

На вашем компьютере уже должна быть развернута база данных demo.

- Войдите в систему как пользователь postgres:

```
su - postgres
```

- Должен быть запущен сервер баз данных PostgreSQL.

```
pg_ctl start -D /usr/local/pgsql/data -l postgres.log
```

- Для проверки запуска сервера выполните команду

```
pg_ctl status -D /usr/local/pgsql/data
```

или

```
ps -ax | grep postgres | grep -v grep
```

- Запустите утилиту psql и подключитесь к базе данных demo

```
psql -d demo -U postgres (можно просто psql -d demo)
```

- Для останова сервера баз данных PostgreSQL служит команда

```
pg_ctl stop -D /usr/local/pgsql/data -l postgres.log
```

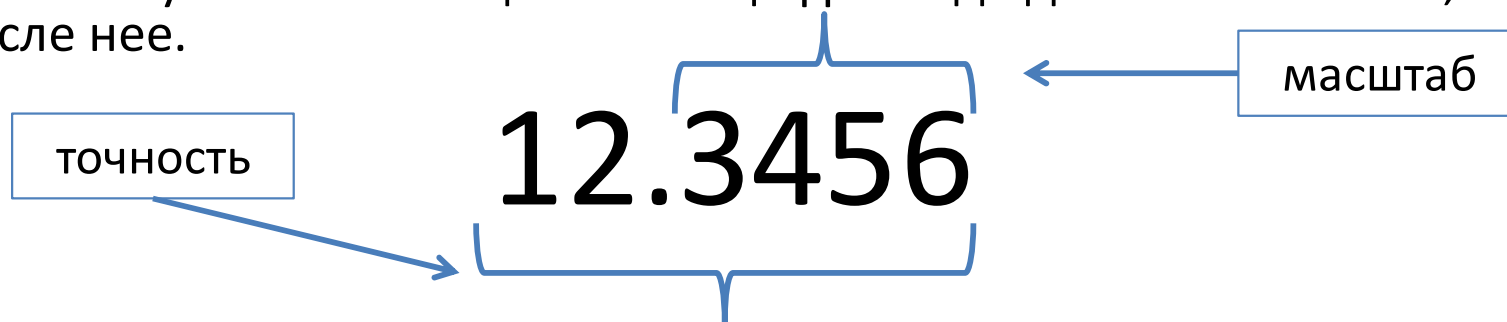
- PostgreSQL имеет очень разнообразный набор встроенных типов данных, т. е. тех типов, которые СУБД предоставляет в распоряжение пользователя, как говорят, по умолчанию.
- Полная информация о типах данных в главе 8 документации:
<https://postgrespro.ru/docs/postgresql/10/datatype>
- Пользователь имеет возможность создавать и свои **собственные типы данных**, которые затем можно включить в систему и использовать их так же, как и встроенные.
- Такая возможность адаптации системы типов данных к конкретным ситуациям является одной из отличительных черт PostgreSQL.

2.1. Числовые типы

- целочисленные типы
- числа фиксированной точности
- типы данных с плавающей точкой
- последовательные типы (serial)

- В составе целочисленных типов находятся следующие представители: **smallint**, **integer**, **bigint**. Если атрибут таблицы имеет один из этих типов, то он позволяет хранить *только целочисленные* данные.
- При этом перечисленные типы различаются по количеству байтов, выделяемых для хранения данных.
- В PostgreSQL существуют псевдонимы для этих стандартизированных имен типов, а именно: **int2**, **int4** и **int8**.
- Число байтов отражается в имени типа.
- При выборе конкретного целочисленного типа принимают во внимание диапазон допустимых значений и затраты памяти.
- Зачастую тип `integer` считается оптимальным выбором с точки зрения достижения компромисса между этими показателями.

- Представлены двумя типами — **numeric** и **decimal**. Однако они являются идентичными по своим возможностям. Поэтому мы будем проводить изложение на примере типа **numeric**.
- Для задания значения этого типа используются два базовых понятия: **масштаб (scale)** и **точность (precision)**.
- Масштаб показывает число значащих цифр, стоящих справа от десятичной точки (запятой).
- Точность указывает общее число цифр как до десятичной точки, так и после нее.



- Например, у числа 12.3456 точность составляет 6 цифр, а масштаб — 4 цифры.
- Параметры этого типа данных указываются в скобках: `numeric(точность, масштаб)`. Например, `numeric(6, 2)`.

