SQLWorkshop — Tutorial D2

version **#v1.0.6**

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

Copyright

This document is for internal use at EPITA (website) only.

Contents

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

If you are using the database you used during the previous tutorial you might have modified it. We suggest you to relaunch script.sql if you want the same results as ours.

1 Joins

1.1 Introduction

Now that your are aware of the different syntaxes of PostgreSQL and the basic statements to realize simple selections of data, we are going to see how to make a select on several tables. There are different ways to do it but today we are going to view the joins.

1.2 Link between tables

A relational database without link between tables would not be very interesting. When you establish a model for a database, the entities are generally all linked. In our example, it is the case between the shop and can entities, which is normal because shops are composed of several cans.

In practice, the fields used to identify the other entities are primary keys of these entities. When a table contains the primary key of another table to reference an entity, we call this information a **foreign key**. The use of these keys is important because the DBMS will guarantee that the element you want to reference must exist in the database.

For example, in the given file, an `address` is linked to a `student`. To mark this link in the database, the primary key of an address is in the student table and the `address` field in the `student` table is a foreign key. There is a constraint on this field to guarantee that this key matches the address which belongs to an existing table. Thus, if you try to inquire an address which does not exist during the student insertion, it is going to fail.

1.3 Get information from several tables

You will often need to get information from several tables. For example, if you want to get the students living at one address, you will have to look in the `student` table (to get the names of the students) and in the `address` table (to get the address). To do so, we use the joins. It can be decomposed in two steps:

1. A Cartesian product which is going to link all the elements of one table with another one.
2. A restriction on this Cartesian product which will eliminate all the elements that do not respect one given condition.

Several joins exist:

- inner
- outer
- cross

1.4 Inner joins

1.4.1 Generalities

The most used join is the inner join. The goal is to select all the columns of two tables in one query. These tables have to be linked by a common key.

```
SELECT * FROM A, B
WHERE A.id_B = B.id
```

Here, we used a `WHERE` to make the link between the two tables. To optimize and increase the readability of the queries, we are going to use the expected syntax for this purpose:

```
SELECT * FROM A
INNER JOIN B ON A.id_B = B.id
```

Although the syntax is different, the join is still the same. However, the joined tables are clearly indicated and the DBMS is aware of the join and owns all the elements to do it in an optimal way.

We can also use the keyword `JOIN` alone: it corresponds to an inner join.

```
SELECT * FROM A
JOIN B ON A.id_B = B.id
```

1.4.2 Using

If your condition is made on two columns that have the exact same name, a keyword can simplify the notation: `USING`. Admitting the table A and B have the same column `uid`, we can use the `USING` keyword here:

```
SELECT * FROM A
INNER JOIN B USING(uid)
```

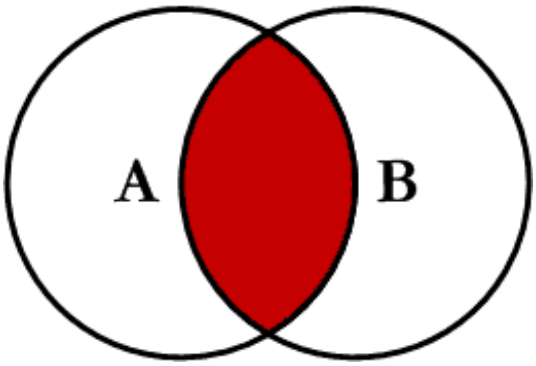
This keyword can be useful to shorten the length of the query but it does not allow to use an alias. The other inconvenient is that queries are less readable. For these reasons, you should use it with caution.

1.4.3 Illustration

To help you have an idea of how this join works, here is a sketch to illustrate what is going on.

We can clearly see that the query is going to return the result of the left table (table A) which has a result matching the right table (table B).

Note that it is possible to make several joins and to add the `WHERE` clause to specify your queries.



1.4.4 Exercises

Query 1

Display the first name, last name and country of the students living in the United Kingdom.

Query 2

For each purchase, display the student's login, the shop's name and the can's name.

Query 3

Display the first name, last name and purchase time for the students who bought some Coke or Diet Coke.

