ДЗ 3. Глава 4 (упражнения 2, 4, 8, 12, 15, 21, 30, 33, 35).

2. Предположим, что возникла необходимость хранить в одном столбце таблицы данные, представленные с различной точностью. Это могут быть, например, результаты физических измерений разнородных показателей или различные медицинские показатели здоровья пациентов (результаты анализов). В таком случае можно использовать тип `numeric` без указания масштаба и точности.

   Команда для создания таблицы может быть, например, такой:

```
CREATE TABLE test_numeric
( measurement numeric,
  description text
);
```

Если у вас в базе данных уже есть таблица с таким же именем, то можно предварительно ее удалить с помощью команды

```
DROP TABLE test_numeric;
```

Вставьте в таблицу несколько строк:

```
INSERT INTO test_numeric
  VALUES ( 1234567890.0987654321,
          'Точность 20 знаков, масштаб 10 знаков' );

INSERT INTO test_numeric
  VALUES ( 1.5,
           'Точность 2 знака, масштаб 1 знак' );

INSERT INTO test_numeric
  VALUES ( 0.12345678901234567890,
          'Точность 21 знак, масштаб 20 знаков' );

INSERT INTO test_numeric
  VALUES ( 1234567890,
          'Точность 10 знаков, масштаб 0 знаков (целое число)' );
```

Теперь сделайте выборку из таблицы и посмотрите, что все эти разнообразные значения сохранены именно в том виде, как вы их вводили.

```
template1=# CREATE TABLE test_numeric (
template1(# m numeric,
template1(# d text);
CREATE TABLE
```

```
template1=# INSERT INTO test_numeric VALUES (1234567890.0987654321,
'Точность 20 знаков, масштаб 10 знаков');
INSERT 0 1
```

```
template1=# INSERT INTO test_numeric
VALUES ( 1.5,
'Точность 2 знака, масштаб 1 знак' );
INSERT 0 1
template1=# INSERT INTO test_numeric
VALUES ( 0.12345678901234567890,
'Точность 21 знак, масштаб 20 знаков' );
INSERT 0 1
template1=# INSERT INTO test_numeric
VALUES ( 1234567890,
'Точность 10 знаков, масштаб 0 знаков (целое число)' );
INSERT 0 1
template1=# SELECT * FROM test_numeric
;
           m            |                         d
------------------------+-----------------------------------------------------
  1234567890.0987654321 | Точность 20 знаков, масштаб 10 знаков
                    1.5 | Точность 2 знака, масштаб 1 знак
 0.12345678901234567890 | Точность 21 знак, масштаб 20 знаков
             1234567890 | Точность 10 знаков, масштаб 0 знаков (целое число)
(4 rows)
```

4 номер надо сделать

4. При работе с числами типов `real` и `double precision` нужно помнить, что сравнение двух чисел с плавающей точкой на предмет равенства их значений может привести к неожиданным результатам.

Например, сравним два очень маленьких числа (они представлены в экспоненциальной форме записи):

```
SELECT '5e-324'::double precision > '4e-324'::double precision;

 ?column?
----------
 f
(1 строка)
```

Чтобы понять, почему так получается, выполните еще два запроса.

```
SELECT '5e-324'::double precision;

 float8
----------------------
 4.94065645841247e-324
(1 строка)
```

```
SELECT '4e-324'::double precision;

 float8
----------------------
 4.94065645841247e-324
(1 строка)
```

Самостоятельно проведите аналогичные эксперименты с очень большими числами, находящимися на границе допустимого диапазона для чисел типов `real` и `double precision`.

```
template1=# SELECT 0.2::real * 10 = 1.0::real;
 ?column?
----------
 f
(1 row)
```

запросы на верхней границе:

```
template1=# SELECT '10e37'::real > '11e37'::real;
 ?column?
----------
 f
(1 row)

template1=#
```

```
template1=# SELECT '1e308'::double precision > '12e307'::double precision;
 ?column?
----------
 f
(1 row)
```

8. Немного усложним определение таблицы из предыдущего задания. Пусть теперь столбец id будет первичным ключом этой таблицы.

```
CREATE TABLE test_serial
( id serial PRIMARY KEY,
  name text
);
```