- Главное достоинство — это обеспечение точных результатов при выполнении вычислений, *когда это, конечно, возможно в принципе*. Это оказывается возможным при выполнении сложения, вычитания и умножения.
- Числа типа `numeric` могут хранить очень большое количество цифр: 131072 цифры — до десятичной точки (запятой), 16383 — после точки.
- Однако нужно учитывать, что такая точность достигается за счет *замедления вычислений* по сравнению с целочисленными типами и типами с плавающей запятой. При этом для хранения числа затрачивается *больше памяти*, чем в случае целых чисел.
- Данный тип следует выбирать для хранения *денежных сумм*, а также в других случаях, когда требуется гарантировать точность вычислений.

- Представителями типов данных с плавающей точкой являются **real** и **double precision**. Они представляют собой реализацию стандарта IEEE «Standard 754 for Binary Floating-Point Arithmetic».
- Тип данных **real** может представить числа в диапазоне, как минимум, от $1E-37$ до $1E+37$ с точностью не меньше 6 десятичных цифр.
- Тип **double precision** имеет диапазон значений примерно от $1E-307$ до $1E+308$ с точностью не меньше 15 десятичных цифр.
- При попытке записать в такой столбец слишком большое или слишком маленькое значение будет генерироваться ошибка.
- Если точность вводимого числа *выше допустимой*, то будет иметь место *округление* значения.
- А вот при вводе *очень маленьких чисел*, которые невозможно представить значениями, отличными от нуля, будет генерироваться ошибка потери значимости, или исчезновения значащих разрядов (**an underflow error**).

- При работе с числами таких типов нужно помнить, что сравнение двух чисел с плавающей точкой на предмет равенства их значений может привести к неожиданным результатам. Например:
SELECT 0.1::real * 10 = 1.0::real;
?column?

f
(1 строка)
- В дополнение к обычным числам эти типы данных поддерживают и специальные значения **Infinity** (бесконечность), **-Infinity** (отрицательная бесконечность) и **NaN** (не число).
- PostgreSQL поддерживает также тип данных **float**, определенный в стандарте SQL. В объявлении типа может использоваться параметр:
float(p)
- Если его значение лежит в диапазоне от 1 до 24, то это будет равносильно использованию типа **real**, а если же значение лежит в диапазоне от 25 до 53, то это будет равносильно использованию типа **double precision**. Если же при объявлении типа **float** параметр не используется, то это также будет равносильно использованию типа **double precision**.

- Этот тип фактически реализован не как настоящий тип, а просто как удобная замена целой группы SQL-команд. Тип **serial** удобен в тех случаях, когда требуется в какой-либо столбец вставлять уникальные целые значения, например, значения суррогатного первичного ключа.
- Синтаксис для создания столбца типа serial таков:
CREATE TABLE tablename (colname SERIAL);

Эта команда эквивалентна следующей группе команд:

```
CREATE SEQUENCE tablename_colname_seq;  
CREATE TABLE tablename  
( colname integer NOT NULL  
    DEFAULT nextval( 'tablename_colname_seq' )  
);  
ALTER SEQUENCE tablename_colname_seq  
OWNED BY tablename.colname;
```

- Для пояснения вышеприведенных команд нам придется немного забежать вперед.
- Одним из видов объектов в базе данных являются так называемые **последовательности**. Это, по сути, *генераторы уникальных целых чисел*. Для работы с этими последовательностями-генераторами используются специальные функции. Одна из них — это функция **nextval**, которая как раз и получает очередное число из последовательности, имя которой указано в качестве параметра функции.
- В команде CREATE TABLE ключевое слово DEFAULT предписывает, чтобы СУБД использовала в качестве значения по умолчанию то значение, которое формирует функция nextval. Поэтому если в команде вставки строки в таблицу INSERT INTO не будет передано значение для поля типа serial, то СУБД обратится к услугам этой функции.

- В том случае, когда в таблице поле типа serial является суррогатным первичным ключом, тогда нет необходимости указывать явное значение для вставки в это поле.
- Кроме типа serial существуют еще два аналогичных типа: **bigserial** и **smallserial**. Им фактически, за кадром, соответствуют типы bigint и smallint.
- Практический совет. При выборе конкретного последовательного типа нужно учитывать предполагаемое число строк в таблице и *частоту удаления и вставки* строк, поскольку даже для небольшой таблицы может потребоваться большой диапазон, если операции удаления и вставки строк выполняются часто.

