Lecture 3: Summarizing & joining tables

Contents

DSCI 513

Databases & Data Retrieval

Lecture outline

- Aggregations
- Grouping
- Joins

Well, things have finally gotten serious with SQL and relational databases! In this lecture, we will learn how to do more advanced operations to gain more insight into the data in the tables of a database. Furthermore, we will explore how to connect the related data in multiple tables together through SQL joins. **It is through using joins in relational databases that we can benefit from the true power of these databases**.

First things first, let's connect to our database:

Skip to main content

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

```
import json
import urllib.parse

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

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

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

Aggregations

So far in the course, we have seen many functions for various purposes. For example, `ROUND()` and `SQRT()` for math operations, `CHAR_LENGTH` and `SUBSTR()` for manipulating strings, or `EXTRACT()` and `to_date()` for working with datetimes. As you may have noticed, these functions produce an individual output for each and every row of a column in an element-wise manner (remember vectorized operations in NumPy?).

There is also a small class of useful functions in SQL called **aggregation** functions, which operate on groups of rows and summarize the data stored in those rows in the form of a single value. Here is a list of standard aggregation functions in SQL:

Function	What it computes
`COUNT(*)`	Count of all rows in a table
`COUNT()`	Count of non-null rows of a column
`MIN()`	Minimum value in a column
`MAX()`	Maximum value in a column
`AVG()`	Average of values in a column
`SUM()`	Total sum of values in a column

A couple of points to remember:

- Except for `COUNT(*)`, all aggregation functions ignore `NULL`s
- In addition to numbers, `MIN()` and `MAX()` also work with strings.

Example: Find the population of the world according to the `country` table in the `world` database.

```
%%sql

SELECT
    SUM(population)
FROM
    country
;
```

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

sum
6078749450

Skip to main content

There are a few things that we need to remember when using aggregation functions:

- It is valid to have multiple aggregations in a SQL query, but it is NOT possible have both aggregations and regular columns in a single query:

```
-- This is CORRECT:
SELECT
    AVG(lifeexpectancy), SUM(population)
FROM
    country
WHERE
    continent = 'North America'
;

-- This is WRONG:
SELECT
    AVG(lifeexpectancy), name
FROM
    country
WHERE
    continent = 'North America'
;
```

There is only one exception to the latter rule, and that's when we have a `GROUP BY` clause (we'll learn about that in a bit).

- An aggregation function CANNOT be used in the `WHERE` clause.

For example, we can't find the name of countries with above-average populations using the following query:

```
-- This is WRONG:
SELECT
    name
FROM
    country
```

```
    population > AVG(population)
;
```

It is, of course, possible to write a query to answer the above question, but we have to wait until we learn about subqueries in a later lecture.

Postgres-specific aggregations (OPTIONAL)

In addition to the standard SQL aggregation functions, Postgres also provides a number of functions of the same kind which can be useful for some statistical calculations (find a comprehensive list in the documentations here). Here is a few examples of Postgres-specific aggregations:

Function	What it computes
`stddev_pop()`	Population standard deviation
`stddev_samp()`	Sample standard deviation
`regr_r2(X, Y)`	Coefficient of determination for linear regression between `X` and `Y`
`regr_slope(X, Y)`	Slope of the regression line of `X` and `Y`
`regr_intercept(X, Y)`	Intercept of the regression line of `X` and `Y`

Example: Compute the average ± sample standard deviation of the population of cities located in the US using `city` table. Write your query such that its output looks like this: `Average ± STDEV population of US cities = <average_population> ± <stdev_population>`

```
SELECT
    'Average ± STDEV population of cities in the US = ' ||
    AVG(population)::INT || ' ± ' || stddev_samp(population)::INT
FROM
    city
WHERE
    countrycode = 'USA'
;
```

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

?column?
Average ± STDEV population of cities in the US = 286955 ± 586583

Grouping

If we divide a table into groups of rows based on values of one or more columns, that is called grouping.

For example, in the `country` table of the `world` database, we find several countries located in the same continent. In this situation, we can group the rows in our `country` table based on the values in the `continent` column. In this way, we would end up with bunch of “sub-tables”: A sub-table for all rows where `continent = 'Asia'`, another sub-table for all rows where `continent = 'Europe'`, and so on.

The formal syntax of the grouping operation in SQL looks like this:

