Базы данных

Лекция 06 – Основы SQL. Типы

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.03.12

Оглавление

Типы

SQL — язык со строгой типизацией. Каждый элемент данных в нём имеет некоторый тип, определяющий его поведение и допустимое использование.

PostgreSQL имеет расширяемую систему типов, более универсальную и гибкую по сравнению с другими реализациями SQL.

Типы бывают встроенные и определяемые пользователем командой **CREATE TYPE**.

У типа есть имя. В Postgres некоторые типы имеют альтернативные имена — псевдонимы.

Типы делятся на:

числовые типы;
денежные типы;
символьные типы;
типы даты/времени;
логический тип;
типы перечислений;
типы, предназначенные для текстового поиска;
массивы;
составные типы;
диапазонные типы;
типы доменов;
идентификаторы объектов;
тип pg_lsn;- двоичные типы данных;
геометрические типы;
типы, описывающие сетевые адреса;
битовые строки;
тип UUID;
тип XML;
типы JSON;
псевдотипы.

Каждый тип данных имеет внутреннее представление, скрытое функциями ввода и вывода.
Многие встроенные типы стандартны и имеют очевидные внешние форматы.
Есть типы, уникальные для Postgres (геометрические пути).

Числовые типы

- целочисленные типы
- числа с произвольной (задаваемой) точностью
- типы с плавающей точкой
- последовательные типы

Имя	Размер	Описание	Диапазон
smallint	2 байта	целое в небольшом диапазоне	-32768 .. +32767
integer	4 байта	типичный выбор для целых чисел	-2147483648 .. +2147483647
bigint	8 байт	целое в большом диапазоне	-9223372036854775808 .. 9223372036854775807
decimal	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
numeric	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
real	4 байта	вещественное число с переменной точностью	точность в пределах 6 десятичных цифр
double precision	8 байт	вещественное число с переменной точностью	точность в пределах 15 десятичных цифр
smallserial	2 байта	небольшое целое с автоувеличением	1 .. 32767
serial	4 байта	целое с автоувеличением	1 .. 2147483647
bigserial	8 байт	большое целое с автоувеличением	1 .. 9223372036854775807

Представление данных в x86/x86_64

Segmetnt word size	32 bit						64 bit			
compiler	MS	Intel Win	Borla	Watc	GCC,Clang	Int Linux	MS	Intel Win	GCC,Clang	Int Linux
bool	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1
wchar_t	2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2
int	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	8	8
int64_t	8	8			8	8	8	8	8	8
enum (typical)	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8
long double	8	16	10	8	12	12	8	16	16	16
__m64	8	8			8	8		8	8	8
__m128	16	16			16	16	16	16	16	16
__m256	32	32			32	32	32	32	32	32
__m512	64	64			64	64	64	64	64	64
pointer	4	4	4	4	4	4	8	8	8	8
function pointer	4	4	4	4	4	4	8	8	8	8
data member ptr (min)	4	4	8	4	4	4	4	4	8	8
data member ptr (max)	12	12	8	12	4	4	12	12	8	8
member func ptr (min)	4	4	12	4	8	8	8	8	16	16
member func ptr (max)	16	16	12	16	8	8	24	24	16	16

Тип **numeric** позволяет хранить числа с очень большим количеством цифр.

Рекомендуется для хранения денежных сумм и других величин, где важна точность.

Вычисления с типом **numeric** дают точные результаты, где это возможно, например, при сложении, вычитании и умножении. Однако операции со значениями **numeric** выполняются гораздо медленнее, чем с целыми числами или с типами с плавающей точкой.

Столбец типа **numeric** объявляется следующим образом:

```
NUMERIC ( [ precision [, scale ]] )
```

precision — (точность) общее количество значимых цифр в числе > 0.

scale — (масштаб) определяет количество десятичных цифр в дробной части, справа от десятичной точки >= 0.

Например, число 123.45678 имеет точность 8 и масштаб 5.

Целочисленные значения можно считать числами с масштабом 0.

```
NUMERIC(precision)
```

Форма без указания точности:

```
NUMERIC
```

создаёт столбец типа «неограниченное число», в котором можно сохранять числовые значения любой длины до предела, обусловленного реализацией. В столбце этого типа входные значения не будут приводиться к какому-либо масштабу, тогда как в столбцах **numeric** с явно заданным масштабом все значения подгоняются под этот масштаб.

Стандарт SQL утверждает, что по умолчанию должен устанавливаться масштаб 0, т. е. значения должны приводиться к целым числам.