Теперь выполните следующие команды для добавления строк в таблицу и удаления одной строки из нее. Для пошагового управления этим процессом выполняйте выборку данных из таблицы с помощью команды SELECT после каждой команды вставки или удаления.

```
INSERT INTO test_serial ( name ) VALUES ( 'Вишневая' );
```

Явно зададим значение столбца id:

```
INSERT INTO test_serial ( id, name ) VALUES ( 2, 'Прохладная' );
```

При выполнении этой команды СУБД выдаст сообщение об ошибке. Почему?

```
INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
```

Повторим эту же команду. Теперь все в порядке. Почему?

```
INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
```

Добавим еще одну строку.

```
INSERT INTO test_serial ( name ) VALUES ( 'Зеленая' );
```

А теперь удалим ее же.

```
DELETE FROM test_serial WHERE id = 4;
```

Добавим последнюю строку.

```
INSERT INTO test_serial ( name ) VALUES ( 'Луговая' );
```

Теперь сделаем выборку.

```
SELECT * FROM test_serial;
```

Вы увидите, что в нумерации образовалась «дыра». Это из-за того, что при формировании нового значения из последовательности поиск максимального значения, уже имеющегося в столбце, не выполняется.

```
 id |    name
----+------------
  1 | Вишневая
  2 | Прохладная
  3 | Грушевая
  5 | Луговая
(4 строки)
```

```
template1=# CREATE TABLE test_serial
( id serial PRIMARY KEY,
name text
);
CREATE TABLE
template1=# INSERT INTO test_serial ( name ) VALUES ( 'Вишневая' );
INSERT 0 1
template1=# INSERT INTO test_serial ( id, name ) VALUES ( 2, 'Прохладная' );
INSERT 0 1
template1=# INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
ERROR:  duplicate key value violates unique constraint "test_serial_pkey"
DETAIL:  Key (id)=(2) already exists.
template1=# INSERT INTO test_serial ( name ) VALUES ( 'Грушевая' );
INSERT 0 1
template1=# INSERT INTO test_serial ( name ) VALUES ( 'Зеленая' );
INSERT 0 1
template1=# INSERT INTO test_serial ( name ) VALUES ( 'Луговая' );
INSERT 0 1
template1=# SELECT * FROM test_serial;
 id |    name
----+------------
  1 | Вишневая
  2 | Прохладная
  3 | Грушевая
  4 | Зеленая
  5 | Луговая
(5 rows)
```

```
template1=# DELETE FROM test_serial WHERE id = 4;
DELETE 1
template1=# SELECT * FROM test_serial;
 id |    name
----+------------
  1 | Вишневая
  2 | Прохладная
  3 | Грушевая
  5 | Луговая
(4 rows)

template1=#
```

Ошибка возникает из-за того, что postgresql пытается добавить уже добавленный индекс.

12.* Формат ввода и вывода даты можно изменить с помощью конфигурационного параметра `datestyle`. Значение этого параметра состоит из двух компонентов: первый управляет форматом вывода даты, а второй регулирует порядок следования составных частей даты (год, месяц, день) при вводе и выводе. Текущее значение этого параметра можно узнать с помощью команды SHOW:

```
SHOW datestyle;
```

По умолчанию он имеет такое значение:

```
 DateStyle
-----------
 ISO, DMY
(1 строка)
```

Продемонстрируем влияние этого параметра на работу с типами данных `date` и `timestamp`. Для экспериментов возьмем дату, в которой число (день) превышает 12, чтобы нельзя было день перепутать с номером месяца. Пусть это будет, например, 18 мая 2016 г.

```
SELECT '18-05-2016'::date;
```

Хотя порядок следования составных частей даты задан в виде DMY, т. е. «день, месяц, год», но при выводе он изменяется на «год, месяц, день».

```
    date
------------
 2016-05-18
(1 строка)
```

Попробуем ввести дату в порядке «месяц, день, год»:

```
SELECT '05-18-2016'::date;
```

В ответ получим сообщение об ошибке. Если бы мы выбрали дату, в которой число (день) было бы не больше 12, например, 9, то сообщение об ошибке не было бы сформировано, т. е. мы с такой датой не смогли бы проиллюстрировать влияние значения DMY параметра `datestyle`. Но главное, что в таком случае мы бы просто не заметили допущенной ошибки.