```
SELECT
    grouping_columns, aggregated_columns
FROM
    table1
WHERE
    condition
GROUP BY
    grouping_columns
ORDER BY
```

Skip to main content

Typically, it is not the sub-tables themselves that we're interested in, but some sort of summary statistics: For example, we might want to know the average population for each continent, i.e. for each sub-table or group. In order to do this, we can use aggregation functions that learned about them in the previous section. The question of **"what is the average population of countries in each continent"** can be asked in SQL terms as follows:

```
%%sql

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

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

continent	avg
Asia	72647562.745098039216
South America	24698571.428571428571
North America	13053864.864864864865
Oceania	1085755.357142857143
Antarctica	0E-20
Africa	13525431.034482758621
Europe	15871186.956521739130

Important points:

- If your SQL query involves grouping as well as filtering with `WHERE` and sorting with `ORDER BY`, the `GROUP BY` clause MUST appear between `WHERE` and `ORDER BY`.
- There can't be any non-aggregated column in a grouping query, except for the columns which are used for grouping (remember the exception I talked about with aggregation functions?). In other words, a non-aggregated column in the `SELECT` clause MUST appear in the `GROUP BY` clause as well.
- If there are null values in the grouping column, there will be a separate group for null

Example: Write a query to return the average and maximum population of cities in the `city` table for China, India, Canada, US, Australia, and Russia.

Show the results for each country using the corresponding country code, and order groups alphabetically in ascending order.

```
%%sql

SELECT
    countrycode, AVG(population), MAX(population)
FROM
    city
WHERE
    countrycode IN ('CHN', 'IND', 'CAN', 'USA', 'AUS', 'RUS')
GROUP BY
    countrycode
ORDER BY
    countrycode
;
```

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

countrycode	avg	max
AUS	808119.000000000000	3276207
CAN	258649.795918367347	1016376
CHN	484720.699724517906	9696300
IND	361579.255131964809	10500000
RUS	365876.719576719577	8389200
USA	286955.379562043796	8008278

Skip to main content

Filtering revisited: the `HAVING` clause

So far, we have used the `WHERE` clause to filter rows. However, I mentioned before that aggregation functions cannot be used inside a `WHERE` clause to enforce a condition on the returned grouped rows.

There is another reserved keyword, `HAVING`, for when we need to do filtering using aggregated values **after grouping rows**. The syntax is as follows (order is important!):

```
SELECT
    grouping_columns, aggregated_columns
FROM
    table1
[WHERE
    condition]
GROUP BY
    grouping_columns
HAVING
    group_condition
[ORDER BY
    grouping_columns]
```

Important note:

- `WHERE` filters rows **before** grouping
- `HAVING` filters rows **after** (or resulting from) grouping

Example:

Write a query to return the average and maximum population of cities for countries that have at least 60 cities listed in the `city` table.

Show the results for each country using the corresponding country code, and order groups

Skip to main content

values to integer type.

```
%%sql

SELECT
    countrycode,
    AVG(population)::int,
    MAX(population)::int,
    COUNT(population) AS city_count
FROM
    city
GROUP BY
    countrycode
HAVING
    COUNT(*) > 60
ORDER BY
    city_count DESC
;
```

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

countrycode	avg	max	city_count
CHN	484721	9696300	363
IND	361579	10500000	341
USA	286955	8008278	274
BRA	343507	9968485	250
JPN	314375	7980230	248
RUS	365877	8389200	189
MEX	345390	8591309	173
PHL	227462	2173831	136
DEU	282209	3386667	93
IDN	441008	9604900	85
GBR	276996	7285000	81
KOR	557141	9981619	70
IRN	388552	6758845	67
NGA	271358	1518000	64
TUR	456888	8787958	62

Note that just like with the `WHERE` clause, the expression used for filtering with `HAVING` does not necessarily need to appear in the `SELECT` clause (why?).