2.2. Символьные (строковые) типы

- Стандартные представители строковых типов — это **character varying(n)** и **character(n)**, где параметр указывает максимальное число символов в строке, которую можно сохранить в столбце такого типа.
- При работе с многобайтовыми кодировками символов, например, UTF-8, нужно учитывать, что речь идет именно о *символах*, а не о байтах.
- Если сохраняемая строка символов будет короче, чем указано в определении типа, то значение типа **character** будет *дополнено* пробелами до требуемой длины, а значение типа **character varying** будет сохранено так, как есть.
- Типы **character varying(n)** и **character(n)** имеют псевдонимы **varchar(n)** и **char(n)** соответственно. На практике, как правило, используют именно эти краткие псевдонимы.

- PostgreSQL дополнительно предлагает еще один символьный тип — **text**. В столбец этого типа можно ввести сколь угодно большое значение, конечно, в пределах, установленных при компиляции исходных текстов СУБД.
- Практический совет. Документация рекомендует использовать типы `text` и `varchar`, поскольку такое отличительное свойство типа `character`, как дополнение значений пробелами, на практике почти не востребовано. В PostgreSQL обычно используется тип `text`.
- Константы символьных типов в SQL-командах заключаются в одинарные кавычки:

```
SELECT 'PostgreSQL' ;
```

```
?column?
```

```
-----
```

```
PostgreSQL
```

```
(1 строка)
```

В том случае, когда в константе содержится символ одинарной кавычки или обратной косой черты, их необходимо удваивать. Например:

```
SELECT 'PGDAY' '17' ;
```

```
?column?
```

```
-----
```

```
PGDAY'17
```

```
(1 строка)
```

Можно использовать символы «\$» в качестве ограничителей. Это расширение, предлагаемое PostgreSQL. При этом уже не нужно удваивать никакие символы, содержащиеся в самой константе: ни одинарные кавычки, ни символы обратной косой черты. Например:

```
SELECT $$PGDAY'17$$;
```

```
?column?
```

```
-----
```

```
PGDAY'17
```

```
(1 строка)
```


PostgreSQL предлагает еще одно расширение стандарта SQL.

Например, для включения в константу символа новой строки «\n» нужно сделать так:

```
SELECT E'PGDAY\n17';
```

```
?column?
```

```
-----
```

```
PGDAY  +
```

```
17
```

```
(1 строка)
```



А для включения в содержимое константы символа обратной кавычки можно либо удвоить ее, либо сделать так:

```
SELECT E'PGDAY\'17';
```

```
?column?
```

```
-----
```

```
PGDAY'17
```

```
(1 строка)
```



2.3. Типы «дата/время»

- PostgreSQL поддерживает все типы данных, предусмотренные стандартом SQL для даты и времени.
- Даты обрабатываются в соответствии с григорианским календарем.
- Для этих типов данных предусмотрены определенные форматы для ввода значений и для вывода. Причем, эти форматы могут *не совпадать*.
- Важно помнить, что при вводе значений их нужно заключать в *одинарные кавычки*, как и текстовые строки.

- Рекомендуемый стандартом ISO 8601 формат ввода дат таков: «yyyy-mm-dd», где символы «y», «m» и «d» обозначают цифру года, месяца и дня соответственно.
- PostgreSQL позволяет использовать и другие форматы для ввода, например: «Sep 12, 2016», что означает 12 сентября 2016 года.
- При выводе значений PostgreSQL использует формат по умолчанию, если не предписан другой формат. По умолчанию используется формат, рекомендуемый стандартом ISO 8601: «yyyy-mm-dd».

```
SELECT '2016-09-12'::date;
```

```
      date
```

```
-----
```

```
2016-09-12
```

```
(1 строка)
```



операция приведения типа

- А в следующем примере используется другой формат ввода, но формат вывода остается тот же самый, поскольку мы его не изменяли:

```
SELECT 'Sep 12, 2016'::date;
```

```
      date
```

```
-----
```

```
2016-09-12
```

```
(1 строка)
```



операция приведения типа

Для получения значения текущей даты:

```
SELECT current_date;
```

```
date
```

```
-----
```

```
2016-09-21
```

```
(1 строка)
```

Если нам требуется вывести дату в другом формате, то для разового преобразования формата можно использовать функцию **to_char**, например:

```
SELECT to_char( current_date, 'dd-mm-yyyy' );
```

СУБД выведет:

```
to_char
```

```
-----
```

```
21-09-2016
```

```
(1 строка)
```

Два типа данных: **time** и **time with time zone**. Первый из них хранит только время суток, а второй — дополнительно — еще и часовой пояс. Однако документация на PostgreSQL *не рекомендует* использовать тип **time with time zone**, поскольку смещение (offset), соответствующее конкретному часовому поясу, может зависеть от даты перехода на летнее время и обратно, но в этом типе дата отсутствует.