Query 4

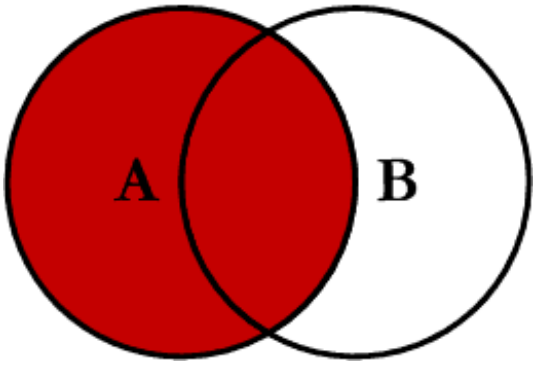
Display the different cans sold by shops ordered by shop names.

1.5 Outer joins

In this part, we are going to see another form of joins: the outer joins. These joins are less restrictive than the inner joins. Indeed, in the previous join, we have seen that, if no lines in the remote table can be joined, no result will be displayed. The various types of outer joins make it possible.

1.5.1 Left join

The left join will allow you to display all the results from the left table (table A) even if no entry matches the join table in the right one (table B).



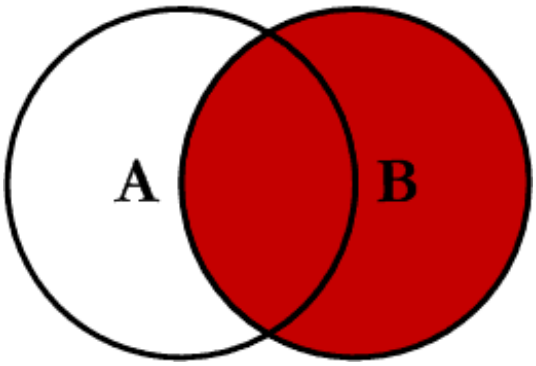
For all the results without neighbors in the other table, the columns are set to NULL.

```
SELECT * FROM A
LEFT JOIN B ON A.id_B = B.id
```

```
SELECT * FROM A
LEFT OUTER JOIN B ON A.id_B = B.id
```

1.5.2 Right join

The right join is the opposite of the previous one. It will display all the results from the right table (table B) even if no entry matches the left table (table A).



The syntax is the same as for the left join, except that the `LEFT` keyword becomes `RIGHT`. Obvious, right ?

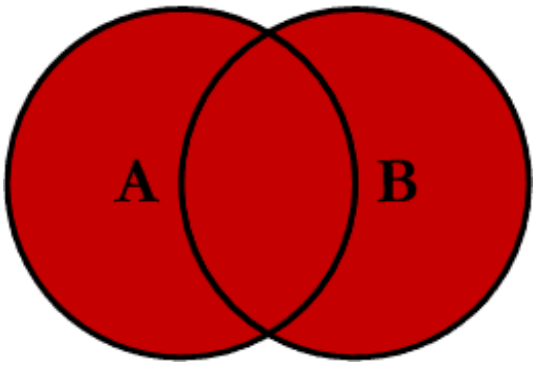
```
SELECT * FROM A
RIGHT JOIN B ON A.id_B = B.id
```

```
SELECT * FROM A
RIGHT OUTER JOIN B ON A.id_B = B.id
```

You must understand what left and right mean here. The left table is the initial table on which the data is selected with `SELECT ... FROM A`. The right table is the one on which we apply the join.

1.5.3 Full join

This join will allow you to perform a left and right join at the same time. It means that it is going to recover all the results given by the two tables by combining only the lines which need it. Thus, there is no restriction on the lines.



This join has a syntax similar to the previous ones. We keep the same skeleton by changing the join keyword only.

```
SELECT * FROM A
FULL JOIN B ON A.id_B = B.id
```

```
SELECT * FROM A
FULL OUTER JOIN B ON A.id_B = B.id
```

Be careful!

`ON` is processed before the join whereas the `WHERE` is processed after the join so the same query with `WHERE` instead of `ON` won't have the same result.