For instance, the `HAVING` clause will still do its job even if `COUNT(population)` is omitted from the `SELECT` clause:

```
%%sql

SELECT
    countrycode,
    AVG(population)::INT,
    MAX(population)::INT
FROM
    city
GROUP BY
    countrycode
HAVING
    COUNT(*) > 60
ORDER BY
```

Skip to main content

```
    COUNT(*) DESC
;
```

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

countrycode	avg	max
CHN	484721	9696300
IND	361579	10500000
USA	286955	8008278
BRA	343507	9968485
JPN	314375	7980230
RUS	365877	8389200
MEX	345390	8591309
PHL	227462	2173831
DEU	282209	3386667
IDN	441008	9604900
GBR	276996	7285000
KOR	557141	9981619
IRN	388552	6758845
NGA	271358	1518000
TUR	456888	8787958

A `GROUP BY` clause can be considered as equivalent to using `DISTINCT` if no aggregate functions are used:

```
%%sql

SELECT
```

```
    country
GROUP BY
    continent
;
```

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

continent
Asia
South America
North America
Oceania
Antarctica
Africa
Europe

```
%%sql

SELECT
    DISTINCT continent
FROM
    country
;
```

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

continent
Asia
South America
North America
Oceania
Antarctica
Africa

Skip to main content

Note: Neither `GROUP BY` nor `DISTINCT` ignore null values.

For example, here you'll see `None` (Jupyter Notebook's way of showing `NULL`s) appearing at the top when we group rows by `headofstate`, which contain null values:

```
%%sql

SELECT
    headofstate
FROM
    country
GROUP BY
    headofstate
ORDER BY
    headofstate DESC
LIMIT
    5
;
```

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

headofstate
None
Ólafur Ragnar Grímsson
Émile Lahoud
tipe Mesic
kenraali Than Shwe

Once again remember that:

As long as aggregated, columns appearing in the `HAVING` clause don't necessarily need to be present in the `SELECT` clause. Here, we're retrieving continents having at least 40 countries:

```
SELECT
    continent
FROM
    country
GROUP BY
    continent
HAVING
    COUNT(name) >= 40
;
```

Note that we didn't use the column `name` in `SELECT`, but we still had access to it.

Question: Why are we able to access columns in the `HAVING` clause that are not listed in the `SELECT` clause?

Multi-level grouping

The `GROUP BY` clause can accommodate more than one column to construct multi-level groups. For example, we can group the rows in the `country` table of the `world` database first based on `continent` and then based on `region`, all in one go:

```
%%sql

SELECT
    continent, region, AVG(population)::INT
FROM
    country
GROUP BY
    continent, region
ORDER BY
    continent, region
;
```

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

Skip to main content

continent	region	avg
Africa	Central Africa	10628000
Africa	Eastern Africa	12349950
Africa	Northern Africa	24752286
Africa	Southern Africa	9377200
Africa	Western Africa	13039529
Antarctica	Antarctica	0
Asia	Eastern Asia	188416000
Asia	Middle East	10465594
Asia	Southeast Asia	47140091
Asia	Southern and Central Asia	106484000
Europe	Baltic Countries	2520633
Europe	British Islands	31699250
Europe	Eastern Europe	30702600
Europe	Nordic Countries	3452343
Europe	Southern Europe	9644947
Europe	Western Europe	20360844
North America	Caribbean	1589167
North America	Central America	16902625
North America	North America	61926400
Oceania	Australia and New Zealand	4550620
Oceania	Melanesia	1294400
Oceania	Micronesia	77571
Oceania	Micronesia/Caribbean	0
Oceania	Polynesia	63305
South America	South America	24698571

Skip to main content

Joins

Joins are probably the most fundamentally important operation in relational databases. The reason is that the whole idea of such databases is that data can be broken down into various tables that are related to each other, and can be joined together whenever related information from multiple tables is required. Consider the following query as an example:

Example: Write a query that returns the name of all countries along with their corresponding continents and their cities.

As we've been working with the `world` database, we immediately notice that information about countries and cities are stored in two different tables, so we should somehow combine or "join" the data from the two tables.

The syntax for a joining tables in SQL is as follows:

```
SELECT
    columns
FROM
    left_table
join_type
    right_table
ON
    join_condition
WHERE
    row_filter
GROUP BY
    columns
HAVING
    group_filter
ORDER BY
    columns
;
```

In this section, we'll learn how to do a join to answer the question we posed for the `world`, but I prefer to use a smaller database to demonstrate various joining methods first, and then use our larger databases.

