# SQLWorkshop — Tutorial D1

version **#v1.0.6**


**Assistant C/UNIX 2021** <assistants@tickets.assistants.epita.fr>

## Copyright



## Contents



*https://intra.assistants.epita.fr

# 1 Welcome to PostgreSQL

During this tutorial, you will use the DBMS (DataBase Management System) PostgreSQL 12.1.

A Database Management System is a system software for creating and managing databases.

## 1.1 Why use PostgreSQL

Some of you might not know PostgreSQL yet. Several reasons made us choose to use it over another SQL database management system during this month. First, PostgreSQL is free to use and open-source. Second, its approach is to rigorously implement what the SQL standard prescribes. Knowing that this month's goal is not to learn PostgreSQL, but to understand how SQL works, this argument alone is enough for PostgreSQL to be a good choice.

## 1.2 Initialize the server

First, let's configure a PostgreSQL-specific environment variable:

```
42sh$ echo 'export PGDATA="$HOME/postgres_data"' >> ~/.bashrc
42sh$ echo 'export PGHOST="/tmp"' >> ~/.bashrc
42sh$ source ~/.bashrc
```

This first line adds a `PGDATA` environment variable to your bashrc, containing the location of your choice for storing `postgres` data. The second specifies the host name of the machine on which the server will run. As the value begins with a slash, it will be used as the directory containing a socket on which postgres will listen.

Then, let's initialize a new PostgreSQL *database cluster*. It will generate default databases and configurations in the `$PGDATA` directory.

```
42sh$ initdb --locale "$LANG" -E UTF8
```

**Be careful!**

If you experience a permission denied, you may need to restore your rights using `chmod 755 ~`

## 1.3 Launch the server

To launch your `PostgreSQL` server, you have to execute the following command. It starts a server using the mandatory path to its data area - provided through the definition of `PGDATA`. `-k` specifies the directory where the socket will be created. Thus, we want it to have the same value as `PGHOST`.

```
42sh$ postgres -k "$PGHOST"
```

### 1.4 Create a database

Your server is up and running! PostgreSQL offers an interactive shell that connects to this server, and serves as a front-end for your operations on databases:

```
42sh$ psql postgres
```

`psql` takes the name of the database you want to connect to. Here, we choose the default database, `postgres`.

These commands will allow you to create and own a database named `tutorial_1`:

```
-- Give yourself all the rights
-- If you are not on the PIE, <login> should be your username
postgres=# ALTER ROLE "<login>" SUPERUSER;

-- Create a database named tutorial_1
postgres=# CREATE DATABASE tutorial_1 OWNER <login>;

-- Exit PostgreSQL
postgres=# \q
```

You can then connect to your new database.

### 1.5 Internal commands

Here is how to launch the shell using your database:

```
# If you are not on the PIE <login> should be your username
42sh$ psql -d tutorial_1 -U <login>
# or simply
42sh$ psql tutorial_1
```

You can type two kinds of commands in the PostgreSQL interactive terminal:

- Standard SQL commands
- PostgreSQL-specific commands

In the psql shell, you can call internal commands which are not SQL commands. These commands all start with a backslash, and the most useful are:

- `\?`: The help. Displays all available postgres commands
- `\h`: Displays the list of all SQL commands
- `\h cmd`: Displays the documentation of an SQL command
- `\i file`: Executes SQL commands from a file
- `\d`: Lists all tables, views and sequences of your database
- `\dt`: Lists all tables
- `\dv`: Lists all views
- `\d elt`: Displays details about a particular object

- `\q`: Exits psql

In this tutorial, you will learn the basics of database management. We provide a script that creates tables and fills them, for you to focus on managing commands. Do not hesitate to check your work regularly using PostgreSQL visualization commands.

# 2 Data selection

## 2.1 Database filling

A database is composed of multiple tables, that will contain some data. You will learn how to create, delete or update such tables on the third tutorial. Thus, for today and tomorrow, we give you a file that creates those for you. Use the `\i <file>` command to execute `script.sql` in order to create the database. This file also fills its tables with some data.