При вводе значений времени допустимы различные форматы, например:

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

При выводе СУБД дополнит введенное значение, в котором присутствуют только часы и минуты, секундами.

```
time
-----
21:15:00
(1 строка)
```

Мы опять использовали операцию приведения типа «::».

```
SELECT '25:15'::time;
```

Получим такое сообщение об ошибке:

ОШИБКА: значение поля типа date/time вне диапазона:

"25:15"

СТРОКА 1: select '25:15'::time;

^

А если число секунд недопустимое, то опять получим сообщение об ошибке.

```
SELECT '21:15:69'::time;
```

ОШИБКА: значение поля типа date/time вне диапазона:

"21:15:69"

СТРОКА 1: select '21:15:69'::time;

^

Время можно вводить не только в 24-часовом формате, но и в 12-часовом, при этом нужно использовать дополнительные суффиксы am и pm. Например:

```
SELECT '10:15:16 am'::time;
```

```
time
```

```
-----
```

```
10:15:16
```

```
(1 строка)
```

```
SELECT '10:15:16 pm'::time;
```

```
time
```

```
-----
```

```
22:15:16
```

```
(1 строка)
```

Для получения значения текущего времени служит функция **current_time**. При ее вызове круглые скобки не используются.

```
SELECT current_time;  
          timetz
```

```
-----  
23:51:57.293522+03  
(1 строка)
```



локальный часовой пояс

- Текущее время выводится с высокой точностью и дополняется числовым значением, соответствующим локальному часовому поясу, который установлен в конфигурационном файле сервера PostgreSQL.
- В приведенном примере значение часового пояса равно +03, но если ваш компьютер находится в другом часовом поясе, то это значение будет другим, например, для регионов Сибири оно может быть +08.

- В результате объединения типов даты и времени получается интегральный тип — временная отметка. Этот тип существует в двух вариантах: с учетом часового пояса — **timestamp with time zone**, либо без учета часового пояса — **timestamp**. Для первого варианта существует сокращенное наименование — **timestampz**, которое является расширением PostgreSQL.
- Вот пример с учетом часового пояса:

```
SELECT timestamp with time zone '2016-09-21 22:25:35';
```

```
timestampz
```

```
-----
```

```
2016-09-21 22:25:35+03
```

```
(1 строка)
```



добавлено при выводе

Обратите внимание, что хотя мы не указали явно значение часового пояса при вводе данных, при выводе это значение «+03» было добавлено.

А это пример без учета часового пояса:

тип



строка



```
SELECT timestamp '2016-09-21 22:25:35';
```

```
timestamp
```

```
-----
```

```
2016-09-21 22:25:35
```

```
(1 строка)
```

В рассмотренных примерах мы использовали синтаксис **type 'string'** для указания конкретного типа простой литеральной константы.

Для получения значения текущей временной отметки (т. е. даты и времени в одном значении) служит функция **current_timestamp**. Она также вызывается без использования круглых скобок.

```
SELECT current_timestamp;
```

```
now
```

```
-----
```

```
2016-09-27 18:27:37.767739+03
```

```
(1 строка)
```

Здесь в выводе присутствует и часовой пояс: «+03».

Оба типа данных — `timestamp` и `timestampz` — занимают один и тот же объем 8 байтов, но значения типа `timestampz` хранятся, будучи приведенными к нулевому часовому поясу (UTC), а перед выводом приводятся к часовому поясу пользователя.

- Практический совет. На практике при принятии решения о том, какой из этих двух типов — `timestamp` или `timestampz` — использовать, необходимо учитывать, требуется ли значения, хранящиеся в таблице, приводить к местному часовому поясу или не требуется.
- Например, в расписании авиарейсов указывается местное время как для аэропорта отправления, так и для аэропорта прибытия. Поэтому в таком случае нужно использовать тип `timestamp`, чтобы это время не приводилось к текущему часовому поясу пользователя, где бы он ни находился.
- Из двух этих типов данных чаще используется `timestampz`.

Этот тип представляет собой продолжительность отрезка времени между двумя моментами времени. Его формат ввода таков:

quantity unit [quantity unit ...] direction

Здесь unit означает единицу измерения, а quantity — количество таких единиц. В качестве единиц измерения можно использовать следующие:

microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium.

Параметр direction может принимать значение ago (т. е. «тому назад») либо быть пустым.

```
SELECT '1 year 2 months ago'::interval;
```

```
interval
```

```
-----
```

```
-1 years -2 mons
```

```
(1 строка)
```

Можно использовать альтернативный формат, предлагаемый стандартом ISO 8601:

P [years-months-days] [T hours:minutes:seconds]

Здесь строка должна начинаться с символа «P», а символ «T» разделяет дату и время (все выражение пишется без пробелов). Например:

```
SELECT 'P0001-02-03T04:05:06'::interval;
```

```
interval
```

```
-----
```

```
1 year 2 mons 3 days 04:05:06
```

```
(1 строка)
```


Поскольку интервал — это отрезок времени между двумя временными отметками, то значение этого типа можно получить при вычитании одной временной отметки из другой.

