Lecture 2: Data types, filtering, functions

Contents

DSCI 513

Databases & Data Retrieval

Lecture outline

- Various data types in SQL
- `WHERE` conditionals, pattern matching
- Derived columns, aliases with `AS`
- Conditionals with `CASE`
- Functions and operators in SQL

```
%load_ext sql
%config SqlMagic.displaylimit = 20
```

Skip to main content

```
import json
import urllib.parse

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

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

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

Data types

You might remember from previous lecture that in relational databases, each column is characterized with its name and its **domain**. A domain is the set of permissible or valid values that a column is allowed to store. This highlights one of the advantages of using a DBMS, which enforces particular data types for the columns of a table.

Postgres supports

- boolean
- character
- number
- datetime
- binary

and some extension types specific to Postgres.

Type conversion

To demonstrate how different data types work in SQL, I first need to show you how we

Skip to main content

the `CAST` function:

```
CAST(<column> AS <data_type>)
```

In Postgres, we can also use the double-colon syntax as a shorthand for the above `CAST` function:

```
<column>::<data_type>
```

Boolean

We can specify this data type using the keyword `BOOLEAN` or `BOOL`.

Valid values are

- `'TRUE'`, `1` (or any other positive integer), `'YES'`, `'Y'`, `'T'`,
- `'FALSE'`, `'0'`, `'NO'`, `'N'`, `'F'`

Note that all of these values will be interpreted as `'TRUE'`, `'FALSE'` by Postgres:

```
%%sql

SELECT
    'TRUE'::BOOLEAN,
    'T'::BOOLEAN,
    '0'::BOOLEAN,
    'NO'::BOOLEAN
;
```

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

bool	bool_1	bool_2	bool_3
True	True	False	False

Skip to main content

Characters

The character data type is used to represent fixed-length and variable length character strings. This type can be defined using the following keywords:

- `CHAR(n)`: a string of exactly `n` characters padded with spaces
- `VARCHAR(n)`: a variable set of `n` characters
- `TEXT` which is a Postgres specific type for which there is practically no limit on the number of characters.

```
%%sql

SELECT
    'Arman'::CHAR(50),
    'Arman'::VARCHAR(2),
    'Arman'::TEXT
;
```

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

bpchar	varchar	text
Arman	Ar	Arman

> Note that you can't see the space-padding for `CHAR(50) 'Arman'` in the Jupyter notebook, but if you run the same statement in `psql`, you will see `'Arman'` + 45 spaces in the output.

Skip to main content

Numbers

Numerical values in Postgres belong to the following general categories:

- Integers
- Floating-point numbers
- Arbitrary precision numbers

Integers:

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`serial`	4 bytes	auto-incrementing integer	1 to 2147483647
`bigserial`	8 bytes	large auto-incrementing integer	1 to 9223372036854775807

We'll learn later that the `serial` type (which is not an actual data type) is a shortcut to tell Postgres create unique "auto-incrementing" often used for the primary key column of table.

Floating-point numbers:

Name	Storage Size	Description	Range
`real`	4 bytes	variable-precision, inexact	at least 6 decimal digits (implementation dependent)
`double`	8 bytes	variable-precision,	at least 15 decimal digits

Skip to main content

Arbitrary precision numbers

Name	Storage Size	Description	Range
`numeric`	variable	user-specified precision, exact	131072 digits before and 16383 digits after the decimal point
`decimal`	variable	user-specified precision, exact	131072 digits before and 16383 digits after the decimal point

> `DECIMAL` and `NUMERIC` data types are exactly the same thing in Postgres.

```
%%sql

SELECT CAST(44.7874 AS SMALLINT);
```

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

int2
45

Note the `::` notation can also be used instead of `CAST()` (and maybe preferred, but specific to Postgres):

```
%%sql

SELECT 44.7874::SMALLINT;
```

```
 * postgresql://postgres:***@localhost:5432/imdb
```

Skip to main content

int2

45

```
%%sql

SELECT CAST(4.54021223948E-8 AS REAL);
```

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

float4

4.540212e-08

With the `numeric` data type, we can specify the total number of significant digits to store (known as precision) as well as the number of digits in the fractional part (known as scale) by specifying `NUMERIC(precision, scale)`:

```
%%sql

SELECT CAST('183.123456789' AS NUMERIC(5, 2));
```

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

numeric

183.12

The `numeric` type is exact (as opposed to other types of floats) and immune to the round-off error, but it is **slow to work with for the DBMS**. It is often used for monetary and financial data, where either numbers with a many digits may be stored or exactness is important.

For example, the following number cannot be represented as `BIGINT` and would throw an error, but it works with `NUMERIC`:

```
%%sql

SELECT CAST(92233720368547758089832778534345309 82 AS NUMERIC);
```

Skip to main content

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

numeric
92233720368547758089832778534345309 82

Date/time

Postgres provides **datetime** and **interval** data types similar to those we've seen in DSCI 511 in Python and Pandas.

Datetimes

- `DATE` for dates
- `TIME` for the time of day

Postgres also provides two ways to store the **timestamp** datatype;

- `TIMESTAMP` for date + time
- `TIMESTAMPTZ` for date + time + timezone (Postgres specific)

When a timestamp value is queried:

- For `TIMESTAMP`, Postgres returns the timestamp as originally stored in the database server
- For `TIMESTAMPTZ`, Postgres converts the timestamp into the local timezone of the database server

Note that Postgres does not store timezone information. It always internally stores `TIMESTAMPTZ` in UTC value, and does the back-conversion using the local time zone of the database server.

Skip to main content

Entering datetime data

Postgres does a pretty good job of getting the datetimes right even if we don't enter them in the standard ISO way. Let's take a look at a few examples:

```
%%sql

SELECT
    'January 23, 2021'::DATE,
    '23 January 2021'::DATE,
    '2021 1 23'::DATE,
    '1/23/2021'::DATE,
    'today'::DATE,
    'tomorrow'::DATE
;
```

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

date	date_1	date_2	date_3	date_4	date_5
2021-01-23	2021-01-23	2021-01-23	2021-01-23	2022-11-17	2022-11-18