First, let's create a new database called `mds` on the local host (i.e. our own computer) and connect to it:

```
%sql CREATE DATABASE mds;
```

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

```
[]
```

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

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

The following cell creates two tables with the names `instructor` and `instructor_course` with the information about MDS instructors and courses they teach. **Don’t worry about the content of this cell!** We will learn how to create tables in the next lecture. For now, just run the cell to create and populate the tables:

```
%%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'),
```

Skip to main content

```
    (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,
    instructor_id INTEGER,
    course TEXT,
    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'),
    (3, 'MDS-V')
.
```

Skip to main content

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

```
[]
```

Awesome! Let’s take a look at the first two tables:

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

```
 * 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 SELECT * FROM instructor_course;
```

Skip to main content

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

id	instructor_id	course	enrollment	begins
1	8	Statistical Inference and Computation I	125	2021-10-01
2	8	Regression II	102	2022-02-05
3	1	Descriptive Statistics and Probability	79	2021-09-10
4	1	Algorithms and Data Structures	25	2021-10-01
5	3	Algorithms and Data Structures	25	2021-10-01
6	3	Python Programming	133	2021-09-07
7	3	Databases & Data Retrieval	118	2021-11-16
8	6	Visualization I	155	2021-10-01
9	6	Privacy, Ethics & Security	148	2022-03-01
10	2	Programming for Data Manipulation	160	2021-09-08
11	7	Data Science Workflows	98	2021-09-15
12	2	Data Science Workflows	98	2021-09-15
13	12	Web & Cloud Computing	78	2022-02-10
14	10	Introduction to Optimization	None	2022-09-01
15	9	Parallel Computing	None	2023-01-10
16	13	Natural Language Processing	None	2023-09-10

Cross join

A cross join is the simplest way to join two tables:

by cross-joining tables A and B, we match each every row from table A with every row from table B!

Skip to main content

In other words, it returns all combinations of rows from table A and table B. This type of join is also sometimes called *the Cartesian product* of two relations or tables:

```
%config SqlMagic.displaylimit = 200
```

```
%%sql

SELECT
    *
FROM
    instructor
CROSS JOIN
    instructor_course
;
```

How to deal with ambiguous column names

Now suppose that we want to return only the names of instructors and their IDs from the `instructor` table, and names of courses and their IDs from the `course` table. Since there is a column named `id` in both tables, we cannot use `id` in the `SELECT` clause, because it would be ambiguous.

In this situation, we should either prepend the column name by the full name of its parent table (e.g. `instructor.id`), or we can create table aliases using the keyword `AS` (just like we did before with columns) and prepend the column name with the parent table alias. A table name followed by a dot and the name of a column is called a ***qualified name***.

Here is an example of using qualified names for ambiguous column names:

```
%%sql

SELECT
    name, i.id, course, ic.id
FROM
    instructor AS i
CROSS JOIN
    instructor_course AS ic
```

Skip to main content

```
LIMIT 10
;
```

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

name	id	course	id_1
Mike	1	Statistical Inference and Computation I	1
Tiffany	2	Statistical Inference and Computation I	1
Arman	3	Statistical Inference and Computation I	1
Varada	4	Statistical Inference and Computation I	1
Quan	5	Statistical Inference and Computation I	1
Joel	6	Statistical Inference and Computation I	1
Florencia	7	Statistical Inference and Computation I	1
Alexi	8	Statistical Inference and Computation I	1
Vincenzo	15	Statistical Inference and Computation I	1
Gittu	19	Statistical Inference and Computation I	1

- The keyword `AS` can be dropped
- Table aliases only exist during the execution of a statement
- Using table aliases is a great way to reduce clutter in SQL join statements
- Once you create an alias for a table, you should only use the alias to refer to that table in the statement. For example, the following query would throw an error:

```
-- This is WRONG
SELECT
    instructor.name, instructor.id, course, ic.id
FROM
    instructor AS i
CROSS JOIN
    instructor_course AS ic
;
```

- When you retrieve data from multiple tables, you can still use `*` to return all columns of a particular table. The only difference is that you should prepend it with the

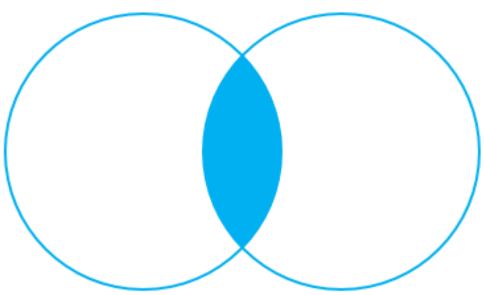
Skip to main content

`instructor` and just the `course` column of the table `instructor_course` in a cross join with the following query:

```
SELECT
    i.*, ic.course
FROM
    instructor AS i
CROSS JOIN
    instructor_course AS ic
;
```