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

```
interval
```

```
-----
```

```
15 days
```

```
(1 строка)
```

Значения временных отметок можно усекать с той или иной точностью с помощью функции **date_trunc**.

Например, для получения текущей временной отметки с точностью до одного часа нужно сделать так:

```
SELECT ( date_trunc( 'hour', current_timestamp ) );
```

```
date_trunc
```

```
-----
```

```
2016-09-27 22:00:00+03
```

```
(1 строка)
```

Из значений временных отметок можно с помощью функции **extract** извлекать отдельные поля, т. е. год, месяц, день, число часов, минут или секунд и т. д.

Например, чтобы извлечь номер месяца, нужно сделать так:

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

```
date_part
```

```
-----
```

```
11
```

```
(1 строка)
```

Напомним, что выражение timestamp '1999-11-27 12:34:56.123459' не означает операцию приведения типа. Оно присваивает тип данных timestamp литеральной константе.

2.4. Логический тип

Логический (**boolean**) тип может иметь три состояния: «true» и «false», а также неопределенное состояние, которое можно представить значением NULL. Таким образом, тип boolean реализует *трехзначную логику*.

В качестве состояния «true» могут служить следующие значения:

TRUE, 't', 'true', 'y', 'yes', 'on', '1'

В качестве состояния «false» могут служить следующие значения:

FALSE, 'f', 'false', 'n', 'no', 'off', '0'

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

Теперь выполним выборку всех строк из этой таблицы:

```
SELECT * FROM databases;
```

is_open_source	dbms_name
t	PostgreSQL
f	Oracle
t	MySQL
f	MS SQL Server

(4 строки)

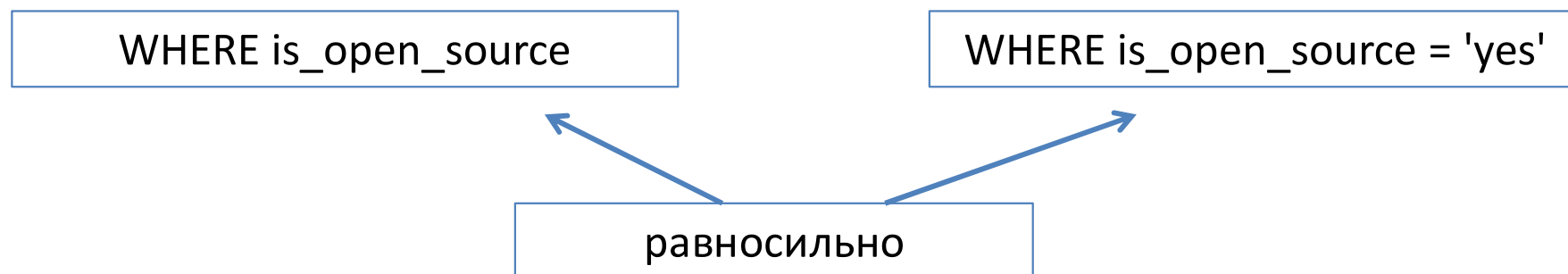
Выберем только СУБД с открытым исходным кодом:

```
SELECT * FROM databases WHERE is_open_source;
```

is_open_source	dbms_name
t	PostgreSQL
t	MySQL

(2 строки)

Обратите внимание, что в условии WHERE для проверки логических значений можно не писать выражение `WHERE is_open_source = 'yes'`, а достаточно просто указать имя столбца, содержащего логическое значение: `WHERE is_open_source`.



2.5. Массивы

- PostgreSQL позволяет создавать в таблицах такие столбцы, в которых будут содержаться не скалярные значения, а массивы переменной длины.
- Эти массивы могут быть многомерными и могут содержать значения любого из встроенных типов, а также типов данных, определенных пользователем.

Предположим, что нам необходимо сформировать и сохранить в базе данных в удобной форме графики работы пилотов авиакомпании, т. е. номера дней недели, когда они совершают полеты. Создадим таблицу, в которой эти графики будут храниться в виде единых списков, т. е. в виде одномерных массивов.