```
%%sql

SELECT
    '14:24:00'::TIME,
    '2:24pm'::TIME,
    '2:24 PM PST'::TIME WITH TIME ZONE,
    'now'::TIME,
    'now'::TIME WITH TIME ZONE
;
```

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

time	time_1	timetz	time_2	timetz_1
14:24:00	14:24:00	14:24:00-08:00	02:11:08.571179	02:11:08.571179-08:00

Skip to main content

When datetime is stored without timezone, it is oblivious to the local server timezone:

```
%sql SELECT '2021-11-18 8:30:00'::TIMESTAMP;
```

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

timestamp
2021-11-18 08:30:00

```
%sql SELECT '2021-11-18 8:30:00'::TIMESTAMPTZ;
```

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

timestamptz
2021-11-18 08:30:00-08:00

```
%sql SHOW TIMEZONE;
```

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

TimeZone
America/Vancouver

```
%sql SET timezone = 'America/New_York';
```

Skip to main content

```
 * postgresql://postgres:***@localhost:5432/
Done.
```

```
[]
```

```
%sql SELECT '2021-11-18 8:30:00 -08:00'::TIMESTAMPTZ;
```

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

timestamptz
2021-11-18 11:30:00-05:00

```
%sql SET timezone = 'America/Vancouver';
```

```
 * postgresql://postgres:***@localhost:5432/
Done.
```

```
[]
```

```
%sql SELECT '2021-11-18 8:30:00'::TIMESTAMPTZ;
```

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

timestamptz
2021-11-18 08:30:00-08:00

Skip to main content

Intervals (OPTIONAL)

There is also another datatype for storing intervals of time. Intervals are useful for doing date and time arithmetic, such as adding a duration of time to a timestamp.

For more detailed information, refer to the Postgres documentation here.

```
%%sql

SELECT
    '1 day 23 hours 8 minutes'::INTERVAL,
    '2m 18s'::INTERVAL,
    '3 years 2 months'::INTERVAL
;
```

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

interval	interval_1	interval_2
1 day, 23:08:00	0:02:18	1155 days, 0:00:00

Binary data (OPTIONAL)

It is also possible to have binary data in a table (e.g. documents, images, videos). We don't use binary data in this course.

Nulls

A null is marker to indicate that the value for a column is unknown, or not entered yet. A null is not equal to 0, or an empty string. In fact, a null is not even equal to another null!

Skip to main content

- `ipython-sql` -> `None`
- psql -> blank space
- pgAdmin -> `[null]`

Filtering rows with `WHERE`

We've seen the `WHERE` keyword in passing in the last lecture. `WHERE` is an intuitive keyword that is used to filter rows based on a particular condition. The syntax is as follows:

```
SELECT
    column1, column2
FROM
    table1
WHERE
    condition
;
```

Condition	Operator
Comparison	`=`, `<>`, `<`, `<=`, `>`, `>=`
Pattern matching	`LIKE`
Range	`BETWEEN`
List	`IN`
Null testing	`IS NULL`

```
%%sql
```

Skip to main content

```
    *
FROM
    movies;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
26058 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10035423	Kate & Leopold	None	2001	None	118	6.4	74982
10042742	Mister 880	None	1950	None	90	7.1	1171
10041181	Black Hand	None	1950	None	92	6.4	666
10041387	Francis	None	1950	None	91	6.4	979
10041719	Orpheus	Orphée	1950	None	95	8.0	9346
10041931	Stromboli	Stromboli, terra di Dio	1950	None	107	7.3	5239
10042052	Woman in Hiding	None	1950	None	92	6.9	553
10042179	Abbott and Costello in the Foreign Legion	None	1950	None	80	6.6	2573
10042200	Annie Get Your Gun	None	1950	None	107	6.9	4050
10042206	Armored Car Robbery	None	1950	None	67	7.0	2077
10042208	The Asphalt Jungle	None	1950	None	112	7.9	22106
10042211	Atom Man vs. Superman	None	1950	None	252	7.0	579
10042219	Backfire	None	1950	None	91	6.6	955
10042229	The Baron of Arizona	None	1950	None	97	7.0	1748
10042249	The Big Lift	None	1950	None	120	6.5	1266
10042256	The Black Rose	None	1950	None	120	6.3	1657
10042265	The Blue Lamp	None	1950	None	84	6.9	1426
10042274	Borderline	None	1950	None	88	6.1	996

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10042275	Born to Be Bad	None	1950	None	94	6.7	1797
10042279	Branded	None	1950	None	104	6.7	702

26058 rows, truncated to displaylimit of 20

Example: Retrieve rows for movies produced in or after 2010.

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    start_year >= 2010
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
8804 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10069049	The Other Side of the Wind	None	2018	None	122	6.9	4904
10176694	The Tragedy of Man	Az ember tragédiája	2011	None	160	7.8	610
10293069	Dark Blood	None	2012	None	86	6.5	1073
10315642	Wazir	None	2016	None	103	7.1	15796
10337692	On the Road	None	2012	None	124	6.1	38216
10359950	The Secret Life of Walter Mitty	None	2013	None	114	7.3	278645
10365907	A Walk Among the Tombstones	None	2014	None	114	6.5	106413
10369610	Jurassic World	None	2015	None	124	7.0	547391
10376136	The Rum Diary	None	2011	None	119	6.2	95417
10376479	American Pastoral	None	2016	None	108	6.1	13376
10398286	Tangled	None	2010	None	100	7.7	373355
10401729	John Carter	None	2012	None	132	6.6	244429
10409379	In Secret	None	2013	None	107	6.1	7146
10409847	Cowboys & Aliens	None	2011	None	119	6.0	197867
10420293	The Stanford Prison Experiment	None	2015	None	122	6.9	33319
10429493	The A-Team	None	2010	None	117	6.7	237537
10433035	Real Steel	None	2011	None	127	7.1	285715
10435761	Toy Story 3	None	2010	None	103	8.3	701340

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10437086	Alita: Battle Angel	None	2019	None	122	7.4	162513

8804 rows, truncated to displaylimit of 20

Example: Retrieve the row for the movie called “Lost Highway”.