We now have a filled database. Feel free to use `\d` to display the tables and `\d <table_name>` to see the details of a specific table.

## 2.2 SELECT

To see the content of a specific table, we are going to use the `SELECT` statement as following:

```
SELECT * FROM table_name;
```

The `*` is used for `all`. In this case, you are going to see all the columns of the given table.

### 2.2.1 Example

```
SELECT * FROM can;
```

```
 id |     name     | capacity_cl
----+--------------+-------------
  1 | Coke         |          25
  2 | Oasis        |          25
  3 | Diet Coke    |          25
  4 | Orangina     |          25
  5 | Sprite       |          25
  6 | Pepsi        |          25
  7 | Ice Tea      |          25
  8 | Water        |          50
  9 | Fanta        |          25
 10 | Orange Juice |          33
(10 rows)
```

## 2.3 Filter data

If you do not want to see all the columns of a table (or if you want to see the columns in a specific order), you can specify the columns as below:

```
SELECT column_1, column_2, column_3 FROM table_name;
```

You can also give a temporary name to a result table or column to format the requests results thanks to the AS keyword after the desired element.

```
SELECT column_1 AS col1 FROM table_name;
```

### 2.3.1 Example

```
SELECT name, capacity_cl AS capacity FROM can;
```

```
     name      | capacity
---------------+----------
 Coke          |       25
 Oasis         |       25
 Diet Coke     |       25
 Orangina      |       25
 Sprite        |       25
 Pepsi         |       25
 Ice Tea       |       25
 Water         |       50
 Fanta         |       25
 Orange Juice  |       33
(10 rows)
```

## 2.4 WHERE

You can filter the result of the SELECT by using the WHERE statement. For example, to select one can, you can write:

```
SELECT name, capacity_cl AS capacity FROM can
WHERE id = 8;
```

```
 name  | capacity
-------+----------
 Water |       50
(1 row)
```

If you want your selection to match more than one condition, you can add them by separating them with OR or AND.

## 2.5 Order results

You can sort your results thanks to the `ORDER BY` statement. By default, the sort is ascendant (increasing, alphabetic, ...). You can invert this order by adding `DESC` after the column name.

### 2.5.1 Example

```
SELECT firstname, lastname
FROM student
ORDER BY lastname, firstname;
```

```
 firstname  |  lastname
------------+-------------
 Anne       | Aghavni
 André      | Alphonsine
 Philomène  | Arnaud
 Paul       | Batrien
 Guy        | Bedros
 Claire     | Billy
 Tiphaine   | Cosée
 Aimé       | Cyrille
 Noémi      | Danièle
 Solène     | Dominique
 Anas       | El Halouani
 Aurélie    | Emericé
 Valentin   | Firmin
 Hélène     | Geinaux
 Eugène     | Gevorg
 Brice      | Guyrée
 Corentin   | Joel
 Paul       | Khuat-Duy
 Marjolaine | Léo
 Isidore    | Marcellin
 Léonie     | Muriel
 PAUL       | Nazare
 Arnaud     | Pauline
 Joseph     | Prosper
 Cédric     | Roch
 Hadrien    | Roch
 Ulysse     | Rouben
 Alexandre  | Théophile
 Noé        | Toros
 Théophane  | Vié
            |
(31 rows)
```

## 2.6 Control the number of results

LIMIT and OFFSET are two keywords that allow you to control the number of results you are going to see. In a request, they must be used after WHERE and ORDER BY.

```
SELECT * FROM table_name
[LIMIT { number | ALL}]
[OFFSET number];
```

LIMIT is used, as its name suggests, to limit the maximum number of results found by the request. OFFSET is used to skip a precise number of lines which are not going to be displayed in the results. If LIMIT and OFFSET are used in the same request, OFFSET lines are skipped before LIMIT applies.