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

```
CREATE TABLE
```

```
INSERT INTO pilots VALUES
( 'Ivan',    '{ 1, 3, 5, 6, 7 }'::integer[] ),
( 'Petr',    '{ 1, 2, 5, 7 }'::integer[] ),
( 'Pavel',   '{ 2, 5 }'::integer[] ),
( 'Boris',   '{ 3, 5, 6 }'::integer[] );
```

```
INSERT 0 4
```

список значений

приведение типа

```
SELECT * FROM pilots;
```

pilot_name	schedule
Ivan	{1, 3, 5, 6, 7}
Petr	{1, 2, 5, 7}
Pavel	{2, 5}
Boris	{3, 5, 6}

(4 строки)

Предположим, что руководство компании решило, что каждый пилот должен летать 4 раза в неделю. Значит, нам придется обновить значения в таблице. Пилоту по имени Boris добавим один день с помощью операции конкатенации:

```
UPDATE pilots
SET schedule = schedule || 7
WHERE pilot_name = 'Boris';
UPDATE 1
```

Пилоту по имени Pavel добавим один день в конец списка (массива) с помощью функции `array_append`:

```
UPDATE pilots
SET schedule = array_append( schedule, 6 )
WHERE pilot_name = 'Pavel';
UPDATE 1
```

Ему же добавим один день в начало списка с помощью функции **array_prepend** (обратите внимание, что параметры функции поменялись местами):

```
UPDATE pilots
SET schedule = array_prepend( 1, schedule )
WHERE pilot_name = 'Pavel';
UPDATE 1
```

У пилота по имени Ivan имеется лишний день в графике. С помощью функции **array_remove** удалим из графика пятницу (второй параметр функции указывает значение элемента массива, а не индекс):

```
UPDATE pilots
SET schedule = array_remove( schedule, 5 )
WHERE pilot_name = 'Ivan';
UPDATE 1
```

У пилота по имени Petr изменим дни полетов, не изменяя их общего количества.

- Воспользуемся индексами для работы на уровне отдельных элементов массива.
- По умолчанию нумерация индексов начинается с единицы, а не с нуля. При необходимости ее можно изменить.
- К элементам одного и того же массива можно обращаться в предложении SET по отдельности, как будто это разные столбцы.

```
UPDATE pilots
SET schedule[ 1 ] = 2, schedule[ 2 ] = 3
WHERE pilot_name = 'Petr';
UPDATE 1
```

А можно было бы, используя срез (slice) массива, сделать и так:

```
UPDATE pilots
SET schedule[ 1:2 ] = ARRAY[ 2, 3 ]
WHERE pilot_name = 'Petr';
UPDATE 1
```

- Нотация с использованием ключевого слова **ARRAY** — это альтернативный способ создания массива (он соответствует стандарту SQL).
- В вышеприведенной команде запись 1:2 означает индексы первого и последнего элементов диапазона массива.
- Таким образом, присваивание новых значений производится сразу целому диапазону элементов массива.

```
SELECT * FROM pilots;
```

pilot_name	schedule
Boris	{3, 5, 6, 7}
Pavel	{1, 2, 5, 6}
Ivan	{1, 3, 6, 7}
Petr	{2, 3, 5, 7}

(4 строки)

Получим список пилотов, которые летают каждую среду:

```
SELECT * FROM pilots
WHERE array_position( schedule, 3 ) IS NOT NULL;
```

pilot_name	schedule
Boris	{3, 5, 6, 7}
Ivan	{1, 3, 6, 7}
Petr	{2, 3, 5, 7}

(3 строки)

Функция **array_position** возвращает индекс первого вхождения элемента с указанным значением в массив. Если же такого элемента нет, она возвратит NULL.

Выберем пилотов, летающих по понедельникам и воскресеньям:

```
SELECT *  
FROM pilots  
WHERE schedule @> '{ 1, 7 }'::integer[];
```

```
  pilot_name | schedule  
-----+-----
```

```
  Ivan      | {1,3,6,7}
```

(1 строка)

Оператор `@>` означает проверку того факта, что в левом массиве содержатся все элементы правого массива. Конечно, при этом в левом массиве могут находиться и другие элементы, что мы и видим в графике этого пилота.