Числовые значения физически хранятся без каких-либо дополняющих нулей слева или справа.

Объявленные точность и масштаб столбца определяют максимальный,
а не фиксированный размер хранения.

В дополнение к обычным числовым значениям тип **numeric** принимает следующие специальные значения IEEE 754 — **Infinity**, **-Infinity**, **NaN**. В команде SQL, их нужно заключать в апострофы:

```
UPDATE table SET x = '-Infinity'
```

Регистр символов в этих строках не важен.

В качестве альтернативы значения бесконечности могут быть записаны как **inf** и **-inf**.

В IEEE 754 и большинстве реализаций **NaN** считается не равным любому другому значению, в том числе и самому **NaN**.

Однако, чтобы значения **numeric** можно было сортировать и использовать в древовидных индексах, Postgres считает, что значения **NaN** равны друг другу и при этом больше любых числовых значений, которые не **NaN**.

При округлении значений тип **numeric** выдаёт число, большее по модулю, тогда как (на большинстве платформ по умолчанию) типы **real** и **double precision** выдают ближайшее «чётное» число:

```
SELECT x, round(x::numeric) AS num_round, round(x::double precision) AS dbl_round
       FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
```

Типы с плавающей точкой

Типы данных **real** и **double precision** хранят приближённые числовые значения с переменной точностью. На всех поддерживаемых в настоящее время платформах эти типы реализуют стандарт IEEE 754 для двоичной арифметики с плавающей точкой с одинарной и двойной точностью, соответственно, в той мере, в какой его поддерживают процессор, операционная система и компилятор.

- Если нужна точность при хранении и вычислениях (например, для денежных сумм), необходимо использовать тип **numeric**.

- Если необходимо выполнять с этими типами сложные вычисления, имеющие большую важность, следует тщательно изучить реализацию операций в целевой среде и поведение в крайних случаях (бесконечность, антипереполнение).

- Проверка равенства двух чисел с плавающей точкой может не всегда давать ожидаемый результат.

В дополнение к обычным числовым значениям типы с плавающей точкой принимают следующие специальные значения:

Infinity
-Infinity
NaN

Они представляют особые значения, описанные в IEEE 754 (см. **numeric**).

Последовательные (сериальные) типы

Часто в Postgres используются для создания столбца с автоувеличением.

Способ, соответствующий стандарту SQL, заключается в использовании столбцов идентификации в **`CREATE TABLE`**.

Типы данных **`smallserial`**, **`serial`** и **`bigserial`** не являются настоящими типами, а представляют собой просто удобное средство для создания столбцов с уникальными идентификаторами (подобное свойству **`AUTO_INCREMENT`** в некоторых СУБД). В реализации 14 запись:

```
CREATE TABLE tablename (
    colname SERIAL
);
```

эквивалентна следующим командам:

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

Т.е. создаётся целочисленный столбец со значением по умолчанию, извлекаемым из генератора последовательности.

Чтобы в столбец нельзя было вставить NULL, в его определение добавляется ограничение NOT NULL. Во многих случаях также имеет смысл добавить для этого столбца ограничения **`UNIQUE`** или **`PRIMARY KEY`** для защиты от ошибочного добавления дублирующихся значений, поскольку автоматически это не происходит.

Последняя команда определяет, что последовательность «принадлежит» столбцу, так что она будет удалена при удалении столбца или таблицы.

Поскольку типы **smallserial**, **serial** и **bigserial** реализованы через последовательности, в числовом ряду значений столбца могут образовываться пропуски, даже если никакие строки не удалялись — значение, выделенное из последовательности, считается «использованным», даже если строку с этим значением не удалось вставить в таблицу. Это может произойти, например, при откате транзакции, добавляющей данные.

Чтобы вставить в столбец **serial** следующее значение последовательности, ему нужно присвоить значение по умолчанию. Это можно сделать, либо исключив его из списка столбцов в операторе **INSERT**, либо с помощью ключевого слова **DEFAULT**.

Имена типов **serial** и **serial4** эквивалентны — они создают столбцы типа **integer**.

bigserial и **serial8** создают столбцы **bigint**.

Тип **bigserial** следует использовать, если за всё время жизни таблицы планируется использовать больше чем 2^{31} значений.

smallserial и **serial2** создают столбец **smallint**.

Последовательность, созданная для столбца **serial**, автоматически удаляется при удалении связанного с ней столбца.

Последовательность можно удалить и отдельно от столбца, но при этом также будет удалено определение значения по умолчанию.

Денежные типы

Тип **money** хранит денежную сумму с фиксированной дробной частью.