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    title = 'Lost Highway'
;
```

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

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10116922	Lost Highway	None	1997	None	134	7.6	120549

> Note that in SQL, strings are enclosed in single quotes, i.e. `'string'`.

> While SQL syntax is case-insensitive, SQL is **case-sensitive** when it comes to **comparing strings**. In the above example, `'Lost highway'` will not return any rows.

Skip to main content

Logical operators `AND`, `OR`, and `NOT`

Just like in Python, we can combine multiple conditions logical/boolean operators `AND`, `OR`, and `NOT`.

When there are multiple logical operators, `NOT` is evaluated first, then `AND` and finally `OR`.

We can enclose each condition in parentheses if we want. This can be done either for readability, or to override the default precedence rules.

Example: Retrieve the rows for movies that are produced in 2015 and are rated higher than 8.

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    start_year = 2015
    AND
    rating > 8
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
0 rows affected.
```

id	title	orig_title	start_year	end_year	runtime	rating	nvotes

---

Example: Retrieve the rows for movies that are produced either in 2015 or 2018, and are rated higher than 8.

```
%%sql
```

Skip to main content

```
    *
FROM
    movies
WHERE
    start_year = 2015
    OR
    start_year = 2018
    AND
    rating > 8
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1048 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10369610	Jurassic World	None	2015	None	124	7.0	547391
10420293	The Stanford Prison Experiment	None	2015	None	122	6.9	33319
10478970	Ant-Man	None	2015	None	117	7.3	517941
10790770	Miles Ahead	None	2015	None	100	6.4	8650
10884732	The Wedding Ringer	None	2015	None	101	6.6	67575
11533089	Tab Hunter Confidential	None	2015	None	90	7.8	2852
11596363	The Big Short	None	2015	None	130	7.8	318033
11598642	Z for Zachariah	None	2015	None	98	6.0	25985
11618448	Racing Extinction	None	2015	None	90	8.3	7042
11638355	The Man from U.N.C.L.E.	None	2015	None	116	7.3	245184
11655441	The Age of Adaline	None	2015	None	112	7.2	141949
11658801	Freeheld	None	2015	None	103	6.6	10772
11663202	The Revenant	None	2015	None	156	8.0	636721
11666801	The Duff	None	2015	None	101	6.5	76639
11674771	Entourage	None	2015	None	104	6.6	72367
11683048	De Palma	None	2015	None	110	7.4	4202
11698654	Catching the Sun	None	2015	None	75	6.8	517
11702429	Eisenstein in	None	2015	None	105	6.2	2317

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
11708135	Elser	None	2015	None	114	7.0	7455
11727770	Absolutely Anything	None	2015	None	85	6.0	35659

1048 rows, truncated to displaylimit of 20

What? This isn't the right result! We have multiple returned movies that are rated below 8.

The reason is that the `AND` operator takes precedence over `OR`. Therefore, `start_year = 2018 AND rating > 8` gets evaluated first, and then the result is passed to the `OR` part of the condition. In order to override this behaviour, we can rewrite our query in the following way:

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    (start_year = 2015
    OR
    start_year = 2018)
    AND
    rating > 8
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
119 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
11618448	Racing Extinction	None	2015	None	90	8.3	7042
12096673	Inside Out	None	2015	None	95	8.2	550606
12473476	Be Here Now	None	2015	None	100	8.7	2863
12631186	Baahubali: The Beginning	Bahubali: The Beginning	2015	None	159	8.1	94989
12865822	All the World in a Design School	None	2015	None	59	8.4	1270
13170832	Room	None	2015	None	118	8.2	326042
13270538	Requiem for the American Dream	None	2015	None	73	8.1	8061
13717510	The Drop Box	None	2015	None	79	8.1	604
13865286	My Lonely Me	None	2015	None	95	8.2	671
14112208	Kuttram Kadithal	None	2015	None	120	8.1	638
14145178	Listen to Me Marlon	None	2015	None	103	8.2	6447
14154756	Avengers: Infinity War	None	2018	None	149	8.5	711500
14393514	Bitter Lake	None	2015	None	136	8.2	2356
14430212	Drishyam	None	2015	None	163	8.3	57527
14429128	Papanasam	None	2015	None	179	8.4	4802
14449576	Tomorrow	Demain	2015	None	118	8.1	2887
14476736	Hamlet	National Theatre Live: Hamlet	2015	None	217	8.6	1670

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
14519488	Mudras Calling	None	2018	None	95	8.7	1196
14635372	Masaan	None	2015	None	109	8.1	19638
14633694	Spider-Man: Into the Spider-Verse	None	2018	None	117	8.4	262505

119 rows, truncated to displaylimit of 20

Example: Count the number of movies that have no less than 1 million votes.

We need to use the `COUNT()` function to count the number of returned rows (more on `COUNT()` in a later lecture):

```
%%sql

SELECT
    COUNT(*)
FROM
    movies
WHERE
    NOT nvotes < 1000000
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1 rows affected.
```

count
33

Skip to main content

Example: Find the genres listed for the movie “The Godfather”.

We will learn in later lectures how to answer this question in a single SQL query in various ways, but for now, we have to take a two step process. Here, we are trying to find information related to each are that are stored in two tables. This is the first time we actually encounter the notion of a **relational** database in practice!

It turns out that are the `id` column in the `movie` table and `movie_id` in `movies_genre` table reference the same movies. These columns actually relate to two tables together. For our query, we have to find out the `id` of the movie `'The Godfather'` first, and then use it to retrieve the genres associated with that movie:

```
%%sql

SELECT
    id, title
FROM
    movies
WHERE
    title = 'The Godfather'
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1 rows affected.
```

id	title
10068646	The Godfather

Alright, the id for `'The Godfather'` is `10068646`. Now let’s retrieve the genres for this id from the `movie_genres` table:

```
%%sql

SELECT
    *
FROM
    movie_genres
WHERE
    movie_id = 10068646
```

Skip to main content

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
2 rows affected.
```

movie_id	genre
10068646	crime
10068646	drama

Question: Do you think using the following query, you can find the number of movies in the `movie_genres` table that are NOT listed as `'drama'`?

```
SELECT
    COUNT(DISTINCT movie_id)
FROM
    movie_genres
WHERE
    genre <> 'drama'
;
```

Pattern matching

It is a quite common situation that we want to find rows for which the values of one or more columns match a particular pattern. In SQL, this can be done either using `LIKE` or by using regular expressions. The syntax is as follows:

```
SELECT
    column1, column2
FROM
    table1
```

Skip to main content

```
    column1 [NOT] LIKE '<pattern>'
;
```

Postgres provides us with two wild-cards that we can use with `LIKE`:

- `%` matches any string of characters
- `_` matches a single character.