А вот использовать порядок «год, месяц, день» при вводе можно несмотря на то, что параметр `datestyle` предписывает «день, месяц, год». Порядок «год, месяц, день» является универсальным, его можно использовать всегда, независимо от настроек параметра `datestyle`.

```
SELECT '2016-05-18'::date;
```

```
    date
------------
 2016-05-18
(1 строка)
```

Продолжим экспериментирование с параметром `datestyle`. Давайте изменим его значение. Сделать это можно многими способами, но мы упомянем лишь некоторые:

- изменив его значение в конфигурационном файле postgresql.conf, который в нашей инсталляции PostgreSQL, описанной в главе 2, находится в каталоге /usr/local/pgsql/data;
- назначив переменную системного окружения PGDATESTYLE;
- воспользовавшись командой SET.

Сейчас выберем третий способ, а первые два рассмотрим при выполнении других заданий. Поскольку параметр `datestyle` состоит фактически из двух частей, которые можно задавать не только обе сразу, но и по отдельности, изменим только порядок следования составных частей даты, не изменяя формат вывода с `ISO` на какой-либо другой.

```
SET datestyle TO 'MDY';
```

Повторим одну из команд, выполненных ранее. Теперь она должна вызвать ошибку. Почему?

```
SELECT '18-05-2016'::date;
```

А такая команда, наоборот, теперь будет успешно выполнена:

```
SELECT '05-18-2016'::date;
```

Теперь приведите настройку параметра `datestyle` в исходное состояние:

```
SET datestyle TO DEFAULT;
```

Самостоятельно выполните команды `SELECT`, приведенные выше, но замените в них тип `date` на тип `timestamp`. Вы увидите, что дата в рамках типа `timestamp` обрабатывается аналогично типу `date`.

Сейчас изменим сразу обе части параметра `datestyle`:

```
SET datestyle TO 'Postgres, DMY';
```

Проверьте полученный результат с помощью команды `SHOW`.

Самостоятельно выполните команды `SELECT`, приведенные выше, как для значения типа `date`, так и для значения типа `timestamp`. Обратите внимание, что если выбран формат `Postgres`, то порядок следования составных частей даты (день, месяц, год), заданный в параметре `datestyle`, используется не только при вводе значений, но и при выводе. Напомним, что вводом мы считаем команду `SELECT`, а выводом — результат ее выполнения, выведенный на экран.

В документации (см. раздел 8.5.2 «Вывод даты/времени») сказано, что формат вывода даты может принимать значения `ISO`, `Postgres`, `SQL` и `German`. Первые два варианта мы уже рассмотрели. Самостоятельно поэкспериментируйте с двумя оставшимися по той же схеме, по которой вы уже действовали ранее при выполнении этого задания. Можно воспользоваться и стандартными функциями `current_date` и `current_timestamp`.

```
template1=# SET datestyle TO 'DMY';
SET
```

```
template1=# SHOW datestyle;
 DateStyle
-----------
 ISO, DMY
(1 row)

template1=# SELECT '29-09-2023'::timestamp;
      timestamp
---------------------
 2023-09-29 00:00:00
(1 row)
```

```
template1=# SELECT '29-09-2023'::timestamp;
      timestamp
---------------------
 2023-09-29 00:00:00
(1 row)

template1=# SELECT '09-29-2023'::timestamp;
ERROR:  date/time field value out of range: "09-29-2023"
LINE 1: SELECT '09-29-2023'::timestamp;
               ^
HINT:  Perhaps you need a different "datestyle" setting.
```

```
template1=# SET datestyle TO 'Postgres, DMY';
SET
template1=# SHOW datestyle;
   DateStyle
---------------
 Postgres, DMY
(1 row)

template1=# SELECT current_date;
 current_date
--------------
 17-09-2023
(1 row)

template1=# SELECT current_timestamp;
          current_timestamp
-----------------------------------
 Sun 17 Sep 13:53:32.101531 2023 MSK
(1 row)
```

```
template1=# SET datestyle TO 'SQL, DMY';
SET
template1=# SELECT current_date;
 current_date
--------------
 17/09/2023
(1 row)

template1=# SELECT current_timestamp;
       current_timestamp
-------------------------------
 17/09/2023 13:54:21.597424 MSK
(1 row)
```

```
template1=# SET datestyle TO 'German, DMY';
SET
template1=# SHOW datestyle;
  DateStyle
-------------
 German, DMY
(1 row)

template1=# SELECT current_timestamp;
       current_timestamp
-------------------------------
 17.09.2023 13:55:41.869325 MSK
(1 row)

template1=# SELECT current_date;
 current_date
--------------
 17.09.2023
(1 row)
```