## 2.7 Exercises

### 2.7.1 Request 1

Display all the can names, ordered in alphabetical order, in one column named “can”.

### 2.7.2 Request 2

Display all the cans whose capacity is greater than 30cl.

### 2.7.3 Request 3

Display the logins of the assistants in the student table in alphabetical order.

### 2.7.4 Request 4

Display the first names of the students in alphabetical order limiting to 10.

### 2.7.5 Request 5

Display 10 student last names in alphabetical order, beginning from the third.

## 2.8 DISTINCT

Sometimes, you will need to keep only one element of each row. It can be to remove redundancy or grouping elements. To do so, you can use the DISTINCT clause. It is used with the SELECT clause to remove duplicated value from a result set. The DISTINCT clause keeps one row for each group of duplicates.

```
SELECT DISTINCT column_name FROM table_name;
```

Here is an example below:

```
SELECT DISTINCT price
FROM shop_can;
```

```
 price
-------
 1
 1,1
 1,2
```

If you specify multiple columns, the DISTINCT clause will evaluate the duplicate based on the combination of values of these columns.

# 3 Data management

You are provided with a `sql-d1-movies-schema.sql` and a `sql-d1-movies-data.sql`. Execute those files in your interactive shell, in this order. Now that you have your tables ready in your database, the next step would be to know how to manage the data.

## 3.1 Insertion

Inserting data into tables can be done thanks to the `INSERT INTO` statement:

```
INSERT INTO table_name (column_names)
VALUES  (values);
```

There are things to know about insertions:

- The serial type is an integer which auto-increments itself for each addition in the database. So, when the type of one column is serial and if you have decided to write the column names, you do not need to give a value for this column. If you do not want to write the columns name, you must give the value `DEFAULT` (see example below);
- More generally, each missing data for a column will receive its default value, if it has one. If that column was set to not accept NULL value and you do not provide one, the insertion will fail;
- The order of the data is not important as long as it corresponds to the column name;
- The name of the column is not necessary. In this case, you must give the values in the same order as the declaration of the table.

```
INSERT INTO table_name
VALUES (values);
```

Note that if you have multiple rows to insert, it is possible to insert them all with only one INSERT request, by separating the different rows with commas.

```
INSERT INTO table_name
VALUES (row1_values), (row2_values);
```

### 3.1.1 Examples:

These are some insertions with different syntaxes:

```
-- Add the movie `The Usual Suspects` to the `movie` table
INSERT INTO movie (movie_title)
    VALUES ('The Usual Suspects');

-- Add the movie `Citizen Kane` to the `movie` table
INSERT INTO movie
    VALUES (DEFAULT, 'Citizen Kane');

-- Add a rating to Citizen Kane in the rating table
INSERT INTO rating (rating_author, rating_value, rating_comment, rating_movie)
    VALUES ('John Doe', 3, NULL, 2);

-- Add a rating to Citizen Kane in the rating table
INSERT INTO rating
    VALUES ('John Smith', 4, 'Breathtaking all the way to the end !', 2);

-- Add a rating to The Usual Suspects in the rating table
INSERT INTO rating (rating_author, rating_value, rating_movie)
    VALUES ('John Doe', 5, 1);

-- Add a rating to The Usual Suspects in the rating table
INSERT INTO rating
    VALUES ('John Smith', 4, 'One of the finest films ever made.', 1);

-- Add multiple movies with only one request:
INSERT INTO movie (movie_title) VALUES
    ('Pulp Fiction'),
    ('Fight Club'),
    ('Léon: The Professional');

-- Add multiple ratings
INSERT INTO rating VALUES
    ('John Smith', 5, 'Simply amazing !', 4),
    ('John Doe', 5, 'Very good movie.', 5),
    ('John Smith', 4, NULL, 3),
    ('John Smith', 3, NULL, 5);
```

## 3.2 Alteration

Altering data of a table can be done with the `UPDATE` statement. It is possible to change all the lines of the table or just a subset.

```
UPDATE table_name
SET column_name = new_value
[WHERE column_name = value];
```

The `WHERE` condition is very important because it allows you to specify which lines are going to be changed.