Pattern matching with `LIKE` is case sensitive; however, Postgres also provides the `ILIKE` keyword that has the same functionality as `LIKE` but is case-insensitive.

> **Note:** With `LIKE` or `ILIKE`, the entire string should match the pattern.

```
%%sql

SELECT
    'Arman' LIKE '%a_',
    'UBC' LIKE '_B_',
    'MDS is awesome!' LIKE '%!_'
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1 rows affected.
```

?column?	?column?_1	?column?_2
True	True	False

Example: Retrieve those movies from the `movie` table whose title contains the word `'violin'` (note that `LIKE` is picky about letter cases in strings!)

```
%%sql

SELECT
```

Skip to main content

```
FROM
    movies
WHERE
    title LIKE '%Violin%'
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
5 rows affected.
```

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10120802	The Red Violin	Le violon rouge	1998	None	130	7.6	30285
10451966	The Violin	El violín	2005	None	98	7.7	2212
12401715	The Devil's Violinist	None	2013	None	122	6.1	3033
14972904	The Violin Teacher	Tudo Que Aprendemos Juntos	2015	None	102	6.8	645
10053987	The Steamroller and the Violin	Katok i skripka	1961	None	46	7.5	4867

Example: Retrieve those movies from the `movie` table whose title starts with the word `'Zero'`.

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    title LIKE 'Zero%'
;
```

Skip to main content

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
18 rows affected.
```

id	title	orig_title	start_year	end_year	runtime	rating	nvo
10095244	Zerograd	Gorod Zero	1988	None	103	7.5	1
10113557	Zero Kelvin	Kjærlighetens kjøtere	1995	None	118	7.3	
10120906	Zero Effect	None	1998	None	116	6.9	13
10198837	Zero Tolerance	Noll tolerans	1999	None	108	6.4	3
10283693	Zero Woman: Red Handcuffs	Zeroka no onna: Akai wappa	1974	None	88	6.6	
10365960	Zero Day	None	2002	None	92	7.2	3
10421090	Zerophilia	None	2005	None	90	6.2	2
11592292	Zero 2	None	2010	None	90	7.6	5
11790885	Zero Dark Thirty	None	2012	None	157	7.4	254
12294965	Zero Charisma	None	2013	None	86	6.2	2
13576084	Zero Motivation	Efes beyahasei enosh	2014	None	97	7.3	3
15446858	Zero Days	None	2016	None	116	7.8	8
10051221	Zero Hour!	None	1957	None	81	6.6	1
10968712	Zero. Lilac Lithuania	Zero. Alyvine Lietuva	2006	None	82	7.5	2
11297858	Zero: An Investigation Into 9/11	None	2008	None	104	7.8	
11519661	Zero	None	2009	None	110	6.8	
12323372	Zero Point	Nullpunkt	2014	None	115	7.4	
16194850	Zero 3	None	2017	None	88	6.6	1

Skip to main content

Example: Retrieve those movies from the `movie` table whose title is 4 letters long and ends with the letter `'e'`.

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    title LIKE '___e'
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
71 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10043539	Five	None	1951	None	93	6.3	1068
10064694	More	None	1969	None	112	6.5	2155
10066500	Hope	Umut	1970	None	100	8.2	2770
10067814	Love	Szerelem	1971	None	88	7.9	1582
10068306	Bone	None	1972	None	95	6.8	905
10069158	Rage	None	1972	None	100	6.3	765
10071803	Mame	None	1974	None	132	6.1	2490
10080716	Fame	None	1980	None	134	6.6	18864
10087182	Dune	None	1984	None	137	6.5	113255
10088930	Clue	None	1985	None	94	7.3	71433
10106438	Blue	None	1993	None	79	7.3	1778
10112431	Babe	None	1995	None	91	6.7	110464
10114323	Safe	None	1995	None	119	7.2	11475
10116308	Fire	None	1996	None	108	7.2	5386
10116722	Jude	None	1996	None	123	6.9	9547
10123948	Cure	None	1997	None	111	7.4	8963
10123755	Cube	None	1997	None	90	7.2	195849
10123964	Life	None	1999	None	108	6.8	41588
10188568	Rage	Do koske	1997	None	97	6.9	1623
10217019	Sade	None	2000	None	100	6.2	1609

71 rows, truncated to displaylimit of 20

Example: Retrieve those movies from the `movie` table whose title contains the character `'%'`.

Skip to main content

We can specify an escape character using the keyword `ESCAPE` that tells SQL to not interpret a `%` or `_` that immediately follows it:

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    title LIKE '%$%%' ESCAPE '$'
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
3 rows affected.
```

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10487092	Who the #$&% Is Jackson Pollock?	None	2006	None	74	7.0	1134
12662228	10%: What Makes a Hero?	None	2013	None	88	6.8	543
11869226	100% Love	None	2011	None	141	7.0	2369

`IN`

Sometimes we want to check whether a column value matches any one of the items in a list. We can express this with an `OR` operator:

```
SELECT
    column1, column2
FROM
    table1
```

Skip to main content

```
    OR
    column1 = value2
    OR
    column1 = value3
;
```

This can be rewritten more succinctly using the `IN` operator:

```
SELECT
    column1, column2
FROM
    table1
WHERE
    column1 [NOT] IN (value1, value2, value3)
;
```

Example: Retrieve rows from the `movie` table that correspond to the movies `'Donnie Brasco'`, `'The Usual Suspects'`, `'Schindler''s List'`, `'Shutter Island'`, `'A Beautiful Mind'`.