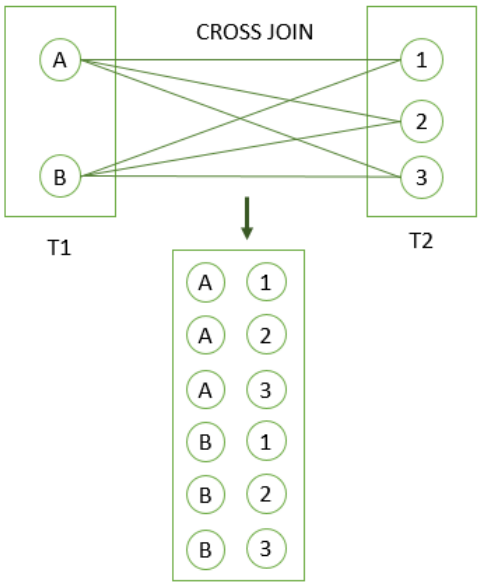
1.6 More joins!

To allow you to see a big part of the possible joins, here is a special chapter about the other existing joins.

1.6.1 Cross join

This type of join is quite special. It corresponds to a Cartesian product. The query will display all the possible combinations of the lines from the table A and B. For a table A having n lines and a table B having m lines, the results will be composed of n * m lines in the end.

```
SELECT * FROM A
CROSS JOIN B
```



1.6.2 Natural join

There is a keyword which will allow the DBMS to find how to join two tables by itself. This keyword must be placed before the previous keywords of joins and doesn't need a clause `ON` or `USING` after that. It behaves like the `USING`.

```
SELECT * FROM A
NATURAL LEFT JOIN B
```

If the natural join does not achieve to find a column with the same name in each table, it is going to work like a `CROSS JOIN`. When two columns can be used for the join, there is an undefined behavior. So you should be aware of the risks.

1.7 Exercises

1.7.1 Query 5

Display the login of each student, followed by his address even if he does not have one.

2 View

2.1 Definition

You know how to get information from several tables using `JOIN`s. These requests may be tedious to write. If you use them frequently, you may want to add a view in your database.

A **view** will allow you to give a name to a request in order to be able to call it as easily as a table.

Using views is a good practice. They will allow you to encapsulate the implementation of your database, and also provide an easier check of its behaviour.

2.2 Syntax

The syntax to create a view is the following:

```
CREATE [OR REPLACE] VIEW view_name (column_name, ...) AS
request;
```

The view will bear the name `view_name` and will allow the creation of an alias for the SQL request `request`. The request could be any request of type `SELECT` or `VALUES`. The part that allows the definition of the column name is optional. You will see its effect later.

For instance, if we decide to display the name of each country for every student, we could do the following:

```
CREATE VIEW v_student_country AS
    SELECT student.login,
           lower(address.country)
    FROM student
        INNER JOIN address
            ON address.id = student.address_id
    ORDER BY address.country;
```

Thus, to list this information, we can use the view and obtain the result thanks to a simple `SELECT` query:

```
SELECT * FROM v_student_country;
```

Note that the use of a view is exactly like the normal use of a `SELECT` on a table. The name of the columns are the same. In this example, the view doesn't abstract entirely from the internal representation of the database.

In this situation, the best would be to change the name of the columns used by the view. This can be done thanks to the following syntax:

```
CREATE VIEW v_student_country (student, country) AS
    SELECT student.login,
           lower(address.country)
    FROM student
        INNER JOIN address
            ON address.id = student.address_id
    ORDER BY address.country;
```

As you can see, an error is generated. If you want to change a column, you need to drop the view first.

```
DROP VIEW v_student_country;
```

This time, the name of the column is the one we just defined. Now, we are independent of the way the database is built.

2.2.1 Reminder

Do not forget to `DROP VIEW` to delete a view if you want to modify it after creating it. If you want to avoid deleting a view before defining a new version, you can use `CREATE OR REPLACE VIEW` instead of `CREATE VIEW`.

2.3 Exercises

2.3.1 Request 1

Create a view called `v_student_can` to display the cans purchased by the students. It should contain two columns: the student login and the can name. If the student bought three Fanta cans, there should be three entries. Order the results by alphabetical order of the logins first, then of the cans.