Имя	Размер	Описание	Диапазон
money	8 байт	денежная сумма	-92 233 720 368 547 758.08 .. +92 233 720 368 547 758.07

Точность дробной части определяется на уровне базы данных параметром **LC_MONETARY**.

Для диапазона, показанного в таблице, предполагается, что число содержит два знака после запятой.

Входные данные могут быть записаны по-разному, в том числе в виде целых и дробных чисел, а также в виде строки в денежном формате, например **'$1,000.00'**.

Выводятся эти значения обычно в денежном формате, зависящем от региональных стандартов.

Выводимые значения этого типа зависят от региональных стандартов, поэтому попытка загрузить данные типа **money** в базу данных (например, при восстановлении из бэкапа) с другим параметром **LC_MONETARY** может быть неудачной.

Значения типов **numeric**, **int** и **bigint** можно привести к типу **money**.

Преобразования типов **real** и **double precision** возможны через тип **numeric**:

```
SELECT '12.34'::float8::numeric::money;
```

Использовать числа с плавающей точкой для денежных сумм не рекомендуется из-за возможных ошибок округления.

Значение **money** можно привести к типу **numeric** без потери точности. Преобразование в другие типы может быть неточным и также должно выполняться в два этапа:

```
SELECT '52093.89'::money::numeric::float8;
```

При делении значения типа `money` на целое число выполняется отбрасывание дробной части и получается целое, ближайшее к нулю.

Чтобы получить результат с округлением, необходимо либо:

- выполнять деление значения с плавающей точкой;
- до деления привести значение типа **`money`** к **`numeric`**, после чего привести результат к типу **`money`**.

Последний вариант предпочтительнее — он исключает риск потери точности.

Когда значение **`money`** делится на другое значение **`money`**, результатом будет значение типа **`double precision`**, то есть просто число, а не денежная величина.

Денежные единицы измерения при делении сокращаются.

LC_NUMERIC & LC_MONETARY

```
$ cat /usr/include/locale.h
#ifndef _LOCALE_H
#define _LOCALE_H   1

#include <features.h>

#define __need_NULL
#include <stddef.h>
#include <bits/locale.h>

__BEGIN_DECLS
/* These are the possibilities for the first argument to setlocale.
   The code assumes that the lowest LC_* symbol has the value zero.  */
#define LC_CTYPE          __LC_CTYPE
#define LC_NUMERIC        __LC_NUMERIC
#define LC_TIME           __LC_TIME
#define LC_COLLATE        __LC_COLLATE
#define LC_MONETARY       __LC_MONETARY
#define LC_MESSAGES       __LC_MESSAGES
#define LC_ALL            __LC_ALL
#define LC_PAPER          __LC_PAPER
#define LC_NAME           __LC_NAME
#define LC_ADDRESS        __LC_ADDRESS
#define LC_TELEPHONE      __LC_TELEPHONE
#define LC_MEASUREMENT    __LC_MEASUREMENT
#define LC_IDENTIFICATION __LC_IDENTIFICATION
```

```
/* Structure giving information about numeric and monetary notation.  */
struct lconv { /* Numeric (non-monetary) information.  */
  char *decimal_point;     /* Decimal point character.  */
  char *thousands_sep;     /* Thousands separator.  */
...

  /* Monetary information.  */
  /* First three chars are a currency symbol from ISO 4217. */
  /* Fourth char is the separator.  Fifth char is '\0'.  */
  char *int_curr_symbol;
  char *currency_symbol;   /* Local currency symbol.  */
  char *mon_decimal_point; /* Decimal point character.  */
  char *mon_thousands_sep; /* Thousands separator.  */
  char *mon_grouping;      /* Like `grouping' element (above).  */
  char *positive_sign;     /* Sign for positive values.  */
  char *negative_sign;     /* Sign for negative values.  */
  ...
};

/* Set and/or return the current locale.  */
extern char *setlocale (int __category, const char *__locale) __THROW;
/* Return the numeric/monetary information for the current locale.  */
extern struct lconv *localeconv (void) __THROW;
...
__END_DECLS
#endif /* locale.h */
```

Символьные типы

Имя	Описание
`character varying(`***n***`), varchar(`***n***`)`	строка ограниченной переменной длины
`character(`**n**`), char(`***n***`)`	строка фиксированной длины, дополненная пробелами
`text`	строка неограниченной переменной длины

SQL определяет два основных символьных типа:

`character varying(`*n*`)`
`character(`*n*`)`

где ***n*** — положительное число.

