skip to main content

| name | continent | dense_rank | lead |
|---|---|---|---|
| Algeria | Africa | 1 | Morocco |
| Morocco | Africa | 2 | Egypt |
| Egypt | Africa | 3 | Ghana |
| Ghana | Africa | 4 | Sudan |
| Sudan | Africa | 5 | Madagascar |
| Madagascar | Africa | 6 | Cameroon |
| Cameroon | Africa | 7 | Tanzania |
| Tanzania | Africa | 8 | Nigeria |
| Nigeria | Africa | 9 | South Africa |
| South Africa | Africa | 10 | Congo, The Democratic Republic of the |
| Congo, The Democratic Republic of the | Africa | 11 | Kenya |
| Kenya | Africa | 12 | Burkina Faso |
| Burkina Faso | Africa | 13 | Mali |
| Mali | Africa | 13 | Somalia |
| Somalia | Africa | 14 | Ethiopia |
| Ethiopia | Africa | 15 | Côte d Ivoire |
| Côte d Ivoire | Africa | 15 | Uganda |
| Uganda | Africa | 16 | Niger |
| Niger | Africa | 17 | Angola |
| Angola | Africa | 18 | Zimbabwe |
| Zimbabwe | Africa | 19 | Malawi |
| Malawi | Africa | 20 | Mozambique |
| Mozambique | Africa | 21 | None |
| Japan | Asia | 1 | Taiwan |
| Taiwan | Asia | 2 | South Korea |

Skip to main content

| name | continent | dense_rank | lead |
|---|---|---|---|
| Sri Lanka | Asia | 4 | China |
| China | Asia | 5 | Turkey |
| Turkey | Asia | 6 | Malaysia |
| Malaysia | Asia | 7 | North Korea |

*78 rows, truncated to displaylimit of 30*

```
%%sql

WITH cte(name, continent, rank, lead_rank) AS
(
SELECT
    name,
    continent,
    DENSE_RANK()
        OVER (PARTITION BY continent ORDER BY lifeexpectancy DESC),
    LEAD(name, 1)
        OVER (PARTITION BY continent ORDER BY lifeexpectancy DESC)
FROM
    country
WHERE
    population > 10000000
)
SELECT
    *
FROM
    cte
WHERE
    rank = 1
```

```
   postgresql://postgres:***@localhost/imdb
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
6 rows affected.
```

Skip to main content

| name | continent | rank | lead_rank |
|---|---|---|---|
| Algeria | Africa | 1 | Morocco |
| Japan | Asia | 1 | Taiwan |
| Italy | Europe | 1 | France |
| Canada | North America | 1 | United States |
| Australia | Oceania | 1 | None |
| Chile | South America | 1 | Argentina |

**Example:** Write a query that in each row returns each movie's name, production year (`start_year`), and its number of votes, for all movies produced in or after 2018. Your query should also return a column that ranks each movie in descending order, according to its number of votes in the year it was produced.

- Sort your results according the rank value in ascending order
- Retrieve only 20 rows

```
%sql postgresql://{username}:{password}@{host}:{port}/imdb
```

```
'Connected: postgres@imdb'
```

```
%%sql

SELECT
    title,
    start_year,
    nvotes,
    RANK()
        OVER (PARTITION BY start_year ORDER BY nvotes DESC)
            AS rank
FROM
    movies
WHERE
```

Skip to main content

```
    rank
LIMIT
    20
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/imdb
   postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
20 rows affected.
```

| title | start_year | nvotes | rank |
|---|---|---|---|
| Avengers: Infinity War | 2018 | 711500 | 1 |
| Avengers: Endgame | 2019 | 567968 | 1 |
| Black Panther | 2018 | 543002 | 2 |
| Captain Marvel | 2019 | 362609 | 2 |
| Once Upon a Time... in Hollywood | 2019 | 204266 | 3 |
| Deadpool 2 | 2018 | 416811 | 3 |
| Bohemian Rhapsody | 2018 | 382622 | 4 |
| Spider-Man: Far from Home | 2019 | 198519 | 4 |
| A Quiet Place | 2018 | 325428 | 5 |
| Shazam! | 2019 | 182200 | 5 |
| John Wick: Chapter 3 - Parabellum | 2019 | 172900 | 6 |
| Ready Player One | 2018 | 316326 | 6 |
| Alita: Battle Angel | 2019 | 162513 | 7 |
| Venom | 2018 | 305111 | 7 |
| Glass | 2019 | 159064 | 8 |
| Aquaman | 2018 | 300064 | 8 |
| Us | 2019 | 150951 | 9 |
| A Star Is Born | 2018 | 276484 | 9 |
| Spider-Man: Into the Spider-Verse | 2018 | 262505 | 10 |
| Aladdin | 2019 | 139370 | 10 |

Skip to main content

**Example:** We want to find out how movies are ranked in their respective genres. Write a query that in each row returns movie name, genre, and production year (`start_year`) for movies with more than 1,000,000 votes. Your query should also have a column that shows the ranking of each movie according to its rating in descending order.