A select on the view should return the following result:

```
  login   |     name
----------+--------------
 alphon_a | Coke
 alphon_a | Coke
 alphon_a | Coke
 arnaud_p | Orange Juice
 arnaud_p | Orange Juice
 arnaud_p | Orange Juice
 arnaud_p | Orange Juice
 arnaud_p | Orange Juice
 arnaud_p | Orange Juice
 arnaud_p | Orange Juice
 cyrill_a | Ice Tea
 cyrill_a | Ice Tea
 cyrill_a | Ice Tea
 ...
 theoph_a | Oasis
```

(continues on next page)

(continued from previous page)

```
 theoph_a | Oasis
 theoph_a | Oasis
 theoph_a | Water
 theoph_a | Water
(41 rows)
```

2.3.2 Request 2

Create a view called `v_shop_time` that contains every transaction with the shop name and the time it was made at. Sort by shop name and purchase time.

A select on this view should yield the following result:

```
  name  |    purchase_time
--------+---------------------
 Crocus | 2017-01-29 16:15:00
 Crocus | 2018-01-21 16:23:00
 Crocus | 2018-01-23 17:42:00
 Crocus | 2018-02-02 00:37:00
 Crocus | 2018-02-03 19:11:00
 ...
 Lidl   | 2018-02-13 12:24:00
 Lidl   | 2018-03-01 12:24:00
 Okabe  | 2017-02-01 17:30:00
 Okabe  | 2018-01-30 21:48:00
(41 rows)
```

3 Subrequests

3.1 Generalities

Subrequests are an alternate way to retrieve data from a database. They are similar to `JOIN`s in that they allow you to get equivalent results.

```
SELECT *
FROM ( SELECT column1, column2
       FROM B ) AS C
```

The expression inside the parentheses returns some kind of table, that is named using the `AS` keyword. Naming subquery results is necessary in PostgreSQL.

This request uses a subrequest as a table. Even if it seems useless for now, there are several operators that can be applied to subqueries. We will present the main ones here.

3.2 EXISTS

The `EXISTS` keyword can be used on a subquery to check if it had any result. A boolean value is then returned. This is mainly used in `WHERE` clauses. The main advantage of the `EXISTS` keyword is that it will stop at the first result, because the rest is not needed. It is useful when your query takes a long time to execute. As this subquery may not be fully executed, it is not recommended to apply any modification in it.

```
SELECT *
FROM student
WHERE EXISTS(
            SELECT *
            FROM student_can_shop
            WHERE student.login = student_can_shop.login
          );
```

This request does not serve a real purpose, it displays the content of the student table if the subquery returns a result.

3.3 IN/NOT IN

The `IN` keyword is used to check if a value is inside a list. This list can be obtained with a subquery that must return only one column. Like with the `EXISTS` keyword, the complete execution of the query is not guaranteed.

For example, if you run the following command, you will obtain an error message:

```
SELECT *
FROM can
WHERE 'firmin_v' IN (SELECT login, assistant FROM student);
```

The error message shows that the subquery has too many columns. It is not a list. The following example is correct:

```
SELECT *
FROM can
WHERE 'firmin_v' IN (SELECT login FROM student);
```

The `NOT IN` keyword is used in the same way, but checks if a value is not inside a list.

The `ROW` keyword can be used to create a table row. That is why it can be used to match several columns in one subrequest with the `IN` keyword. For example, if the expression is `ROW(value1, value2)`, the subrequest must return two columns.

```
SELECT *
FROM can
WHERE ROW('nazare_p', FALSE)
IN (SELECT login, assistant FROM student);
```

3.4 ANY/SOME

ANY and SOME are synonyms, they serve the same purpose.

Their behaviour is close to the IN keyword, but they can be used with an operator. The additional operator makes comparisons possible.

```
SELECT *
FROM can
WHERE id > ANY (SELECT id FROM shop);
```

This request returns all the cans that have an id higher than the id of any shop in the shop table.

It is possible to get the same behaviour as the IN keyword using ANY/SOME. Using = ANY is exactly the same as using IN.