### 3.2.1 Example:

Imagine an error in an insertion:

```
INSERT INTO rating
VALUES ('John Doe', 0, 'A shameful praise of a fantastic film.', 4);
```

Here is the way to fix your mistake:

```
UPDATE rating
SET rating_value = 4
WHERE rating_author = 'John Doe' AND rating_movie = 4;
```

Now, imagine you added some ratings with a bad value which is too low:

```
UPDATE rating
SET rating_value = rating_value + 1;
```

## 3.3 Deletion

Now that you know how to add and alter some data, here is how to delete some. You are going to use the `DELETE` statement this way:

```
DELETE FROM table_name
[WHERE column_name = value];
```

As with the `UPDATE` statement, you can use the `WHERE` condition to specify which lines need to be deleted.

## 3.4 Exercises

### 3.4.1 Insertion

Add some cans to your `can` table:

- A Red Bull of 33 cl
- An Apple Juice of 33 cl

Add yourself and 2 people of your choice in the `student` table.

### 3.4.2 Alteration

Update the Apple Juice name to Pineapple Juice.

### 3.4.3 Deletion

The Red Bull is no longer available. Delete this can from your table.

## 4 Conditional operators

### 4.1 BETWEEN

The BETWEEN operator allows you to select a data interval in a request. The interval can be applied on several types like strings, numerical values or dates.

```
...
WHERE values BETWEEN value1 AND value2
```

The opposite operation can be done with NOT BETWEEN.

### 4.2 IN/NOT IN

It is possible to use IN/NOT IN to check if a given value matches one of a given set of values. The syntax is:

```
value IN (value1, value2)
```

The list is not limited to a simple enumeration. It is also possible to use a complete request as long as it returns a list that can be compared to the tested value.

### 4.3 NULL

The NULL value represents a missing data. It is not a 0, nor a proper value. It is just a value which is not in the table. During the creation of a table you can specify if you want to allow - or not - a NULL value for a specific column. You can check if a value is NULL thanks to the IS NULL operator. The IS NOT NULL operator also exists and tests the opposite result.

```
...
WHERE value IS NULL
```

### 4.3.1 COALESCE(value [, ...])

The COALESCE function returns the first argument that is not NULL. NULL is returned only if all arguments are null. It is often used to substitute null values for a default value when data is retrieved for display, for example:

```
COALESCE(value1, value2, '(none)')
```

## 4.4 Exercises

### 4.4.1 Request 1

Display all the cans whose capacity is greater than 30cl and smaller than 40cl.

### 4.4.2 Request 2

Display all the logins of students without their first names.

### 4.4.3 Request 3

For each can purchased between the 1st and the 10th of February 2018 display the purchase_time and the student's login in alphabetical order.

### 4.4.4 Request 4

Display all student first names. If a student has no first name, display 'no_name' instead.

## 4.5 CASE

The `CASE ... WHEN` statement is similar to the `if ... then` structure you are familiar with in other languages. It allows you to return a result depending on a condition. It can be used in a `SELECT`, a `WHERE` or an `ORDER BY`. For example:

```
SELECT name,
    CASE WHEN capacity_cl < 30 THEN 'Not enough!'
         WHEN capacity_cl < 99 THEN 'No longer thirsty!'
    END AS capacity
FROM can
ORDER BY capacity_cl;
```

```
     name        |      capacity
-----------------+--------------------
 Coke            | Not enough!
 Oasis           | Not enough!
```

(continues on next page)

(continued from previous page)

```
 Diet Coke       | Not enough!
 Orangina        | Not enough!
 Sprite          | Not enough!
 Pepsi           | Not enough!
 Ice Tea         | Not enough!
 Fanta           | Not enough!
 Pineapple Juice | No longer thirsty!
 Orange Juice    | No longer thirsty!
 Water           | No longer thirsty!
(11 rows)
```

### 4.6 Exercises

#### 4.6.1 Request 5

Classify the purchases depending on the date. Before January $31^{th}$ 2017, create a label 'old'. Between the $1^{st}$ of February 2017 and the $1^{st}$ of January 2018 add the label 'outdated'. Finally, for the dates after the $1^{st}$ of January 2018, add the label 'current'.