Еще аналогичный вопрос: кто летает по вторникам и/или по пятницам? Для получения ответа воспользуемся оператором **&&**, который проверяет наличие общих элементов у массивов, т. е. пересекаются ли их множества значений. В нашем примере число общих элементов, если они есть, может быть равно одному или двум. Здесь мы также использовали нотацию с ключевым словом **ARRAY**, а не **'{ 2, 5 }::integer[]**. Вы можете применять ту, которая принята в рамках выполнения вашего проекта.

```
SELECT * FROM pilots
WHERE schedule && ARRAY[ 2, 5 ];
```

pilot_name		schedule
Boris		{3,5,6,7}
Pavel		{1,2,5,6}
Petr		{2,3,5,7}

(3 строки)

Сформулируем вопрос в форме отрицания: кто не летает ни во вторник, ни в пятницу? Для получения ответа добавим в предыдущую SQL-команду отрицание **NOT**:

```
SELECT * FROM pilots
WHERE NOT ( schedule && ARRAY[ 2, 5 ] );
```

pilot_name	schedule
Ivan	{1,3,6,7}

(1 строка)

Как развернуть массив в виде столбца таблицы?

В таком случае поможет функция **unnest**:

```
SELECT unnest( schedule ) AS days_of_week  
FROM pilots  
WHERE pilot_name = 'Ivan';
```

```
days_of_week  
-----  
1  
3  
6  
7
```

(4 строки)

2.6. Типы JSON

- Типы JSON предназначены для сохранения в столбцах таблиц базы данных таких значений, которые представлены в формате JSON (JavaScript Object Notation).
- Существует два типа: **json** и **jsonb**. Основное различие между ними заключается в *быстродействии*.
- Если столбец имеет тип json, тогда сохранение значений происходит быстрее, потому что они записываются в том виде, в котором были введены. Но при последующем использовании этих значений в качестве операндов или параметров функций будет каждый раз выполняться их разбор, что замедляет работу.

- При использовании типа jsonb разбор производится однократно, при записи значения в таблицу. Это несколько замедляет операции вставки строк, в которых содержатся значения данного типа. Но все последующие обращения к сохраненным значениям выполняются быстрее, т. к. выполнять их разбор уже не требуется.
- Тип json сохраняет порядок следования ключей в объектах и повторяющиеся значения ключей, а тип jsonb этого не делает.
- Практический совет. Рекомендуется в приложениях использовать тип jsonb, если только нет каких-то особых аргументов в пользу выбора типа json.

Предположим, что руководство авиакомпании всемерно поддерживает стремление пилотов улучшать свое здоровье, повышать уровень культуры и расширять кругозор. Поэтому разработчики базы данных авиакомпании получили задание создать специальную таблицу, в которую будут заноситься сведения о тех видах спорта, которыми занимается пилот, будет отмечаться наличие у него домашней библиотеки, а также фиксироваться количество стран, которые он посетил в ходе туристических поездок.

```
CREATE TABLE pilot_hobbies  
( pilot_name text,  
  hobbies jsonb  
);  
CREATE TABLE
```

```
INSERT INTO pilot_hobbies
VALUES ( 'Ivan',
        '{ "sports": [ "футбол", "плавание" ],
          "home_lib": true, "trips": 3
        }'::jsonb ),
      ( 'Petr',
        '{ "sports": [ "теннис", "плавание" ],
          "home_lib": true, "trips": 2
        }'::jsonb ),
      ( 'Pavel',
        '{ "sports": [ "плавание" ],
          "home_lib": false, "trips": 4
        }'::jsonb ),
      ( 'Boris',
        '{ "sports": [ "футбол", "плавание", "теннис" ],
          "home_lib": true, "trips": 0
        }'::jsonb );

INSERT 0 4
```

```
SELECT * FROM pilot_hobbies;
```

pilot_name	hobbies
Ivan	<pre>{ "trips": 3, "sports": ["футбол", "плавание"], "home_lib": true }</pre>
Petr	<pre>{ "trips": 2, "sports": ["теннис", "плавание"], "home_lib": true }</pre>
Pavel	<pre>{ "trips": 4, "sports": ["плавание"], "home_lib": false }</pre>
Boris	<pre>{ "trips": 0, "sports": ["футбол", "плавание", "теннис"], "home_lib": true }</pre>

(4 строки)

Как видно, при выводе строк из таблицы порядок ключей в JSON-объектах не был сохранен.

Предположим, что нужно сформировать футбольную сборную команду нашей авиакомпании для участия в турнире. Мы можем выбрать всех футболистов таким способом:

```
SELECT * FROM pilot_hobbies  
WHERE hobbies @> '{ "sports": [ "футбол" ] }'::jsonb;
```

pilot_name	hobbies
Ivan	{"trips": 3, "sports": ["футбол", "плавание"], "home_lib": true}
Boris	{"trips": 0, "sports": ["футбол", "плавание", "теннис"], "home_lib": true}

(2 строки)

Можно было эту задачу решить и таким способом:

```
SELECT pilot_name, hobbies->'sports' AS sports
FROM pilot_hobbies
WHERE hobbies->'sports' @> '[ "футбол" ]'::jsonb;
```

pilot_name	sports
Ivan	["футбол", "плавание"]
Boris	["футбол", "плавание", "теннис"]

(2 строки)

- В этом решении мы выводим только информацию о спортивных предпочтениях пилотов.
- Внимательно посмотрите, как используются одинарные и двойные кавычки.
- Операция «->» служит для обращения к конкретному ключу JSON-объекта.