3.5 ALL

ALL is a keyword that uses a similar syntax as ANY/SOME. It returns TRUE if every single element of the subquery satisfies the comparison, if one of them does not verify the comparison it returns FALSE.

```
SELECT *
FROM can
WHERE id > ALL (SELECT id FROM shop);
```

Once again, note that the expression <> ALL is the same as NOT IN.

3.6 Exercises

3.6.1 Request 1

Write a request that returns all the students that didn't buy any drinks yet. You are not allowed to use JOINs.

This is the output you should obtain:

```
     login
----------------
 aghavn_a
 batrie_p
 bedros_g
 claire.billy
 cosee_t
 domini_s
 el_hal_a
 emeric_a
 geinau_h
 gevorg_e
 guyree_b
 joel_c
 leo_m
```

(continues on next page)

(continued from previous page)

```
 login_x
 marcel_i
 muriel_l
 paulin_a
 paul.khuat-duy
 prospe_j
 roch_c
 roch_h
 rouben_u
 theophane.vie
 toros_n
(24 rows)
```

3.6.2 Request 2

Write a request that displays all the cans the shop 1 (`shop_id = 1`) is selling. You are not allowed to use `JOIN`s.

This is the output you should obtain:

```
  name
----------
 Coke
 Ice Tea
 Oasis
 Orangina
 Sprite
 Water
(6 rows)
```

4 Stored Procedures

4.1 Generalities

SQL is very powerful, but there are limitations as to what is technically possible with the language.

The SQL Procedural Language, or PL/SQL, overcomes most of these difficulties. It can be used to process several requests, use control structures, declare variables and create triggers.

The name of the language you will use is PL/pgSQL.

When you declare a procedure, the block of code that is executed is stored in the database. A code block is structured in the following way:

```
DECLARE
    variable declarations
BEGIN
    instructions
END;
```

When declaring a procedure, you need to add the appropriate header. It needs to comply with the following syntax:

```
CREATE [OR REPLACE] FUNCTION function_name(arg1 ARG1TYPE, arg2 ARG2TYPE, ...)
    RETURNS RETURN_TYPE AS
$$
DECLARE
    variable declaration
BEGIN
    instructions
END;
$$ LANGUAGE plpgsql;
```

The `DECLARE` block is optional. Do not declare useless variables.

This language gathers several characteristics of regular procedural languages. It is possible to take arguments, assign variables, test conditions, loop over instructions, etc.

Procedures can use the `VOID` return type to indicate that they don't return anything. To return a result, the `RETURN` keyword is used.

4.2 Variable declarations

In the `DECLARE` section of the procedure, you can put as many variables as you want. Variables can be of any type supported by PostgreSQL[1]. This is the syntax to declare a variable:

```
var_name VAR_TYPE;
```

For example:

```
my_local_integer INT;
```

4.3 Instruction block

In an instruction block you can do several things. One of the most basic statements is the assignation of a variable.

```
my_variable := expression;
```

Any SQL request can be used inside the instruction block. You can use the arguments passed to your function to construct your request.

For example, this function adds a new type of can to the database of available cans.

```
CREATE FUNCTION add_new_can(name VARCHAR(64), capacity_cl INT)
    RETURNS VOID AS
$$
BEGIN
    INSERT INTO can (name, capacity_cl)
```

(continues on next page)

[1] https://www.postgresql.org/docs/current/static/datatype.html

(continued from previous page)

```
    VALUES (name, capacity_cl);
END
$$ LANGUAGE plpgsql;
```

Once the function is correctly defined in your database, you can call it using the following syntax:

```
SELECT add_new_can('King-size milk tank', 100000);
```

4.4 Retrieving a simple result

You can retrieve a query result easily if it is contained in one row. You just have to use the `INTO` keyword to save the result of the request into a variable.

```
SELECT firstname
INTO a_string
FROM student
WHERE login = 'alphon_a'
```

The variable `a_string` will contain the first name of the student that has `alphon_a` as login. Do not forget to declare it first.

It is possible to get the results of multiple rows by entering multiple variable names after the `INTO` clause.

```
SELECT firstname, lastname
INTO firstname_var, lastname_var
FROM student
WHERE login = 'daniel_n'
```

There is also a way of returning the result of one row, called a `RECORD`. We will see this notion a little further.