#### 4.6.2 Request 6

Associate a shop with a label of your choice. Display the name of the shop and the label, and order by labels.

## 5 Functions and operators on strings

### 5.1 Operators

#### 5.1.1 `LIKE`

The `LIKE` operator allows you to test a string and returns a boolean.

- An underscore (_) in a pattern stands for one character (no matter which one)
- A percent (%) represents zero or several characters

For example:

```
'yaka' LIKE 'yaka' -- true
'yaka' LIKE 'ya'   -- false
'yaka' LIKE 'ya%'  -- true
'yaka' LIKE '_ak_' -- true
'yaka' LIKE '_a_'  -- false
```

If the string you are looking for contains a special character like an underscore or a percent, you must prefix these characters with an escape character. By default, it is a backslash, but it is possible to define another one with the `ESCAPE` clause.

```
'yaka' LIKE 'ya%%'             -- true
'yaka' LIKE 'ya%%' ESCAPE '%' -- false
'ya%'  LIKE 'ya%%' ESCAPE '%' -- true
```

The `LIKE` operator is also accompanied by `NOT LIKE` which returns the opposite result and `ILIKE` which executes the research without paying attention to the case.

```
'yaka' ILIKE 'YakA' -- true
```

#### 5.1.2 `SIMILAR TO`

The `SIMILAR TO` operator works like `LIKE` but uses the SQL standard definition of the regular expression. It is a mix between `LIKE` and the normal usage of a regular expression. Because of this mix, `SIMILAR TO` handles several meta-characters borrowed from POSIX.

- `|`
- `*`
- `+`
- `(...)`
- `[...]`

```
'yaka' SIMILAR TO 'yaka'    -- true
'yaka' SIMILAR TO 'a'       -- false
'yaka' SIMILAR TO '%(j|k)%' -- true
'yaka' SIMILAR TO '(j|k)%'  -- false
```

### 5.2 Functions

#### 5.2.1 trim([leading | trailing | both] [characters] from string)

Allows you to delete a specific character at the beginning and/or at the end of the string.

```
SELECT trim(both 'x' from 'xJohnxx'); -- John
```

#### 5.2.2 substring(string from pattern)

Extracts a substring from a regular expression.

```
SELECT substring('JohnDoe' from '...$'); -- Doe
```

### 5.2.3 concat()

Concatenates all arguments. NULL arguments are ignored. It is also possible to use the operator || which can be used with other data types (numeric...).

```
SELECT concat('abcde', 2, NULL, 22); -- abcde222
SELECT 'Post' || 'greSQL';           -- PostgreSQL
```

### 5.2.4 replace()

Replaces all occurrences of a substring with another one.

```
SELECT replace('abcdefabcdef', 'cd', 'XX'); -- abXXefabXXef
```

### 5.2.5 length()

Number of characters in a string.

```
SELECT length('John'); -- 4
```

### 5.2.6 upper/lower()

Allows you to pass a string respectively to upper or lower case.

### 5.2.7 lpad/rpad()

Fills up the string to the given length by appending the given characters.

```
SELECT rpad('hi', 5, '-'); -- 'hi---'
SELECT lpad('hi', 5, '-'); -- '---hi'
```

If the string is too long, it is truncated to the desired length.

## 5.3 Exercises

### 5.3.1 Request 1

Display all first names and names of the students whose first name starts with an 'A'.