15. В документации в разделе 9.8 «Функции форматирования данных» представлены описания множества полезных функций, позволяющих преобразовать в строку данные других типов, например, `timestamp`. Одна из таких функций — `to_char`.

    Приведем несколько команд, иллюстрирующих использование этой функции. Ее первым параметром является форматируемое значение, а вторым — шаблон, описывающий формат, в котором это значение будет представлено при вводе или выводе. Сначала попробуйте разобраться, не обращаясь к документации, в том, что означает второй параметр этой функции в каждой из приведенных команд, а затем проверьте свои предположения по документации.

    ```
    SELECT to_char( current_timestamp, 'mi:ss' );

     to_char
    ---------
     47:43
    (1 строка)
    ```

```
SELECT to_char( current_timestamp, 'dd' );

 to_char
---------
 12
(1 строка)

SELECT to_char( current_timestamp, 'yyyy-mm-dd' );

 to_char
------------
 2017-03-12
(1 строка)
```

Поэкспериментируйте с этой функцией, извлекая из значения типа `timestamp` различные поля и располагая их в нужном вам порядке.

```
template1=# SELECT to_char( current_timestamp, 'hh:mi:ss' );
 to_char
----------
 01:59:35
(1 row)

template1=# SELECT to_char( current_timestamp, 'yyyy-hh:mi:ss' );
    to_char
---------------
 2023-01:59:40
(1 row)

template1=# SELECT to_char( current_timestamp, 'yyyy-dd-hh:mi:ss' );
     to_char
------------------
 2023-17-01:59:44
(1 row)

template1=# SELECT to_char( current_timestamp, 'yyyy-dd-mm-hh:mi:ss' );
       to_char
--------------------
 2023-17-09-01:59:52
(1 row)

template1=# SELECT to_char( current_timestamp, 'yyyy-dd-mm hh:mi:ss' );
       to_char
---------------------
 2023-17-09 01:59:56
(1 row)
```

21. Можно с высокой степенью уверенности предположить, что при прибавлении интервалов к датам и временны́м отметкам PostgreSQL учитывает тот факт, что различные месяцы имеют различное число дней. Но как это реализуется на практике? Например, что получится при прибавлении интервала в 1 месяц к последнему дню января и к последнему дню февраля? Сначала сделайте обоснованные предположения о результатах следующих двух команд, а затем проверьте предположения на практике и проанализируйте полученные результаты:

```
SELECT ( '2016-01-31'::date + '1 mon'::interval ) AS new_date;
SELECT ( '2016-02-29'::date + '1 mon'::interval ) AS new_date;
```

Postgres прибавит 1 месяц к каждой дате с учетом количества дней для следующего месяца.

```
template1=# SELECT ( '2016-01-31'::date + '1 mon'::interval ) AS new_date;
      new_date
---------------------
 2016-02-29 00:00:00
(1 row)

template1=# SELECT ( '2016-02-29'::date + '1 mon'::interval ) AS new_date;
      new_date
---------------------
 2016-03-29 00:00:00
(1 row)
```

Ответы подтвердили данное предположение.

