Контрольные вопросы и задания

1. Создайте таблицу, содержащую атрибут типа `numeric(precision, scale)`. Пусть это будет таблица, содержащая результаты каких-то измерений.

   Команда может быть, например, такой:

   ```
   CREATE TABLE test_numeric
   ( measurement numeric(5, 2),
     description text
   );
   ```

   Попробуйте с помощью команды `INSERT` продемонстрировать округление вводимого числа до той точности, которая задана при создании таблицы.

   Подумайте, какая из следующих команд вызовет ошибку и почему? Проверьте свои предположения, выполнив эти команды.

   ```
   INSERT INTO test_numeric
     VALUES ( 999.9999, 'Какое-то измерение ' );
   INSERT INTO test_numeric
     VALUES ( 999.9009, 'Еще одно измерение' );
   INSERT INTO test_numeric
     VALUES ( 999.1111, 'И еще измерение' );
   INSERT INTO test_numeric
     VALUES ( 998.9999, 'И еще одно' );
   ```

   Продемонстрируйте генерирование ошибки при попытке ввода числа, количество цифр в котором слева от десятичной точки (запятой) превышает допустимое.

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.

3. Тип данных `numeric` поддерживает специальное значение NaN, которое означает «не число» (not a number). В документации утверждается, что значение NaN считается равным другому значению NaN, а также что значение NaN считается бóльшим любого другого «нормального» значения, т. е. не-NaN. Проверьте эти утверждения с помощью SQL-команды `SELECT`.

   В качестве примера приведем команду:

   ```
   SELECT 'NaN'::numeric > 10000;
   ```

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

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`.

5. Типы данных `real` и `double precision` поддерживают специальные значения Infinity (бесконечность) и −Infinity (отрицательная бесконечность). Проверьте с помощью SQL-команды `SELECT` ожидаемые свойства этих значений. Например, сравните Infinity с наибольшим значением, которое допускается для типа `double precision` (можно использовать сокращенное написание Inf):

```
SELECT 'Inf'::double precision > 1E+308;
```

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

Выполните аналогичный запрос для наименьшего возможного значения типа `double precision`.

6. Типы данных `real` и `double precision` поддерживают специальное значение NaN, которое означает «не число» (not a number).

   В математике существует такое понятие, как *неопределенность*. В качестве одного из ее вариантов служит результат операции умножения нуля на бесконечность. Посмотрите, что выдаст в результате PostgreSQL:

   ```
   SELECT 0.0 * 'Inf'::real;
   ```

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

   В документации утверждается, что значение NaN считается равным другому значению NaN, а также что значение NaN считается бóльшим любого другого «нормального» значения, т. е. не-NaN. Проверьте эти утверждения с помощью SQL-команды `SELECT`.

   Например, сравните значения NaN и Infinity.

   ```
   select 'NaN'::real > 'Inf'::real;
   ```

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

7. Тип `serial` может применяться для столбцов, содержащих числовые значения, которые должны быть уникальными в пределах таблицы, например, идентификаторы каких-то объектов. В качестве иллюстрации применения типа `serial` предложим таблицу, содержащую наименования улиц и площадей:

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

   Введите несколько строк. Обратите внимание, что значение для столбца `id` указывать не обязательно (и даже не нужно). Но поскольку мы задаем значения не для всех столбцов, имеющихся в таблице, мы должны указать в команде `INSERT` не только список значений, но и список столбцов. Конечно, в данном простом случае эти списки состоят лишь из одного элемента.

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

Сделайте выборку данных из таблицы, вы увидите, что значения столбца `id` имеют последовательные значения, начиная с 1.

Давайте проведем эксперимент со столбцом `id`. Выполните команду `INSERT`, в которой укажите явное значение столбца `id`:

```
INSERT INTO test_serial ( id, name ) VALUES ( 10, 'Прохладная' );
```

А теперь добавьте еще одну строку, но уже не указывая явно значение для столбца `id` (как мы поступали в предыдущих командах):

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

Вы увидите, что явное задание значения для столбца `id` не влияет на автоматическое генерирование значений этого столбца.

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)
```

9. Какой календарь используется в PostgreSQL для работы с датами: юлианский или григорианский?

10. Каждый тип данных из группы «дата/время» имеет ограничение на минимальное и максимальное допустимое значение. Найдите в документации в разделе 8.5 «Типы даты/времени» эти значения и подумайте, почему они таковы.

11. Типы `timestamp`, `time` и `interval` позволяют задать точность ввода и вывода значений. Точность предписывает количество десятичных цифр в поле секунд. Проиллюстрируем эту возможность на примере типа `time`, выполнив три запроса: в первом запросе вообще не используем параметр точности, во втором назначим его равным 0, в третьем запросе сделаем его равным 3.