### 5.3.2 Request 2

Display all students whose first name's length is 5 letters or less.

### 5.3.3 Request 3

Display all students named 'Paul'. The research must not depend on the case.

# 6 Functions and operators on dates

## 6.1 Formats

The SQL language supports the set of following types:

- `timestamp`: a date and an hour;
- `interval`: time interval;
- `date`: date only;
- `time`: hour only.

The date formats are used like strings: you must surround them with simple quote. In this example, we are looking for a can purchased on February 2nd, 2018 at 5:40PM.

```
SELECT * FROM student_can_shop
WHERE purchase_time = '2018-02-02 17:40';
```

```
 id |  login   | can_id | shop_id |    purchase_time
----+----------+--------+---------+---------------------
 13 | theoph_a |      8 |       1 | 2018-02-02 17:40:00
(1 row)
```

Note that casts are useful here. They can be done like this:

```
'2018-03-05 23:42:00'::timestamp;    -- 2018-03-05 23:42:00
'2018-03-05 23:42:00'::date;         -- 2018-03-05
'2018-03-05 23:42:00'::time;         -- 23:42:00
```

## 6.2 Operators

In SQL, it is possible to apply arithmetic operators to compare dates and hours. For example, by using the previous request, we can choose to display all the cans purchased 1 day and 10 minutes before February 2nd, 2017.

```
SELECT * FROM student_can_shop
WHERE purchase_time = timestamp '2017-02-02 17:40'
    - interval '1 day 10 min';
```

```
 id |  login   | can_id | shop_id |    purchase_time
----+----------+--------+---------+---------------------
 12 | theoph_a |      7 |       3 | 2017-02-01 17:30:00
 25 | firmin_v |      7 |       1 | 2017-02-01 17:30:00
(2 rows)
```

## 6.3 Functions

### 6.3.1 Current date and hour

The function `now()` allows you to get the current date and hour.

### 6.3.2 Information about a date

The `extract()` function allows you, as its name suggests, to extract information about the given date/hour. Here is an example to show you how the function works:

```
SELECT extract(hour from timestamp '2018-03-05 23:42:00');
```

### 6.3.3 Date format

to_date() allows you to transform a string into its equivalent date/hour format. The following example generates a date depending on the given format:

```
SELECT to_date('05032017', 'DDMMYYYY');
```

## 6.4 Exercises

### 6.4.1 Request 1

Display all the logins and purchase_time of the students who bought some cans on January 25th, 2017.

### 6.4.2 Request 2

Create and display a timestamp for June 1st, 2017 at 10:56AM.

### 6.4.3 Request 3

Display the result of the previous timestamp with the subtraction of the timestamp of May 19th, 2017 at 00:00AM. Call the column 'interval'.

### 6.4.4 Request 4

Display in 3 different columns the day, the month and the year of the purchased cans.

# 7 Transactions

## 7.1 Information

As you may know, the real strength of a database is its ability to handle simultaneous requests from several users. This works very well with single requests.

Now imagine you want to execute several requests, but you want to guarantee that every single one succeeded. If the very last request you execute violates a constraint you need to cancel all previous requests.

Transactions can accomplish this.

A transaction groups several requests into one atomic request. At the end of a transaction, you can either choose to commit the changes or roll them back. If an error happens inside a transaction, it is automatically rolled back after the end of the transaction.

To start a transaction you can use:

```
BEGIN;
```

After this keyword all the queries are temporary, and will only be definitive after a `COMMIT` or an `END` occurs, and if nothing went wrong. If something went wrong, you can choose to apply a `ROLLBACK` to undo your changes. If an error occur you will not be able to `COMMIT` your changes.