4.5 Control structures

The PL/SQL control structures are very similar to the ones you have seen in other languages. Depending on the result of a condition you can execute a bloc of code or another. The `IF` block has the following structure:

```
IF boolean_expression THEN
    instructions
[ ELSIF boolean_expression THEN
    instructions ]
    ...
[ ELSE
    instructions ]
END IF;
```

Here is an example using two locally declared variables, `result` and `number`:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSE
    result := 'negative';
END IF;
```

4.6 Found Variable

The special variable FOUND is set to TRUE if the last request returned at least one row. It can be used to check if a request was successful with the IF structure for example. A word of caution though: in PL/pgSQL, when you do not store the returned value of a SELECT, you need to use the PERFORM keyword instead.

```
CREATE FUNCTION test_existence_student(student_login VARCHAR(64))
    RETURNS BOOLEAN AS
$$
BEGIN
    PERFORM * FROM student WHERE login = student_login;
    RETURN FOUND;
END;
$$ LANGUAGE plpgsql;
```

Here you can see we use FOUND like a variable to check if a student exists.

4.7 Loop Statements

The PL/pgSQL handles loop instructions to repeat a block of instructions. The well-known WHILE and FOR structures can be found in the PostgreSQL documentation.

The FOR statement can also be used to iterate over a collection of rows yielded by a SELECT query. This is useful if you need to apply an operation on every row one by one.

```
FOR target IN query LOOP
    instructions
END LOOP;
```

The target will, for each iteration, receive the result of one row returned by the query. You can specify a list of variables to retrieve several columns. Do not forget the target must be a declared variable. You will need the DECLARE block.

```
FOR current_id, current_name IN
    SELECT id, name FROM shop
LOOP
    /* instructions */
END LOOP;
```

There is another type of variable that is easier to use as target. PostgreSQL handles a RECORD type that can encapsulate all the columns of one row. In the previous example, we could have used it instead of the two variables.

```
FOR record_var IN
    SELECT id, name FROM shop
LOOP
    /* instructions */
END LOOP;
```

To access the variables inside the record, you can use it like this:

```
record_var.id
record_var.name
```

If a procedure returns a `RECORD`, it must be called by naming the contained record variables and specifying their type.

```
SELECT * FROM my_function() AS (i INT, n TEXT);
```

The type of the variables and the number of columns must match what is returned by the function.

4.8 Exceptions

Sometimes, errors may happen during the execution of your function. For example, if a new element does not respect a table's constraint, an exception will be raised, and the execution will be interrupted to maintain the table in a sane state.

The `EXCEPTION` clause allows you to handle these events and take appropriate measures. The syntax is the following:

```
DECLARE
    variables
BEGIN
    instructions
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        instructions_for_error_handling
    [ WHEN condition [ OR condition ... ] THEN
        instructions_for_error_handling
    ... ]
END;
```

To handle all other types of exceptions, you can use the `OTHERS` condition that will catch all remaining ones.

```
DECLARE
    variables
BEGIN
    instructions
EXCEPTION
    WHEN OTHERS THEN
        [ ... ]
END;
```

It is possible to raise your own exceptions by using the `RAISE` keyword. If you want to display a message you can use `RAISE NOTICE`. This can be very useful for quick debugging. We strongly suggest you read

the documentation to see all of its uses.

4.9 Return a set of elements

The PL/pgSQL language is able to return a list of values of the same type. The `SETOF` keyword serves this purpose. The returned value is represented as several rows of values of the selected type.

To declare a function that returns a set of `TEXT` type variables, you can use the following syntax:

```
FUNCTION a_function() RETURNS setof text
```

The array must be constructed by returning every element one by one. This is done using the `RETURN NEXT` keywords. When you have returned all the values, you can quit the function with a single `RETURN`.

The following function shows an example of how you can use this feature:

```
CREATE FUNCTION test_set()
    RETURNS SETOF TEXT AS
$$
BEGIN
    RETURN NEXT 'element 1';
    RETURN NEXT 'element 2';
    RETURN NEXT 'element 3';
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

You can use a function returning a set as a table, and apply a query on it. It will produce several rows as you can see below:

```
SELECT * FROM test_set();
```