Оба эти типа могут хранить текстовые строки длиной до ***n*** символов (не байт !!!).

Попытка сохранить в столбце такого типа более длинную строку приведёт к ошибке, но только если все лишние символы не являются пробелами — пробелы будут усечены до максимально допустимой длины (SQL).

Если длина сохраняемой строки оказывается меньше объявленной, значения типа **`character`** будут дополнятся пробелами, а тип **`character varying`** просто сохранит короткую строку.

При приведении значения к типу **`character varying(`*n*`)`** или **`character(`*n*`)`**, часть строки, выходящая за границу в ***n*** символов, удаляется, не вызывая ошибки (SQL).

Postgres

Записи **varchar(n)** и **char(n)** являются синонимами **character varying(n)** и **character(n)**, соответственно.

Записи **character** без указания длины соответствует **character(1)**.

Если длина не указана для **character varying**, этот тип будет принимать строки любого размера.

Тип **text** позволяет хранить строки произвольной длины.

Тип **text** не в стандарте SQL, но его поддерживают многие СУБД SQL.

Значения типа **character** физически дополняются пробелами до **n** символов, хранятся и отображаются в таком виде.

При сравнении двух значений типа character дополняющие пробелы считаются незначащими и игнорируются.

С правилами сортировки, где пробельные символы являются значащими, это поведение может приводить к неожиданным результатам:

```
SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)
```

вернёт **true**, хотя в локали «C» символ пробела считается больше символа новой строки.

При приведении значения character к другому символьному типу дополняющие пробелы отбрасываются.

Дополняющие пробелы несут смысловую нагрузку в типах **character varying** и **text**, в проверках по шаблонам, и в регулярных выражениях.

Какие именно символы можно сохранить в этих типах данных, зависит от того, какой набор символов был выбран при создании базы данных.

В любом случае символ с кодом 0 (NUL) сохранить нельзя, вне зависимости от выбранного набора символов.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой строки, включая дополняющие пробелы для типа **character**. Для строк длиннее требуется не 1, а 4 дополнительных байта. Система может автоматически сжимать длинные строки, так что физический размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные таблицы, чтобы они не замедляли работу с другими столбцами.

Примеры

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- (1)
  a   | char_length
------+-------------
 ok   |           2
```

```
CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ОШИБКА:  значение не умещается в тип character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- явное усечение
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

Максимально возможный размер строки составляет около 1 ГБ.

Допустимое значение ***n*** в объявлении типа данных будет меньше этого числа — в зависимости от кодировки каждый символ может занимать несколько байт.

Если необходимо хранить строки без определённого предела длины, следует использовать типы **`text`** или **`character varying`** без указания длины.

В Postgres есть ещё два символьных типа фиксированной длины.

Имя	Размер	Описание
`"char"`	1 байт	внутренний однобайтный тип
`name`	64 байта	внутренний тип для имён объектов

Тип **`name`** создан только для хранения идентификаторов во внутренних системных таблицах и не предназначен для обычного применения пользователями.

В настоящее время его длина составляет 64 байта (63 ASCII-символа плюс терминатор).

В исходном коде на C длина задаётся константой **`NAMEDATALEN`**. Эта константа определяется во время компиляции и её можно менять в некотороых случаях.

Максимальная длина по умолчанию в следующих версиях может быть увеличена.

Тип **`"char"`** (двойные кавычки) отличается от **`char(1)`** тем, что он фактически хранится в одном байте. Используется во внутренних системных таблицах для простых перечислений.

Типы даты/времени

- ввод даты/времени
- вывод даты/времени
- часовые пояса
- ввод интервалов
- вывод интервалов

Юлианская дата — число суток, прошедших начиная с полудня понедельника, 1 января 4713 года до н. э. *пролептического* юлианского календаря или, что то же самое, 24 ноября 4714 года до н. э. *пролептического* григорианского календаря (соответственно, −4712 год и −4713 год по астрономическому счёту лет).

Даты сменяются в полдень.

В гражданском счёте первому году н. э. предшествует первый год до н. э.

В астрономическом счёте первому году н. э. предшествует нулевой год.

Юлианский календарь введен 1 января 45 года до н. э.

Григорианский календарь уточняет юлианский. Введен 4 октября 1582 года взамен юлианского, при этом следующим днём после четверга 4 октября стала пятница 15 октября (+11).

```
typedef struct {
    int    year; //
    int    ymon; // месяц года григорианский
    int    mday; // день месяца 1..31
    int    hour; // 0..23
    int    min;  // 0..59
    double sec;  // 0..60 -- (високосная секунда)
    double yday; // день года 1..365
} UTC;
```

PostgreSQL поддерживает полный набор типов даты и времени SQL.