30.* Обратимся к таблице, создаваемой с помощью команды

```
CREATE TABLE test_bool
( a boolean,
  b text
);
```

Как вы думаете, какие из приведенных ниже команд содержат ошибку?

```
INSERT INTO test_bool VALUES ( TRUE, 'yes' );
INSERT INTO test_bool VALUES ( yes, 'yes' );
INSERT INTO test_bool VALUES ( 'yes', true );
INSERT INTO test_bool VALUES ( 'yes', TRUE );
INSERT INTO test_bool VALUES ( '1', 'true' );
INSERT INTO test_bool VALUES ( 1, 'true' );
INSERT INTO test_bool VALUES ( 't', 'true' );
INSERT INTO test_bool VALUES ( 't', truth );
INSERT INTO test_bool VALUES ( true, true );
INSERT INTO test_bool VALUES ( 1::boolean, 'true' );
INSERT INTO test_bool VALUES ( 111::boolean, 'true' );
```

Проверьте свои предположения практически, выполнив эти команды.

```
template1=# CREATE TABLE test_bool
( a boolean,
b text
);
CREATE TABLE
template1=# INSERT INTO test_bool VALUES ( TRUE, 'yes' );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( yes, 'yes' );
ERROR:  column "yes" does not exist
LINE 1: INSERT INTO test_bool VALUES ( yes, 'yes' );
                                       ^
template1=# INSERT INTO test_bool VALUES ( 'yes', true );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( 'yes', TRUE );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( '1', 'true' );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( 1, 'true' );
ERROR:  column "a" is of type boolean but expression is of type integer
LINE 1: INSERT INTO test_bool VALUES ( 1, 'true' );
                                       ^
HINT:  You will need to rewrite or cast the expression.
template1=# INSERT INTO test_bool VALUES ( 't', 'true' );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( 't', truth );
ERROR:  column "truth" does not exist
LINE 1: INSERT INTO test_bool VALUES ( 't', truth );
                                            ^
template1=# INSERT INTO test_bool VALUES ( true, true );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( 1::boolean, 'true' );
INSERT 0 1
template1=# INSERT INTO test_bool VALUES ( 111::boolean, 'true' );
INSERT 0 1
```

Первая вставка без ошибки
Вторая без ошибки
Третья без ошибки
Четвертая с ошибкой
Пятая без ошибки
Шестая с ошибкой
Седьмая без ошибки
Восьмая без ошибки
Девятая без ошибки

33.* В разделе документации 8.15 «Массивы» сказано, что массивы могут быть многомерными и в них могут содержаться значения любых типов. Давайте сначала рассмотрим одномерные массивы *текстовых* значений.

Предположим, что пилоты авиакомпании имеют возможность высказывать свои пожелания насчет конкретных блюд, из которых должен состоять их обед во время полета. Для учета пожеланий пилотов необходимо модифицировать таблицу `pilots`, с которой мы работали в разделе 4.5.

```
CREATE TABLE pilots
( pilot_name text,
  schedule integer[],
  meal text[]
);
```

Добавим строки в таблицу:

```
INSERT INTO pilots
  VALUES ( 'Ivan', '{ 1, 3, 5, 6, 7 }'::integer[],
           '{ "сосиска", "макароны", "кофе" }'::text[]
         ),
         ( 'Petr', '{ 1, 2, 5, 7 }'::integer [],
           '{ "котлета", "каша", "кофе" }'::text[]
         ),
         ( 'Pavel', '{ 2, 5 }'::integer[],
           '{ "сосиска", "каша", "кофе" }'::text[]
         ),
         ( 'Boris', '{ 3, 5, 6 }'::integer[],
           '{ "котлета", "каша", "чай" }'::text[]
         );
```

```
INSERT 0 4
```

Обратите внимание, что каждое из текстовых значений, включаемых в литерал массива, заключается в двойные кавычки, а в качестве типа данных указывается `text[]`.