Inner join

Except for a cross join, all other types of joins use a condition using the `ON` keyword to figure out which rows from the two tables to pair up. An inner join is a type of join that only returns the matching rows from the left and right tables. The image below (source) shows Venn diagram of an inner join:



INNER JOIN

For example, in our `instructor` table there are some instructors who are assigned one or more courses in the `instructor_course` table, some who are not. Similarly, there are courses in the `instructor_course` table that have an instructor, and some that don't have an instructor yet. With an inner join based on `instructor.id` and `instructor_course.id` columns, we would retrieve matching rows, meaning that only instructors are retrieved that have one or more assigned courses, and vice versa:

```
%%sql
```

Skip to main content

```
    name, i.id, ic.instructor_id, course
FROM
    instructor AS i
INNER JOIN
    instructor_course AS ic
ON
    i.id = ic.instructor_id
;
```

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

name	id	instructor_id	course
Alexi	8	8	Statistical Inference and Computation I
Alexi	8	8	Regression II
Mike	1	1	Descriptive Statistics and Probability
Mike	1	1	Algorithms and Data Structures
Arman	3	3	Algorithms and Data Structures
Arman	3	3	Python Programming
Arman	3	3	Databases & Data Retrieval
Joel	6	6	Visualization I
Joel	6	6	Privacy, Ethics & Security
Tiffany	2	2	Programming for Data Manipulation
Florencia	7	7	Data Science Workflows
Tiffany	2	2	Data Science Workflows

In the above returned table, “Quan” and “Varada” are missing as instructors since they are not yet assigned any courses. Also, the courses “Web & Cloud Computing”, “Parallel Computing”, and “Introduction to Optimization” are missing, since there not yet any instructors assigned for these courses.

> **Note:** The `INNER` keyword is optional.

Skip to main content

Self join

Sometimes we want to compare a table to itself. For example, we may want to know which paris of instructors in the `instructors` table are from the same department. In order to find out, we need to compare the values in the `department` column of each row to all other rows to find matches:

```
%%sql

SELECT
    i1.name, i1.department, i2.department, i2.name
FROM
    instructor i1
JOIN
    instructor i2
ON
    i1.department = i2.department
    AND
    i1.id <> i2.id
;
```

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

name	department	department_1	name_1
Mike	Computer Science	Computer Science	Jessica
Mike	Computer Science	Computer Science	Varada
Varada	Computer Science	Computer Science	Jessica
Varada	Computer Science	Computer Science	Mike
Joel	Biomedical Engineering	Biomedical Engineering	Gittu
Alexi	Statistics	Statistics	Vincenzo
Vincenzo	Statistics	Statistics	Alexi
Gittu	Biomedical Engineering	Biomedical Engineering	Joel
Jessica	Computer Science	Computer Science	Varada
Jessica	Computer Science	Computer Science	Mike

The `i1.id <> i2.id` join condition ensures that a row does not match itself.

Natural join

For joins involving a join condition, e.g. inner or self joins, we have so far explicitly specified the matching condition. In a situation that columns in different tables have the same name and we want to simply match rows with similar values **in all similarly named columns**, we can do a **natural join** using the keywords `NATURAL JOIN`. For example, the `id` column in the `course_cohort` refers to the `id` column let's find which courses are offered for which cohorts using a natural join:

```
%sql SELECT * FROM course_cohort;
```

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

Skip to main content

id	cohort
13	MDS-CL
8	MDS-CL
1	MDS-CL
3	MDS-CL
1	MDS-V
9	MDS-V
3	MDS-V

```
%%sql

SELECT
    ic.course, cc.cohort
FROM
    instructor_course ic
NATURAL JOIN
    course_cohort cc
;
```

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

course	cohort
Web & Cloud Computing	MDS-CL
Visualization I	MDS-CL
Statistical Inference and Computation I	MDS-CL
Descriptive Statistics and Probability	MDS-CL
Statistical Inference and Computation I	MDS-V
Privacy, Ethics & Security	MDS-V
Descriptive Statistics and Probability	MDS-V

If there are no matching columns in the two tables, `NATURAL JOIN` acts like `JOIN ... ON TRUE` and results in a cross-product join between the participating tables.