- Sort your results by genre and rank in ascending order
- Limit your results to 50 rows

```
%%sql

SELECT
    title,
    genre,
    start_year,
    RANK()
        OVER (PARTITION BY genre ORDER BY rating)
            AS rank
FROM
    movies m
JOIN
    movie_genres mg
ON
    m.id = mg.movie_id
WHERE
    nvotes > 1000000
ORDER BY
    genre, rank
LIMIT
    50
;
```

```
   postgresql://postgres:***@localhost/imdb
 * postgresql://postgres:***@localhost:5432/imdb
   postgresql://postgres:***@localhost:5432/mds
   postgresql://postgres:***@localhost:5432/world
50 rows affected.
```

Skip to main content

| title | genre | start_year | rank |
|---|---|---|---|
| Avatar | action | 2009 | 1 |
| The Avengers | action | 2012 | 2 |
| Batman Begins | action | 2005 | 3 |
| The Dark Knight Rises | action | 2012 | 4 |
| Gladiator | action | 2000 | 5 |
| Star Wars: Episode IV - A New Hope | action | 1977 | 6 |
| The Matrix | action | 1999 | 7 |
| Star Wars: Episode V - The Empire Strikes Back | action | 1980 | 7 |
| Inception | action | 2010 | 9 |
| The Dark Knight | action | 2008 | 10 |
| Avatar | adventure | 2009 | 1 |
| The Avengers | adventure | 2012 | 2 |
| Batman Begins | adventure | 2005 | 3 |
| Inglourious Basterds | adventure | 2009 | 4 |
| Gladiator | adventure | 2000 | 5 |
| Interstellar | adventure | 2014 | 6 |
| Star Wars: Episode IV - A New Hope | adventure | 1977 | 6 |
| Star Wars: Episode V - The Empire Strikes Back | adventure | 1980 | 8 |
| The Lord of the Rings: The Two Towers | adventure | 2002 | 8 |
| The Lord of the Rings: The Fellowship of the Ring | adventure | 2001 | 10 |
| Inception | adventure | 2010 | 10 |
| The Lord of the Rings: The Return of the King | adventure | 2003 | 12 |
| The Wolf of Wall Street | biography | 2013 | 1 |
| Schindler's List | biography | 1993 | 2 |
| The Wolf of Wall Street | crime | 2013 | 1 |
| The Departed | crime | 2006 | 2 |
| Se7en | crime | 1995 | 3 |

Skip to main content

| title | genre | start_year | rank |
|---|---|---|---|
| The Green Mile | crime | 1999 | 3 |
| The Silence of the Lambs | crime | 1991 | 3 |
| Pulp Fiction | crime | 1994 | 6 |

*50 rows, truncated to displaylimit of 30*

It is also possible to define multiple windows over which multiple window functions may operate:

```
%sql postgresql://{username}:{password}@{host}:{port}/world
```

```
'Connected: postgres@world'
```

```
%%sql

SELECT
    name,
    RANK() OVER w1 AS region_rank,
    RANK() OVER w2 AS continent_rank
FROM
    country
WHERE
    population > 1000000
WINDOW
    w1 AS (PARTITION BY region ORDER BY lifeexpectancy DESC),
    w2 AS (PARTITION BY continent ORDER BY lifeexpectancy DESC)
ORDER BY
    region_rank
```

```
   postgresql://postgres:***@localhost/imdb
   postgresql://postgres:***@localhost:5432/imdb
   postgresql://postgres:***@localhost:5432/mds
 * postgresql://postgres:***@localhost:5432/world
154 rows affected.
```

Skip to main content

| name | region_rank | continent_rank |
|---|---|---|
| Cameroon | 1 | 12 |
| Czech Republic | 1 | 18 |
| Israel | 1 | 4 |
| Cuba | 1 | 3 |
| Senegal | 1 | 7 |
| Canada | 1 | 1 |
| Mauritius | 1 | 3 |
| Chile | 1 | 1 |
| Sri Lanka | 1 | 10 |
| South Africa | 1 | 17 |
| Libyan Arab Jamahiriya | 1 | 1 |
| Italy | 1 | 3 |
| United Kingdom | 1 | 10 |
| Papua New Guinea | 1 | 3 |
| Sweden | 1 | 1 |
| Estonia | 1 | 29 |
| Switzerland | 1 | 1 |
| Australia | 1 | 1 |
| Singapore | 1 | 2 |
| Japan | 1 | 1 |
| Costa Rica | 1 | 4 |
| Jordan | 2 | 5 |
| Hong Kong | 2 | 3 |
| Spain | 2 | 4 |
| Malaysia | 2 | 16 |
| Ireland | 2 | 14 |
| Ghana | 2 | 8 |

Skip to main content

| name | region_rank | continent_rank |
|---|---|---|
| Slovakia | 2 | 20 |
| Chad | 2 | 21 |
| Norway | 2 | 6 |

*154 rows, truncated to displaylimit of 30*

# Indexing and performance (optional)

Check the optional notebook if you want to learn more about indexing and performance.