Вот что получилось:

```
SELECT * FROM pilots;
```

```
 pilot_name |  schedule   |         meal
------------+-------------+-------------------------
 Ivan       | {1,3,5,6,7} | {сосиска,макароны,кофе}
 Petr       | {1,2,5,7}   | {котлета,каша,кофе}
 Pavel      | {2,5}       | {сосиска,каша,кофе}
 Boris      | {3,5,6}     | {котлета,каша,чай}
(4 строки)
```

Давайте получим список пилотов, предпочитающих на обед сосиски:

```
SELECT * FROM pilots WHERE meal[ 1 ] = 'сосиска';
```

```
 pilot_name |   schedule   |         meal
------------+-------------+-------------------------
 Ivan       | {1,3,5,6,7} | {сосиска,макароны,кофе}
 Pavel      | {2,5}       | {сосиска,каша,кофе}
(2 строки)
```

Предположим, что руководство авиакомпании решило, что пища пилотов должна быть разнообразной. Оно позволило им выбрать свой рацион на каждый из четырех дней недели, в которые пилоты совершают полеты. Для нас это решение руководства выливается в необходимость модифицировать таблицу, а именно: столбец `meal` теперь будет содержать двумерные массивы. Определение этого столбца станет таким: `meal text[][]`.

Задание. Создайте новую версию таблицы и соответственно измените команду INSERT, чтобы в ней содержались литералы *двумерных* массивов. Они будут выглядеть примерно так:

```
'{ { "сосиска", "макароны", "кофе" },
   { "котлета", "каша", "кофе" },
   { "сосиска", "каша", "кофе" },
   { "котлета", "каша", "чай" } }'::text[][]
```

Сделайте ряд выборок и обновлений строк в этой таблице. Для обращения к элементам двумерного массива нужно использовать два индекса. Не забывайте, что по умолчанию номера индексов начинаются с единицы.

```
template1=# CREATE TABLE pilots (
template1(# pilot_name text,
schedule integer[],
meal text[][]
);
CREATE TABLE
template1=# SELECT * FROM pilots;
 pilot_name | schedule | meal
------------+----------+------
(0 rows)
```

```
template1=# INSERT INTO pilots
VALUES ( 'Ivan', '{ 1, 3, 5, 6, 7 }'::integer[],
'{ { "сосиска", "макароны", "кофе" },
{ "котлета", "каша", "кофе" },
{ "сосиска", "каша", "кофе" },
{ "котлета", "каша", "чай" } }'::text[][]);
INSERT 0 1
template1=# SELECT * FROM pilots;
 pilot_name |  schedule   |                                          meal
------------+-------------+-------------------------------------------------------------------------------------
 Ivan       | {1,3,5,6,7} | {{сосиска,макароны,кофе},{котлета,каша,кофе},{сосиска,каша,кофе},{котлета,каша,чай}}
(1 row)

template1=# SELECT pilot_name, schedule, meal[1][1] FROM pilots;
 pilot_name |  schedule   |  meal
------------+-------------+---------
 Ivan       | {1,3,5,6,7} | сосиска
(1 row)
```

```
                                                  ^
template1=# SELECT pilot_name, schedule, meal[1][1:3] FROM pilots;
 pilot_name |  schedule   |            meal
------------+-------------+----------------------------
 Ivan       | {1,3,5,6,7} | {{сосиска,макароны,кофе}}
(1 row)
```

35. Изучая приемы работы с типами JSON, можно, как и в случае с массивами, пользоваться способностью команды SELECT обходиться без создания таблиц.

    Покажем лишь один пример. Добавить новый ключ и соответствующее ему значения в уже существующий объект можно оператором ||:

```
SELECT '{ "sports": "хоккей" }'::jsonb || '{ "trips": 5 }'::jsonb;

              ?column?
---------------------------------
 {"trips": 5, "sports": "хоккей"}
(1 строка)
```

    Для работы с типами JSON предусмотрено много различных функций и операторов, представленных в разделе документации 9.15 «Функции и операторы JSON». Самостоятельно ознакомьтесь с ними, используя описанную технологию работы с командой SELECT.

```
template1=# SELECT '{ "sports": "хоккей" }'::jsonb || '{ "trips": 5 }'::jsonb;
             ?column?
---------------------------------
 {"trips": 5, "sports": "хоккей"}
(1 row)
```

```
template1=# SELECT '{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb;
 ?column?
----------
 t
(1 row)
```

```
template1=# SELECT json_object('{a, 1, b, "def", c, 3.5}');
              json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)
```

```
template1=# SELECT jsonb_path_query_array('{"x": "20", "y": 32}', '$.keyvalue()');
                          jsonb_path_query_array
-----------------------------------------------------------------------------
 [{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]
(1 row)
```