```
%%sql

SELECT
    *
FROM
    movies
WHERE
    title IN ('Donnie Brasco',
              'The Usual Suspects',
              'Schindler''s List',
              'Shutter Island',
              'A Beautiful Mind'
               )
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
5 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10108052	Schindler's List	None	1993	None	195	8.9	1110590
10114814	The Usual Suspects	None	1995	None	106	8.5	922333
10119008	Donnie Brasco	None	1997	None	127	7.7	258120
10268978	A Beautiful Mind	None	2001	None	135	8.2	784095
11130884	Shutter Island	None	2010	None	138	8.1	1027318

`BETWEEN`

The `BETWEEN` keyword is helpful for when we want to select a range of values, and it can be used for number, character and datetime ranges:

```
SELECT
    column1, column2
FROM
    table1
WHERE
    column1 [NOT] BETWEEN value1 AND value2
;
```

> **Note:** `BETWEEN` is **inclusive** of both ends of the interval.

We can try it out using a `SELECT` statement without any tables:

```
%%sql

SELECT
    5 BETWEEN 1 AND 10,
```

Skip to main content

```
    DATE '2021-11-01' BETWEEN DATE '2021-01-01' AND '2021-11-10',
    'w' BETWEEN 'e' AND 'm';
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1 rows affected.
```

?column?	?column?_1	?column?_2
True	True	False

Example: Retrieve the name, production year and rating of the top 5 movies from the `movie` table that are produced between 2018 and 2020, and have a rating of at least 8.5 with at least 100000 votes. Sort the results in descending order based on ratings.

```
%%sql

SELECT
    title, start_year, rating
FROM
    movies
WHERE
    start_year BETWEEN 2018 AND 2020
    AND
    rating >= 8.5
    AND
    nvotes >= 100000
ORDER BY
    rating DESC
LIMIT
    5
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
5 rows affected.
```

Skip to main content

title	start_year	rating
Once Upon a Time... in Hollywood	2019	8.0
Toy Story 4	2019	8.0
Bohemian Rhapsody	2018	8.0
Green Book	2018	8.2
Spider-Man: Into the Spider-Verse	2018	8.4

`IS NULL`

Trying to find `NULL` values using `WHERE column = NULL` fails. This is because a `NULL` value is by definition not known and *could be anything*, so it's not necessarily equal to another `NULL`. To find `NULL` values in a column, we can use `IS NULL`:

```
SELECT
    column1, column2
FROM
    table1
WHERE
    column1 IS [NOT] NULL
;
```

Example: Find movies the `movie` whose `orig_title` is different from that listed in the `title` column.

```
%%sql

SELECT
```

Skip to main content

```
    movies
WHERE
    orig_title IS NOT NULL
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
8270 rows affected.
```

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
10041719	Orpheus	Orphée	1950	None	95	8.0	9346
10041931	Stromboli	Stromboli, terra di Dio	1950	None	107	7.3	5239
10042355	Story of a Love Affair	Cronaca di un amore	1950	None	98	7.1	2209
10042619	Diary of a Country Priest	Journal d'un curé de campagne	1951	None	115	8.0	8621
10042692	Variety Lights	Luci del varietà	1950	None	97	7.1	2416
10042804	The Young and the Damned	Los olvidados	1950	None	85	8.3	16453
10042810	Operation Disaster	Morning Departure	1950	None	102	7.0	668
10042876	Rashomon	Rashômon	1950	None	88	8.2	138304
10042906	La Ronde	La ronde	1950	None	93	7.6	4456
10043048	To Joy	Till glädje	1950	None	98	7.2	2109
10043303	The Red Inn	L'auberge rouge	1951	None	98	7.4	1201
10043313	Early Summer	Bakushû	1951	None	125	8.2	6349
10043386	Casque d'Or	Casque d'or	1952	None	96	7.7	4414
10043411	The Emperor and the Golem	Císaruv pekar - Pekaruv císar	1952	None	144	8.1	1069
10043511	Europe '51	Europa '51	1952	None	118	7.5	2874
10043567	Miss Julie	Fröken Julie	1951	None	90	7.3	1499
10043614	The Idiot	Hakuchi	1951	None	166	7.3	4110
10043652	One	Hon	1951	None	103	6.9	623

Skip to main content

id	title	orig_title	start_year	end_year	runtime	rating	nvotes
	of Happiness	en sommar					
10043668	I'll Never Forget You	The House in the Square	1951	None	90	7.1	716
10043686	Forbidden Games	Jeux interdits	1952	None	86	8.1	10142

8270 rows, truncated to displaylimit of 20

Column Aliases with `AS`

In SQL, we are not required to use the same column and table names in the schema. We can create **aliases** for a column or a table with the following syntax:

```
SELECT
    column1 [AS] c1,
    column2 [AS] c2
FROM
    table1 [AS] t1
;
```

Note that the keyword `AS` is optional. I usually choose to use it because it makes the query more readable.

We will use table aliases a lot when we work on SQL joins in the upcoming lectures!

```
%%sql

SELECT
    title AS movieTitle,
    orig_title AS "oringinal Title",
    runtime AS Duration
FROM
    movies;
```

Skip to main content

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb
26058 rows affected.
```

movietitle	oringinal Title	duration
Kate & Leopold	None	118
Mister 880	None	90
Black Hand	None	92
Francis	None	91
Orpheus	Orphée	95
Stromboli	Stromboli, terra di Dio	107
Woman in Hiding	None	92
Abbott and Costello in the Foreign Legion	None	80
Annie Get Your Gun	None	107
Armored Car Robbery	None	67
The Asphalt Jungle	None	112
Atom Man vs. Superman	None	252
Backfire	None	91
The Baron of Arizona	None	97
The Big Lift	None	120
The Black Rose	None	120
The Blue Lamp	None	84
Borderline	None	88
Born to Be Bad	None	94
Branded	None	104

26058 rows, truncated to displaylimit of 20

Note that we've used a column alias with a space in its name. This is generally not a good practice, but if you absolutely need to do it, in Postgres you should enclose the alias in double quotes, e.g. `"alias name"`. A situation where double quotes are necessary is when you want to name a column with a word that is reserved keyword in Postgres, e.g. `"COUNT"`

Skip to main content

Order of execution/processing in SQL

If you try running the following cell, you'll notice that you **cannot** use column aliases in the `WHERE` clause, since it is evaluated by SQL before setting aliases:

```
%%sql

SELECT
    title AS movieTitle,
    orig_title AS "oringinal Title",
    runtime AS Duration
FROM
    movies
WHERE
    Duration > 100
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb
(psycopg2.errors.UndefinedColumn) column "duration" does not exist
LINE 8:     Duration > 100
            ^

[SQL: SELECT
    title AS movieTitle,
    orig_title AS "oringinal Title",
    runtime AS Duration
FROM
    movies
WHERE
    Duration > 100
;]
(Background on this error at: https://sqlalche.me/e/14/f405)
```

SQL-based database engines have a particular order by which they execute different clauses in your SQL query:

Order of execution/processing in SQL:

Skip to main content

```
FROM and JOIN
      |
    WHERE
      |
  GROUP BY
      |
    HAVING
      |
    SELECT
      |
   DISTINCT
      |
   ORDER BY
      |
    LIMIT
```

Contrast this with the **order of SQL clauses in a statement:**

```
    SELECT
      |
     FROM
      |
     JOIN
      |
    WHERE
      |
  GROUP BY
      |
    HAVING
      |
   ORDER BY
      |
    LIMIT
```

Derived columns

Derived columns in SQL are columns that are the result of doing operations on existing

Skip to main content

For example, suppose that we want to convert the `runtime` column of our table `movies` from minutes to hours. We can do that by manipulating the `runtime` column right in the `SELECT` statement:

```
%%sql

SELECT
    title,
    runtime / 60.
FROM
    movies;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb
26058 rows affected.
```

Skip to main content

title	?column?
Kate & Leopold	1.9666666666666667
Mister 880	1.5000000000000000
Black Hand	1.5333333333333333
Francis	1.5166666666666667
Orpheus	1.5833333333333333
Stromboli	1.7833333333333333
Woman in Hiding	1.5333333333333333
Abbott and Costello in the Foreign Legion	1.3333333333333333
Annie Get Your Gun	1.7833333333333333
Armored Car Robbery	1.1166666666666667
The Asphalt Jungle	1.8666666666666667
Atom Man vs. Superman	4.2000000000000000
Backfire	1.5166666666666667
The Baron of Arizona	1.6166666666666667
The Big Lift	2.0000000000000000
The Black Rose	2.0000000000000000
The Blue Lamp	1.4000000000000000
Borderline	1.4666666666666667
Born to Be Bad	1.5666666666666667
Branded	1.7333333333333333

26058 rows, truncated to displaylimit of 20

> Note that I've written `60.` with the decimal point on purpose. If you divide by `60` instead, SQL assumes that the result of this operation should also be an integer (given that the column `runtime` is also of type integer), and will return truncated integer values instead of floats.

Skip to main content

SQL doesn't know what to call the derived column, and by default you will see `?column?` as the column name. We can use an alias to name the new derived column:

```
%%sql

SELECT
    title,
    runtime / 60. AS runtime_hours
FROM
    movies;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb
26058 rows affected.
```

Skip to main content

title	runtime_hours
Kate & Leopold	1.9666666666666667
Mister 880	1.5000000000000000
Black Hand	1.5333333333333333
Francis	1.5166666666666667
Orpheus	1.5833333333333333
Stromboli	1.7833333333333333
Woman in Hiding	1.5333333333333333
Abbott and Costello in the Foreign Legion	1.3333333333333333
Annie Get Your Gun	1.7833333333333333
Armored Car Robbery	1.1166666666666667
The Asphalt Jungle	1.8666666666666667
Atom Man vs. Superman	4.2000000000000000
Backfire	1.5166666666666667
The Baron of Arizona	1.6166666666666667
The Big Lift	2.0000000000000000
The Black Rose	2.0000000000000000
The Blue Lamp	1.4000000000000000
Borderline	1.4666666666666667
Born to Be Bad	1.5666666666666667
Branded	1.7333333333333333

26058 rows, truncated to displaylimit of 20

Remember I mentioned that the `SELECT` statement is powerful, but not dangerous? Derived columns returned by Postgres are not saved anywhere, nor do they change existing columns.

Skip to main content

Example: Using table `names` from the `imdb` database, find the age of all actors/actresses who are still alive. Who is the youngest person alive listed in the table?

```
%%sql

SELECT
    name,
    2022 - birth_year AS age
FROM
    names
WHERE
    birth_year IS NOT NULL
    AND
    death_year IS NULL
ORDER BY
    age
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb
35413 rows affected.
```

Skip to main content

name	age
Blue Ivy Carter	10
Julia Butters	13
Run-yin Bai	13
Mckenna Grace	16
Timothy Radford	16
Momone Shinokawa	16
Jacob Tremblay	16
Rina Endô	17
Eun-hyung Jo	17
Sunny Suljic	17
Teo Briones	17
Merve Ates	17
Kristina Pimenova	17
Alexis Neblett	17
Ella Anderson	17
Noah Jupe	17
Towa Araki	17
Lulu Wilson	17
Farrah Mackenzie	17
Fantine Harduin	17

35413 rows, truncated to displaylimit of 20

Skip to main content

Conditionals with `CASE`

The `CASE` structure is very useful in SQL: it enables us to treat a column differently based on the values in each row. Here is the syntax:

```
SELECT
    column1,
    CASE
        WHEN condition THEN expression
        WHEN condition THEN expression
        .
        .
        .
        ELSE expression
    END,
    column3,
    column4
FROM
    table1
;
```

For example, let's say we want to retrieve the name of movies and also want to have a column in the results that in each row has the value "long" if a movie is over 90 minutes long, "normal" if it's between 30 to 90 minutes, and "short" if it's under 30 minutes:

```
%%sql

SELECT
    title,
    runtime,
    CASE
        WHEN runtime > 90 THEN 'long'
        WHEN runtime BETWEEN 30 AND 90 THEN 'normal'
        ELSE 'short'
    END AS duration
FROM
    movies
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
26058 rows affected.
```

Skip to main content

title	runtime	duration
Kate & Leopold	118	long
Mister 880	90	normal
Black Hand	92	long
Francis	91	long
Orpheus	95	long
Stromboli	107	long
Woman in Hiding	92	long
Abbott and Costello in the Foreign Legion	80	normal
Annie Get Your Gun	107	long
Armored Car Robbery	67	normal
The Asphalt Jungle	112	long
Atom Man vs. Superman	252	long
Backfire	91	long
The Baron of Arizona	97	long
The Big Lift	120	long
The Black Rose	120	long
The Blue Lamp	84	normal
Borderline	88	normal
Born to Be Bad	94	long
Branded	104	long

26058 rows, truncated to displaylimit of 20

Skip to main content

Functions & operators

Math

We've just seen how arithmetic operators (i.e. `+`, `-`, `*`, `/`) can be used to make derived columns. Like other programming languages, PostgreSQL comes built-in with the most common mathematical operators (for a full list of operators see the documentation of Postgres here):

Operators:

Operator	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulo (remainder)
`^`	exponentiation
`@`	absolute value

Functions

Skip to main content

Description	Example
absolute value	`abs(-17.4)`
smallest integer not less than argument	`ceil(-42.8)`
exponential	`exp(1.0)`
largest integer not greater than argument	`floor(-42.8)`
natural logarithm	`ln(2.0)`
logarithm to base b	`log(2.0, 64.0)`
“π” constant	`pi()`
a raised to the power of b	`power(9.0, 3.0)`
round to s decimal places	`round(42.4382, 2)`
square root	`sqrt(2.0)`

> Note the order of evaluation for different operators:
>
> 1. arithmetic operators (e.g. `+`, `*`)
> 2. comparison operators (e.g. `>`, `<=`)
> 3. logical operators (e.g. `AND`, `OR`)

```
%%sql

SELECT
    25 * 2,
    ABS(-2^10),
    ROUND(23.24545, 2),
    SQRT(25),
    PI()
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
```

Skip to main content

```
1 rows affected.
```

?column?	abs	round	sqrt	pi
50	1024.0	23.25	5.0	3.141592653589793

> **Note:** The `ROUND()` function only works with the `NUMERIC` (or equivalently `DECIMAL`) data type.

Strings

documentation

Function	Description
`string || string`	String concatenation
`string || non-string or non-string || string`	String concatenation with one non-string input
`char_length(string) or character_length(string)`	Number of characters in string
`lower(string)`	Convert string to lower case
`position(substring in string)`	Location of specified substring
`substring(string)`	Extract substring
`upper(string)`	Convert string to upper case
`length(string)`	Number of characters in string

One of the useful string operators is `||`, or the concatenation operators. It can be used with multiple strings, or non-string values

Skip to main content

Example: Using table `movies` from the `imdb` database, print: `"<title>" is <hours> hours long, and rated <rating> / 10.`. Round the hour to 1 decimal point.

```
%%sql

SELECT
    '"' || title || '"'
    || ' is ' || ROUND(runtime / 60., 1)
    || ' hours long, and rated '
    || rating || ' / 10.'
FROM
    movies
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
26058 rows affected.
```

Skip to main content

?column?
"Kate & Leopold" is 2.0 hours long, and rated 6.4 / 10.
"Mister 880" is 1.5 hours long, and rated 7.1 / 10.
"Black Hand" is 1.5 hours long, and rated 6.4 / 10.
"Francis" is 1.5 hours long, and rated 6.4 / 10.
"Orpheus" is 1.6 hours long, and rated 8 / 10.
"Stromboli" is 1.8 hours long, and rated 7.3 / 10.
"Woman in Hiding" is 1.5 hours long, and rated 6.9 / 10.
"Abbott and Costello in the Foreign Legion" is 1.3 hours long, and rated 6.6 / 10.
"Annie Get Your Gun" is 1.8 hours long, and rated 6.9 / 10.
"Armored Car Robbery" is 1.1 hours long, and rated 7 / 10.
"The Asphalt Jungle" is 1.9 hours long, and rated 7.9 / 10.
"Atom Man vs. Superman" is 4.2 hours long, and rated 7 / 10.
"Backfire" is 1.5 hours long, and rated 6.6 / 10.
"The Baron of Arizona" is 1.6 hours long, and rated 7 / 10.
"The Big Lift" is 2.0 hours long, and rated 6.5 / 10.
"The Black Rose" is 2.0 hours long, and rated 6.3 / 10.
"The Blue Lamp" is 1.4 hours long, and rated 6.9 / 10.
"Borderline" is 1.5 hours long, and rated 6.1 / 10.
"Born to Be Bad" is 1.6 hours long, and rated 6.7 / 10.
"Branded" is 1.7 hours long, and rated 6.7 / 10.

26058 rows, truncated to displaylimit of 20

We can use the `SUBSTRING(string FROM pos FOR num)` function to extract parts of a string value, starting with a particular position and continuing for a specified number of characters.

The function `SUBSTR(string, pos, num)` also exists in Postgres which is pretty similar to `SUBSTRING`, but with a different syntax.

Skip to main content

Example: Using table `movies` from the `imdb` database, print the `title` column in upper-case letters. Also, create two more columns with the first and last three characters of the title. Name these columns “First 3 characters” and “Last 3 characters”.