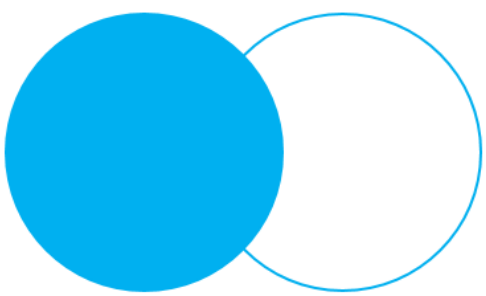
Outer joins

An outer join is a type of join that returns all the rows from one or both of the tables that takes part in the joining. Outer joins are useful in questions that involve missing values.

Left outer join

In the joining process, the first table from which data is retrieved using `SELECT` is called the **left** table, and the table that is joined onto that is called the **right** table. In other words, the first table that appears in the query is the left table (table on the left of the query), and the one appearing later is the right table (table on the right of the query).

A left outer join is a type of join that returns all rows from the left table (matching or not), in addition to the matching rows from both tables. The non-matching rows from the left table are assigned null values in the columns that belong to the right table. This is schematically shown in the diagram below (source):



LEFT OUTER JOIN

For example, in the inner join example, instructors who don't teach any course are not returned by the join operation. Let's say we want to retrieve a list of all instructors and the courses they teach, as well as those who don't teach any courses:

```
%%sql

SELECT
    name, i.id, ic.instructor_id, course
```

Skip to main content

```
LEFT OUTER JOIN
    instructor_course AS ic
ON
    i.id = ic.instructor_id
;
```

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

name	id	instructor_id	course
Alexi	8	8	Statistical Inference and Computation I
Alexi	8	8	Regression II
Mike	1	1	Descriptive Statistics and Probability
Mike	1	1	Algorithms and Data Structures
Arman	3	3	Algorithms and Data Structures
Arman	3	3	Python Programming
Arman	3	3	Databases & Data Retrieval
Joel	6	6	Visualization I
Joel	6	6	Privacy, Ethics & Security
Tiffany	2	2	Programming for Data Manipulation
Florencia	7	7	Data Science Workflows
Tiffany	2	2	Data Science Workflows
Vincenzo	15	None	None
Quan	5	None	None
Gittu	19	None	None
Jessica	16	None	None
Varada	4	None	None

> **Note:** The keyword `OUTER` is optional.

How can this be helpful? As an example, we can return the name of instructors who don't

Skip to main content

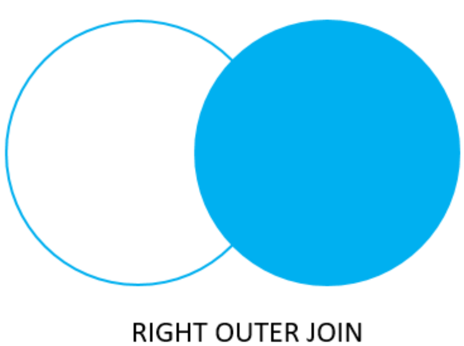
```
%%sql

SELECT
    name
FROM
    instructor AS i
LEFT JOIN
    instructor_course AS ic
ON
    i.id = ic.instructor_id
WHERE
    ic.course IS NULL
;
```

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

name
Vincenzo
Quan
Gittu
Jessica
Varada

Right outer join

A right join acts exactly in the same way as a left join, except that it keeps all rows from the right table and only the matching ones from the left table. The diagram below demonstrates a right join schematically (source):



Skip to main content

Let’s retrieve a list of all courses and their appointed instructors, as well as those courses without an instructor:

```
%%sql

SELECT
    name, i.id, ic.instructor_id, course
FROM
    instructor AS i
RIGHT OUTER JOIN
    instructor_course AS ic
ON
    i.id = ic.instructor_id
;
```

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

name	id	instructor_id	course
Alexi	8	8	Statistical Inference and Computation I
Alexi	8	8	Regression II
Mike	1	1	Descriptive Statistics and Probability
Mike	1	1	Algorithms and Data Structures
Arman	3	3	Algorithms and Data Structures
Arman	3	3	Python Programming
Arman	3	3	Databases & Data Retrieval
Joel	6	6	Visualization I
Joel	6	6	Privacy, Ethics & Security
Tiffany	2	2	Programming for Data Manipulation
Florencia	7	7	Data Science Workflows
Tiffany	2	2	Data Science Workflows
None	None	12	Web & Cloud Computing
None	None	10	Introduction to Optimization
None	None	9	Parallel Computing