```
SELECT current_time;
```

```
      timetz
--------------------
 19:46:14.584641+03
(1 строка)
```

```
SELECT current_time::time( 0 );
```

```
   time
----------
 19:39:45
(1 строка)
```

```
SELECT current_time::time( 3 );
```

```
     time
--------------
 19:39:54.085
(1 строка)
```

Выполните подобные команды для типов `timestamp` и `interval`.

Тип `date` такой возможности — задавать точность — не имеет. Как вы думаете, почему?

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`.

13. Установить новое значение параметра `datestyle` можно с помощью создания переменной системного окружения `PGDATESTYLE`. Назначить эту переменную можно в конфигурационных файлах операционной системы. Но если нам нужно сделать это только на время текущего сеанса работы клиентской программы, например утилиты psql, то можно ввести значение этой переменной непосредственно в командной строке:

    ```
    PGDATESTYLE="Postgres" psql -d test -U имя-пользователя
    ```

    Проделайте эти действия, а затем уже из командной строки утилиты psql проверьте текущее значение параметра `datestyle` с помощью команды `SHOW`.

14. Назначить значение параметра `datestyle` можно в конфигурационном файле postgresql.conf, который находится в каталоге /usr/local/pgsql/data. Предварительно сохраните текущую (корректно работающую) версию этого файла, а затем измените в нем значение параметра `datestyle`, например, на `Postgres, YMD`. Перезапустите сервер PostgreSQL, чтобы изменения вступили в силу.

    Для проверки полученного результата выполните несколько команд `SELECT`, например:

    ```
    SELECT '05-18-2016'::timestamp;
    SELECT current_timestamp;
    ```

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.

16. При выполнении приведения типа данных производится проверка значения на допустимость. Попробуйте ввести недопустимое значение даты, например, 29 февраля в невисокосном году.

```
SELECT 'Feb 29, 2015'::date;
```

Получите сообщение об ошибке.

17. При выполнении приведения типа данных производится проверка значения на допустимость. Попробуйте ввести недопустимое значение времени, например, с нарушением формата.

```
SELECT '21:15:16:22'::time;

ОШИБКА: неверный синтаксис для типа time: "21:15:16:22"
СТРОКА 1: select '21:15:16:22'::time;
                 ^
```

18. Как вы думаете, значение какого типа будет получено при вычитании одной даты из другой?

Например:

```
SELECT ( '2016-09-16'::date - '2016-09-01'::date );
```

Сначала попробуйте получить ответ, рассуждая логически, а затем проверьте на практике в утилите psql.

19. С типами даты и времени можно выполнять различные арифметические операции. Как правило, их применение является интуитивно понятным. Выполните следующую команду и проанализируйте результат.

    ```
    SELECT ( '20:34:35'::time - '19:44:45'::time );
    ```

    А теперь попробуйте предположить, какой результат будет получен, если в этой команде знак «минус» заменить на знак «плюс»? Проверьте ваши предположения с помощью утилиты psql. Подробное описание всех допустимых арифметических операций с датами и временем приведено в документации в разделе 9.9 «Операторы и функции даты/времени».

20. Значение типа `interval` можно получить при вычитании одной временно́й отметки из другой, например:

    ```
    SELECT ( current_timestamp - '2016-01-01'::timestamp )
    AS new_date;
    ```

    ```
           new_date
    ------------------------
     278 days 00:10:33.33236
    (1 строка)
    ```

    А что получится, если прибавить интервал к временно́й отметке? Сначала попробуйте дать ответ, не прибегая к помощи утилиты psql, а затем проверьте свой ответ с помощью этой утилиты. Например, прибавим интервал длительностью в 1 месяц к текущей к временно́й отметке:

    ```
    SELECT ( current_timestamp + '1 mon'::interval ) AS new_date;
    ```

    В этой команде с помощью ключевого слова AS мы назначили псевдоним для того столбца, который будет выведен в результате. Выполните эту же команду, убрав псевдоним, и найдите отличия.

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;
    ```

22. Форматом ввода и вывода интервалов управляет параметр `intervalstyle`. Его можно изменить с помощью способов, аналогичных тем, что были описаны выше для параметра `datestyle`. Самостоятельно поэкспериментируйте с различными значениями параметра `intervalstyle` аналогично тому, как вы это делали с параметром `datestyle`. Используйте раздел 8.5 «Типы даты/времени» в документации.

    Напомним, что вернуть исходное значение этого параметра в psql можно с помощью команды

    ```
    SET intervalstyle TO DEFAULT;
    ```

23. Выполните следующие две команды и объясните различия в выведенных результатах:

    ```
    SELECT ( '2016-09-16'::date - '2015-09-01'::date );
    SELECT ( '2016-09-16'::timestamp - '2015-09-01'::timestamp );
    ```

24. Выполните следующие две команды и объясните различия в выведенных результатах:

    ```
    SELECT ( '20:34:35'::time - 1 );
    SELECT ( '2016-09-16'::date - 1 );
    ```

    Почему при выполнении первой команды возникает ошибка? Как можно модифицировать эту команду, чтобы ошибка исчезла?

    Для получения полной информации обратитесь к разделу 9.9 «Операторы и функции даты/времени» документации.

25. Значения временны́х отметок можно усекать с той или иной точностью с помощью функции `date_trunc`. Например, с помощью следующей команды можно «отрезать» дробную часть секунды:

    ```
    SELECT ( date_trunc( 'sec',
                         timestamp '1999-11-27 12:34:56.987654' ) );

         date_trunc
    ---------------------
     1999-11-27 12:34:56
    (1 строка)
    ```

Напомним, что в данной команде используется операция приведения типа.

Выполните эту команду, последовательно указывая в качестве первого параметра значения microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium (которые обозначают соответственно микросекунды, миллисекунды, секунды, минуты, часы, дни, недели, месяцы, годы, десятилетия, века и тысячелетия). Допустимы сокращения sec, min, mon, dec, cent, mil.

Обратите внимание, что результирующее значение получается не путем округления исходного значения, а именно путем отбрасывания более мелких единиц. При этом поля времени (часы, минуты и секунды) заменяются нулями, а поля даты (годы, месяцы и дни) — заменяются цифрами 01. Однако при использовании параметра week картина получается более интересная.

26. Функция `date_trunc` может работать не только с данными типа `timestamp`, но также и с данными типа `interval`. Самостоятельно ознакомьтесь с этими возможностями по документации (см. раздел 9.9 «Операторы и функции даты/времени»).

27. Весьма полезной является функция `extract`. С ее помощью можно извлечь значение отдельного поля из временно́й отметки `timestamp`. Наименование поля задается в первом параметре. Эти наименования такие же, что и для функции `date_trunc`. Выполните следующую команду