```
%%sql

SELECT
    UPPER(title),
    SUBSTRING(title FROM 1 FOR 3) AS "First 3 characters",
    SUBSTR(title, LENGTH(title) - 3, 3) AS "Last 3 characters"
FROM
    movies
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
26058 rows affected.
```

Skip to main content

upper	First 3 characters	Last 3 characters
KATE & LEOPOLD	Kat	pol
MISTER 880	Mis	88
BLACK HAND	Bla	Han
FRANCIS	Fra	nci
ORPHEUS	Orp	heu
STROMBOLI	Str	bol
WOMAN IN HIDING	Wom	din
ABBOTT AND COSTELLO IN THE FOREIGN LEGION	Abb	gio
ANNIE GET YOUR GUN	Ann	Gu
ARMORED CAR ROBBERY	Arm	ber
THE ASPHALT JUNGLE	The	ngl
ATOM MAN VS. SUPERMAN	Ato	rma
BACKFIRE	Bac	fir
THE BARON OF ARIZONA	The	zon
THE BIG LIFT	The	Lif
THE BLACK ROSE	The	Ros
THE BLUE LAMP	The	Lam
BORDERLINE	Bor	lin
BORN TO BE BAD	Bor	Ba
BRANDED	Bra	nde

26058 rows, truncated to displaylimit of 20

Skip to main content

Datetimes

In addition to `+`, `-`, `*`, and `/` operators, Postgres also provides useful functions for working with datetimes and intervals. Let's look at a few of those here:

```
%%sql

SELECT
    CURRENT_DATE,
    NOW(),
    CURRENT_TIMESTAMP(0),
    CURRENT_TIME(0),
    LOCALTIMESTAMP(0),
    LOCALTIME(2)
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1 rows affected.
```

current_date	now	current_timestamp	current_time	localtimestamp
2021-11-18	2021-11-18 07:39:44.472676-08:00	2021-11-18 07:39:44-08:00	07:39:44-08:00	2021-11-18 07:39:44

The argument to `CURRENT_TIMESTAMP()` and other functions above specifies the desired precision for the seconds field.

```
%%sql

SELECT
    CURRENT_TIME,
    LOCALTIME,
    LOCALTIMESTAMP
;
```

```
   postgresql://postgres:***@localhost:5432/
```

Skip to main content

```
1 rows affected.
```

current_time	localtime	localtimestamp
07:39:44.756441-08:00	07:39:44.756441	2021-11-18 07:39:44.756441

`NOW()` and `CURRENT_TIMESTAMP` are equivalent, with the latter being SQL-standard.

> Note that both of these functions/variables are timezone-aware.

```
%%sql

SELECT
    EXTRACT(hour FROM NOW()),
    EXTRACT(year FROM '2021-11-15 8:00:00'::TIMESTAMP)
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
1 rows affected.
```

extract	extract_1
7	2021

```
%%sql

SELECT
    age(NOW(), '1979-01-05'::TIMESTAMP)
;
```

```
   postgresql://postgres:***@localhost:5432/
```

Skip to main content

```
1 rows affected.
```

age
16007 days, 2:04:37.906168

The `age()` function formats the output nicely in `psql` and pgAdmin, but it unfortunately shows up as the number of days in `ipython-sql`.

There are a couple of functions for conversions between datetimes and intervals to and from strings:

```
%%sql

SELECT
    to_date('2021', 'YYYY'),
    to_date('05 Dec 2000', 'DD Mon YYYY'),
    to_timestamp('05 Dec 2021', 'DD Mon YYYY')
;
```

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

to_date	to_date_1	to_timestamp
2021-01-01	2000-12-05	2021-12-05 00:00:00-08:00

```
%%sql

SELECT
    to_char(current_timestamp, 'Day, DD  HH:MI'),
    to_char(interval '15h 2m 12s', 'HH12:MI:SS')
;
```

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

Skip to main content

to_char	to_char_1
Thursday , 17 02:04	03:02:12

For a full description of various datetime formatting functions and detailed string formatting patterns, see the documentation here.

Example: Using table `names` from the `imdb` database, retrieve the name and age of each alive person (according to the table!) in years using the `AGE()` function. You need to convert data types for this.

```
%%sql

SELECT
    name,
    EXTRACT(
        year FROM AGE(NOW(), to_date(birth_year::varchar, 'YYYY')))
        AS "Age"
FROM
    names
WHERE
    death_year IS NULL
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb
95083 rows affected.
```

Skip to main content

name	Age
Brigitte Bardot	88
Olivia de Havilland	106
Kirk Douglas	106
Sophia Loren	88
Raquel Welch	82
Li Gong	57
Armin Mueller-Stahl	92
Gérard Pirès	80
John Cleese	83
Brad Pitt	59
Woody Allen	87
Gillian Anderson	54
Jennifer Aniston	53
Patricia Arquette	54
Rowan Atkinson	67
Dan Aykroyd	70
Kevin Bacon	64
Fairuza Balk	48
Antonio Banderas	62
Adrienne Barbeau	77

95083 rows, truncated to displaylimit of 20

Use interval datatype (OPTIONAL)

We can easily do time arithmetic:

Skip to main content

SELECT '2h 50m'::INTERVAL, ' 5.5h'::INTERVAL + 3 * '14:00'::TIME, '2021-11-1'::DATE + '3 months 12 days'::INTERVAL, '2:00'::TIME + '18 hours 9 seconds'::INTERVAL, 14 * '1 day'::INTERVAL ;

Nulls

`NULLIF(value1, value2)` returns null if `value1` and `value2` are equal. This is helpful for replacing known values with nulls, or prevent, for example, division by zero.

```
%%sql

SELECT
    *,
    NULLIF(genre, 'drama')
FROM
    movie_genres
;
```

```
   postgresql://postgres:***@localhost:5432/
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
57633 rows affected.
```

Skip to main content

movie_id	genre	nullif
10035423	comedy	comedy
10035423	fantasy	fantasy
10035423	romance	romance
10042742	comedy	comedy
10042742	crime	crime
10042742	romance	romance
10041181	crime	crime
10041181	film-noir	film-noir
10041181	thriller	thriller
10041387	comedy	comedy
10041387	drama	None
10041387	family	family
10041719	drama	None
10041719	fantasy	fantasy
10041719	romance	romance
10041931	drama	None
10042052	crime	crime
10042052	drama	None
10042052	film-noir	film-noir
10042179	adventure	adventure

57633 rows, truncated to displaylimit of 20

Skip to main content

(optional) Postgres-specific functions

There are also a number of informative functions that are specific to Postgres. You can find a list of all of them here: link.

The one that I particularly find useful is `pg_typeof()` for when I want to make sure about the type of values after doing computations:

```
%%sql

SELECT
    pg_typeof(54 / 3.),
    pg_typeof(100 > 1)
;
```

```
 * postgresql://postgres:***@localhost:5432/imdb_dsci513
   postgresql://postgres:***@localhost:5432/world_dsci513
1 rows affected.
```

pg_typeof	pg_typeof_1
numeric	boolean