Output:

```
test_set
---------
element 1
element 2
element 3
(3 rows)
```

It is possible to return a set of records using `SETOF RECORD`, but you will have to specify the fields in the select query:

```
SELECT * FROM test_set_record() AS (id INT, name VARCHAR);
```

It is possible for a function to return a table. It is useful, because you can name the different returned columns:

```
CREATE FUNCTION test_query()
    RETURNS TABLE (id_ INT, name_ VARCHAR) AS
$$
BEGIN
```

(continues on next page)

(continued from previous page)

```
    RETURN QUERY
    SELECT id, name FROM can;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT * FROM test_query();
```

Output:

```
id  | name
----+-----
1   | Coke
2   | Oasis
... | ...
(11 rows)
```

4.10 Exercises

4.10.1 Function 1

Write a function that changes the capacity of a can to the `new_capacity`. If the can id does not exist, it does nothing.

```
FUNCTION update_can_capacity(can_id INT, new_capacity INT)
    RETURNS VOID
```

4.10.2 Function 2

Write a function that adds a new shop to the database and returns a boolean that is true on success and false on failure.

```
FUNCTION add_new_shop(name VARCHAR(64))
    RETURNS BOOLEAN
```

4.10.3 Function 3

Write a function that gets the list of people who drank more than what is passed as argument to the function.

```
FUNCTION get_thirsty(amount INT)
    RETURNS SETOF TEXT
```

When called like this:

```
SELECT get_thirsty(200);
```

The output of this function will respect the following format:

```
         get_thirsty
-----------------------------
 Alexandre Théophile (400cl)
 Philomène Arnaud (231cl)
(2 rows)
```

5 Triggers

5.1 Definition

A trigger is a function that is launched automatically when a certain event happens. You can then choose to launch the function before, after, or instead of the user's action. There are three kinds of event that can be supervised:

- `INSERT`
- `UPDATE` on a particular table or column
- `DELETE`

The following syntax is used to declare a trigger:

```
CREATE TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    FOR EACH { ROW | STATEMENT }
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )
```

A trigger can be executed on the modification of a row, or on a particular query.

When configured to fire on the modification of a row, the function will be executed on every row that is modified. If no rows are modified, the trigger will not be called. If the trigger is configured to fire on a particular query, it will be launched every single time this particular query is called, even if no modifications are applied.

Triggers are useful to check the data that is sent to your database, so that you can be sure it stays in a sane state. It could also be used to log and archive modifications to your tables.

5.2 Usage

To warn PostgreSQL that a function will be used by a trigger, you have to set the return type of the function to `trigger`. It will then be able to use special variables specific to triggers.

`OLD` and `NEW` are two special variables you can use in trigger functions. They point to the values of the monitored row before and after the query. When an `INSERT` is executed, `OLD` will be undefined and when a `DELETE` is executed, `NEW` will be undefined. Of course, these variables are only useful on triggers that apply for each row. If the trigger should not impact the current row, then `NEW` should be returned.

```
CREATE TRIGGER check_update
BEFORE UPDATE ON accounts
FOR EACH ROW
WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
EXECUTE PROCEDURE check_account_update();
```

`TG_OP` is a value that informs which action triggered the function. It is possible to make a trigger fire on multiple types of actions using the `OR` keyword. If the trigger is called on an `INSERT`, the value of `TG_OP` will be `INSERT`.

Other variables can be accessed. You can find the detail in the PostgreSQL documentation[1].

5.3 Exercises

5.3.1 Trigger 1

We want to forbid the addition of a can that has a name which contains 'pepsi', regardless of the case. It must also prevent the user from renaming a can to 'pepsi'. Create the function and the trigger for this.

5.3.2 Trigger 2

We want to monitor every action that is done on the student table. Therefore, you must create a trigger that logs every update, addition, or deletion on the student table.

To log actions, use this table:

```
CREATE TABLE student_modification (
    login     VARCHAR(64),
    action    TEXT,
    time      TIMESTAMP
);
```

The only way out is through

[1] https://www.postgresql.org/docs/current/static/plpgsql-trigger.html