```
SELECT extract(
  'microsecond' from  timestamp '1999-11-27 12:34:56.123459'
);
```

Она выводит не просто значение поля микросекунд, т. е. 123459, а дополнительно преобразует число секунд в микросекунды и добавляет значение поля микросекунд.

```
 date_part
-----------
 56123459
(1 строка)
```

Выполните эту команду, последовательно указывая в качестве первого параметра значения microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium. Можно использовать сокращения этих наименований, которые приведены в предыдущем задании.

Обратите внимание, что в ряде случаев выводится не просто конкретное поле (фрагмент) из временно́й отметки, а некоторый продукт переработки этого поля. Например, если в качестве первого параметра функции `extract` в вышеприведенной команде указать cent (век), то мы получим в ответ не 19 (что и было бы буквальным значением поля «век»), а 20, поскольку 1999 год принадлежит двадцатому веку.

28. Функция `extract` может работать не только с данными типа `timestamp`, но также и с данными типа `interval`. Самостоятельно ознакомьтесь с этими возможностями по документации (см. раздел 9.9 «Операторы и функции даты/времени»).

29.* В тексте главы мы создавали таблицу с помощью команды

```
CREATE TABLE databases
( is_open_source boolean,
  dbms_name text
);
```

и заполняли ее данными.

```
INSERT INTO databases VALUES ( TRUE, 'PostgreSQL' );
INSERT INTO databases VALUES ( FALSE, 'Oracle' );
INSERT INTO databases VALUES ( TRUE, 'MySQL' );
INSERT INTO databases VALUES ( FALSE, 'MS SQL Server' );
```

Как вы думаете, являются ли все приведенные ниже команды равнозначными в смысле результатов, получаемых с их помощью?

```
SELECT * FROM databases WHERE NOT is_open_source;
SELECT * FROM databases WHERE is_open_source <> 'yes';
SELECT * FROM databases WHERE is_open_source <> 't';
SELECT * FROM databases WHERE is_open_source <> '1';
SELECT * FROM databases WHERE is_open_source <> 1;
```

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.

31.* Пусть в таблице `birthdays` хранятся даты рождения какой-то группы людей. Создайте эту таблицу с помощью команды

```
CREATE TABLE birthdays
( person text NOT NULL,
birthday date NOT NULL );
```

Добавьте в нее несколько строк, например:

```
INSERT INTO birthdays VALUES ( 'Ken Thompson', '1955-03-23' );
INSERT INTO birthdays VALUES ( 'Ben Johnson', '1971-03-19' );
INSERT INTO birthdays VALUES ( 'Andy Gibson', '1987-08-12' );
```

Давайте выберем из таблицы `birthdays` строки для всех людей, родившихся в каком-то конкретном месяце, например, в марте:

```
SELECT * FROM birthdays
  WHERE extract( 'mon' from birthday ) = 3;
```

В этой команде в вызове функции `extract` имеет место неявное приведение типов, т. к. ее вторым параметром должно быть значение типа `timestamp`. Полагаться на неявное приведение типов можно не всегда.