Skip to main content

Now, let's find out which courses do not have an appointed instructor yet:

```
%%sql

SELECT
    course
FROM
    instructor AS i
RIGHT OUTER JOIN
    instructor_course AS ic
ON
    i.id = ic.instructor_id
WHERE
    i.id IS NULL
;
```

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

course
Web & Cloud Computing
Introduction to Optimization
Parallel Computing
Natural Language Processing

Full outer join

A full outer join is the combination of a left and right join: it retrieves **matching and non-matching** rows from **both** tables. Take a look at the schematic diagram of a full outer join (source):

Skip to main content

Let's do a full outer join between the `instructor` and `instructor_course` tables to retrieve all instructors and courses:

```
%%sql

SELECT
    name, i.id, ic.instructor_id, course
FROM
    instructor AS i
FULL OUTER JOIN
    instructor_course AS ic
ON
    i.id = ic.instructor_id
;
```

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

name	id	instructor_id	course
Alexi	8	8	Statistical Inference and Computation I
Alexi	8	8	Regression II
Mike	1	1	Descriptive Statistics and Probability
Mike	1	1	Algorithms and Data Structures
Arman	3	3	Algorithms and Data Structures
Arman	3	3	Python Programming
Arman	3	3	Databases & Data Retrieval
Joel	6	6	Visualization I
Joel	6	6	Privacy, Ethics & Security
Tiffany	2	2	Programming for Data Manipulation
Florencia	7	7	Data Science Workflows
Tiffany	2	2	Data Science Workflows
None	None	12	Web & Cloud Computing
None	None	10	Introduction to Optimization
None	None	9	Parallel Computing
None	None	13	Natural Language Processing
Vincenzo	15	None	None
Quan	5	None	None
Gittu	19	None	None
Jessica	16	None	None
Varada	4	None	None

We can now write a query to find instructors who are free to teach a course, and courses that need an instructor:

```
%%sql

SELECT
    name, course
FROM
```

```
    instructor_course AS ic
ON
    i.id = ic.instructor_id
WHERE
    i.name IS NULL
    OR
    ic.course IS NULL
;
```

Question: What's the difference between a cross join and a full outer join?

`WHERE` joins (OPTIONAL)

Prior to the SQL-92 standard, there were no reserved keywords (e.g `JOIN`, `CROSS JOIN`, `LEFT JOIN`) for joining tables in SQL. Instead, joins used to be done by pulling data from two (or more) tables directly in the `FROM` clause, and filtering for the desired rows using the `WHERE` clause, which is why this type of join is known as a `WHERE` join.

The following syntax returns the result of a cross join of `table1` and `table2`:

```
SELECT
    table1_columns, table2_columns
FROM
    table1, table2
```

In order to create an **inner** join, for example, we can express the join condition in the `WHERE` clause:

Skip to main content

```
SELECT
    table1_columns, table2_columns
FROM
    table1, table2
WHERE
    table1.column1 = table2.column1
```

Let's give this a try using the tables we created in this section. First, I'll do a simple cross join of `instructor` and `instructor_course`:

```
%%sql

SELECT
    i.*, ic.*
FROM
    instructor i,
    instructor_course ic
;
```

To turn it into an inner join on two related columns, I can use the `WHERE` clause to check for equality of values in those columns:

```
%%sql

SELECT
    i.*, ic.*
FROM
    instructor i,
    instructor_course ic
WHERE
    i.id = ic.instructor_id
;
```

Note that there is not a predefined way in SQL to perform an outer join using this older syntax. Each RDBMS provides its own specific syntax for doing outer joins. This is why using the newer `OUTER JOIN` keywords is almost always preferred, as it is much cleaner and more explicit.

`USING` keyword (OPTIONAL)

If there are multiple columns with the same names in both participating tables of a join, we can take advantage of the keyword `USING` and write:

```
SELECT
    ...
FROM
    left_table t1
JOIN
    right_table t2
USING
    (column1, column2, ...)
```

instead of

```
SELECT
    ...
FROM
    left_table t1
JOIN
    right_table t2
ON
    t1.column1 = t2.column1
    AND
    t1.column2 = t2.column2
    AND
    .
    .
    .
```