For example:

```
/* Begin transaction */
BEGIN;

/* deletion of the entire student_can_shop table
 * (Oops! I forgot the WHERE clause)
 */
DELETE FROM student_can_shop;

/* Everything is gone */
SELECT * FROM student_can_shop;

/* Rollback all changes */
ROLLBACK;
```

(continues on next page)

(continued from previous page)

```
/* Everything is back */
SELECT * FROM student_can_shop;
```

If, instead of the `ROLLBACK`, we had done a `COMMIT` or an `END`, all the changes would have been definitely applied to the table.

# 8 Grouping

## 8.1 The GROUP BY clause

The `GROUP BY` clause can group the results of a `SELECT` request. This means it merges several rows into one. The rows with the same values in the specified columns after the `GROUP BY` clause will be put together.

When the `GROUP BY` clause is used, every column appearing in the `SELECT` clause must also appear after `GROUP BY`. The reason for this is that the columns that are not used would have several possible values when the rows are merged. This rule does not apply to columns that appear inside aggregate functions.

The `GROUP BY` clause is useful when using aggregate functions. An aggregate function is a function that computes a value using the values of several rows.

```
SELECT count(*) AS number_of_rows FROM table;
```

`count(column_name)` is an aggregate function. This request will return a single value: the number of rows in the table. This information can be useful, but with grouping, we can do even more interesting things.

When an aggregate function is used on a query that uses `GROUP BY`, its behaviour changes. Instead of processing all rows of the table and yielding only one result, it applies on every group of items and yields the result of every group.

```
SELECT shop_id, count(*) AS choice
FROM shop_can
GROUP BY shop_id;
```

This request groups all the rows of `shop_can` according to their `shop_id` and counts the number of rows in every group. This basically gives us the number of different cans sold by a shop. You can see that this yields several rows, one for each shop.

If we want to know which shops sell more than 8 different types of can, we could be tempted to add `WHERE count(*) > 8`. This won't work because of the execution order of SQL queries, as the WHERE clause will be used to know which data will be retrieved **before** grouping it. For these particular cases where we need to put a condition on the grouped rows, we use the `HAVING` keyword:

```
SELECT shop_id, count(*) AS choice
FROM shop_can
-- WHERE clause should be here
GROUP BY shop_id
HAVING count(*) > 8;
```

## 8.2 Additional aggregate functions

There are many other aggregate functions in PostgreSQL. We will present a few more here.

```
max(expression)
min(expression)
```

Returns the highest or lowest value of a collection of numeric values, strings or date/time values.

```
SELECT max(price) FROM shop_can;
```

This query returns the highest price of an item sold by any shop. You could return the highest price from each shop using the `GROUP BY` clause.

```
SELECT shop_id, max(price) FROM shop_can
GROUP BY shop_id;
```

Other general mathematical functions are available:

```
sum(expression)
```

Returns the sum of a collection of numerical values.

```
avg(expression)
```

Returns the average of a collection of numerical values.

For more information, please refer to the documentation of PostgreSQL[1].

## 8.3 Exercises

### 8.3.1 Request 1

Display the amount of assistant and non assistant students that are registered in the database.

Expected result:

```
 assistant | count
-----------+-------
 f         |    27
 t         |     4
(2 rows)
```

[1] https://www.postgresql.org/docs/

### 8.3.2 Request 2

We would like to know the average price of a can for each shop.

To get two digits after the comma, you can use the `round()` function. Be careful, the round function expects a numerical value. You will have to `CAST` the value returned by the average function.

```
CAST(value AS numeric)
```

Expected result:

```
  name   | average_price
---------+---------------
 Crocus |          1.20
 Lidl   |          1.10
 Okabe  |          1.00
(3 rows)
```

### 8.3.3 Request 3

Display the login and the total amount of money spent on cans by each student. You have to order your results by total. The student who spent the most must be first.

Expected results:

```
  login   | total
----------+-------
 theoph_a |  15.4
 arnaud_p |   8.4
 cyrill_a |   4.4
 alphon_a |   3.2
 nazare_p |   2.2
 firmin_v |   2.2
 daniel_n |   2.2
(7 rows)
```

*The only way out is through*