```
   person     |  birthday
--------------+------------
 Ken Thompson | 1955-03-23
 Ben Johnson  | 1971-03-19
(2 строки)
```

Если нам потребуется выяснить, кто из этих людей достиг возраста, скажем, 40 лет на момент выполнения запроса, то команда может быть такой (в последнем столбце показана дата достижения возраста 40 лет):

```
SELECT *, birthday + '40 years'::interval
  FROM birthdays
  WHERE birthday + '40 years'::interval < current_timestamp;
```

```
    person    |  birthday  |      ?column?
--------------+------------+---------------------
 Ken Thompson | 1955-03-23 | 1995-03-23 00:00:00
 Ben Johnson  | 1971-03-19 | 2011-03-19 00:00:00
(2 строки)
```

Можно заменить `current_timestamp` на `current_date`:

```
SELECT *, birthday + '40 years'::interval
  FROM birthdays
  WHERE birthday + '40 years'::interval < current_date;
```

А вот если мы захотим определить точный возраст каждого человека на текущий момент времени, то как получить этот результат?

Первый вариант таков:

```
SELECT *, ( current_date::timestamp - birthday::timestamp )::interval
FROM birthdays;
```

```
    person    |  birthday  |  interval
--------------+------------+------------
 Ken Thompson | 1955-03-23 | 22477 days
 Ben Johnson  | 1971-03-19 | 16637 days
 Andy Gibson  | 1987-08-12 | 10647 days
(3 строки)
```

Этот вариант не дает результата, представленного в удобной форме: он показывает возраст в днях, а для пересчета числа дней в число лет нужны дополнительные действия. Хотя, наверное, возможны ситуации, когда требуется определить возраст именно в днях.

В PostgreSQL предусмотрена специальная функция, позволяющая решить нашу задачу простым способом. Самостоятельно найдите ее описание в документации (см. раздел 9.9 «Операторы и функции даты/времени») и напишите команду с ее использованием.

32. Изучая приемы работы с массивами, можно, как и в других случаях, пользоваться способностью команды SELECT обходиться без создания таблиц. Покажем лишь два примера.

    Для объединения (конкатенации) массивов служит функция array_cat:

```
SELECT array_cat( ARRAY[ 1, 2, 3 ], ARRAY[ 3, 5 ] );
```

```
 array_cat
-------------
 {1,2,3,3,5}
(1 строка)
```

    Удалить из массива элементы, имеющие указанное значение, можно таким образом:

```
SELECT array_remove( ARRAY[ 1, 2, 3 ], 3 );
```

```
 array_remove
--------------
 {1,2}
(1 строка)
```

    Для работы с массивами предусмотрено много различных функций и операторов, представленных в разделе документации 9.18 «Функции и операторы для работы с массивами». Самостоятельно ознакомьтесь с ними, используя описанную технологию работы с командой SELECT.

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.

34. В тексте раздела 4.6 мы выполняли обновление JSON-объекта с помощью функции `jsonb_set`: добавляли значение в массив. Для обновления скалярных значений, например, по ключу `trips`, можно сделать так:

```
UPDATE pilot_hobbies
  SET hobbies = jsonb_set( hobbies, '{ trips }', '10' )
  WHERE pilot_name = 'Pavel';
```

```
UPDATE 1
```

Второй параметр функции — это путь в пределах JSON-объекта. Он теперь представляет собой лишь имя ключа. Однако его необходимо заключить в фигурные скобки. Третий параметр — это новое значение. Хотя оно числовое, но все равно требуется записать его в одинарных кавычках.

```
SELECT pilot_name, hobbies->'trips' AS trips FROM pilot_hobbies;
```

```
 pilot_name | trips
------------+-------
 Ivan       | 3
 Petr       | 2
 Boris      | 0
 Pavel      | 10
(4 строки)
```

Задание. Самостоятельно выполните изменение значения по ключу `home_lib` в одной из строк таблицы.

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`.

36.* Объекты JSON в разных строках таблицы могут иметь различные наборы ключей. Добавьте дополнительный ключ и соответствующее ему значение в JSON-объект какой-нибудь строки таблицы `pilots`. Используйте оператор `||`.

37. Объекты JSON позволяют не только добавлять в них новые ключи, но также и удалять из них ключи существующие. Удалите один из ключей из JSON-объекта какой-нибудь строки таблицы `pilots`. Соответствующее ему значение будет также удалено, т. к. без ключа оно не может существовать. Воспользуйтесь оператором `-`.