- При создании столбца с типом данных json или jsonb не требуется задавать структуру объектов, т. е. конкретные имена ключей. Поэтому в принципе возможна ситуация, когда в разных строках в JSON-объектах будут использоваться различные наборы ключей.
- В нашем примере структуры JSON-объектов во всех строках совпадают. А если бы они не совпадали, то как можно было бы проверить наличие ключа? Продемонстрируем это.
- Ключа «sport» в наших объектах нет. Что покажет вызов функции count?

```
SELECT count( * )  
FROM pilot_hobbies  
WHERE hobbies ? 'sport' ;
```

```
count  
-----  
0  
(1 строка)
```



проверка наличия ключа

А вот ключ «sports» присутствует. Выполним ту же проверку:

```
SELECT count( * )  
FROM pilot_hobbies  
WHERE hobbies = 'sports';
```

Да, так и есть. Такие записи найдены.

```
count  
-----  
      4  
(1 строка)
```

Обновление JSON-объектов в строках таблицы

Предположим, что пилот по имени Boris решил посвятить себя только хоккею. Тогда в базе данных мы выполним такую операцию:

```
UPDATE pilot_hobbies
SET hobbies = hobbies || '{ "sports": [ "хоккей" ] }'
WHERE pilot_name = 'Boris';
UPDATE 1
```

Проверим, что получилось:

```
SELECT pilot_name, hobbies
FROM pilot_hobbies
WHERE pilot_name = 'Boris';
```

pilot_name	hobbies
Boris	{"trips": 0, "sports": ["хоккей"], "home_lib": true}

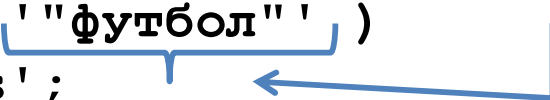
(1 строка)

Еще один способ обновления JSON-объектов

Если впоследствии Boris захочет возобновить занятия футболом, то с помощью функции **jsonb_set** можно будет обновить сведения о нем в таблице:

```
UPDATE pilot_hobbies
SET hobbies = jsonb_set( hobbies, '{ sports, 1 }',
                        '"футбол"' )
WHERE pilot_name = 'Boris';
```

UPDATE 1



одинарные и
двойные кавычки

- Второй параметр функции указывает путь в пределах JSON-объекта, куда нужно добавить новое значение.
- В данном случае этот путь состоит из имени ключа (sports) и номера добавляемого элемента в массиве видов спорта (номер 1).
- Нумерация элементов начинается с нуля.
- Третий параметр имеет тип jsonb, поэтому его литерал заключается в одинарные кавычки, а само добавляемое значение берется в двойные кавычки. В результате получается — '"футбол"'.

```
SELECT pilot_name, hobbies
FROM pilot_hobbies
WHERE pilot_name = 'Boris';
```

pilot_name	hobbies
Boris	{"trips": 0, "sports": ["хоккей", "футбол"], "home_lib": true}

(1 строка)

1. Лузанов, П. В. Postgres. Первое знакомство [Текст] / П. В. Лузанов, Е. В. Рогов, И. В. Лёвшин. – 5-е изд., перераб. и доп. – М. : Постгрес Профессиональный, 2019. – 156 с.
https://edu.postgrespro.ru/introbook_v5.pdf
2. Моргунов, Е. П. PostgreSQL. Основы языка SQL [Текст] : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с. https://edu.postgrespro.ru/sql_primer.pdf
3. Новиков, Б. А. Основы технологий баз данных [Текст] : учеб. пособие / Б. А. Новиков, Е. А. Горшкова ; под ред. Е. В. Рогова. – М. : ДМК Пресс, 2019. – 240 с. https://edu.postgrespro.ru/dbtech_part1.pdf
4. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – <https://www.postgresql.org>.
5. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – <https://postgrespro.ru>.

Для выполнения практических заданий необходимо использовать книгу:

Моргунов, Е. П. PostgreSQL. Основы языка SQL [Текст] : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с.

<https://postgrespro.ru/education/books/sqlprimer>

1. Изучить материал главы 4. Запросы к базе данных выполнять с помощью утилиты `psql`, описанной в главе 2, параграф 2.2.