Все даты считаются по Григорианскому календарю, даже для времени до его введения.

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
`timestamp [ (p) ] [ without time zone ]`	8 байт	дата и время (без часового пояса)	4713 до н. э.	294276 н. э.	1 мкс
`timestamp [ (p) ] with time zone`	8 байт	дата и время (с часовым поясом)	4713 до н. э.	294276 н. э.	1 мкс
`date`	4 байта	дата (без времени суток)	4713 до н. э.	5874897 н. э.	1 день
`time [ (p) ] [ without time zone ]`	8 байт	время суток (без даты)	00:00:00	24:00:00	1 мкс
`time [ (p) ] with time zone`	12 байт	время дня (без даты), с часовым поясом	00:00:00+1559	24:00:00-1559	1 мкс
`interval [ поля ][ (p) ]`	16 байт	временной интервал	-178000000 лет	178000000 лет	1 мкс

Типы **time**, **timestamp** и **interval** принимают необязательное значение точности **p**, определяющее, сколько знаков после запятой должно сохраняться в секундах.

По умолчанию точность не ограничивается. Допустимые значения **p** лежат в интервале от 0 до 6.

Тип **time with time zone** определён стандартом SQL, но ценность его сомнительна.

В большинстве случаев сочетание типов **date**, **time**, **timestamp without time zone** и **timestamp with time zone** удовлетворяет все потребности в функционале дат/времени, возникающие в приложениях.

Значения даты и времени принимаются во многих форматах, включая ISO 8601.
SQL предусматривает следующий синтаксис:

type *[* **(p)** *]* **'value'**

p — необязательное указание точности, определяющее число знаков после точки в секундах.
Точность может быть определена для типов **time**, **timestamp** и **interval** в интервале от 0 до 6.
Если в определении константы точность не указана, она считается равной точности значения в строке (но не больше 6 цифр).

Пример

1999-01-08 ISO 8601; рекомендуемый формат даты
1999-01-08 04:05:06
1999-01-08 04:05:06-8:00

Стандарт SQL различает константы типов **timestamp without time zone** и **timestamp with time zone** по знаку «**+**» или «**-**» и смещению часового пояса, добавленному после времени.
Следовательно, согласно стандарту, записи
TIMESTAMP '2004-10-19 10:23:54' — тип **timestamp without time zone**
TIMESTAMP '2004-10-19 10:23:54+02' — тип **timestamp with time zone**.

PostgreSQL никогда не анализирует содержимое текстовой строки, чтобы определить тип значения, и поэтому обе записи будут обработаны как значения типа **timestamp without time zone**. Чтобы текстовая константа обрабатывалась как **timestamp with time zone**, нужно это указать явно:

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

Значения **timestamp with time zone** внутри всегда хранятся в UTC (Universal Coordinated Time — Всемирное скоординированное время). Вводимое значение, в котором часовой пояс указан явно, переводится в UTC с учётом смещения часового пояса. Если часовой пояс не указан, подразумевается заданный системным параметром **timezone** и время пересчитывается в UTC со смещением **timezone**.

```
$ timedatectl show
Timezone=Europe/Minsk
LocalRTC=no
CanNTP=yes
NTP=yes
NTPSynchronized=no
TimeUSec=Sat 2022-10-29 00:22:42 +03
RTCTimeUSec=Sat 2022-10-29 00:22:43 +03
```

Специальные значения

Postgres поддерживает несколько специальных значений даты/времени. Значения **infinity** и **-infinity** имеют особое представление в системе и они отображаются в том же виде. Другие преобразуются в значения даты/времени. Для использования специальных значений в качестве констант в командах SQL, их нужно заключать в апострофы.

Вводимая строка	Допустимые типы	Описание
epoch	date, timestamp	1970-01-01 00:00:00+00 (точка отсчёта времени в Unix)
infinity	date, timestamp	время после максимальной допустимой даты
-infinity	date, timestamp	время до минимальной допустимой даты
now	date, time, timestamp	время начала текущей транзакции
today	date, timestamp	время начала текущих суток (00:00)
tomorrow	date, timestamp	время начала следующих суток (00:00)
yesterday	date, timestamp	время начала предыдущих суток (00:00)
allballs	time	00:00:00.00 UTC

Ввод интервалов

Значения типа **interval** могут быть записаны в следующей расширенной форме:

[@] **quantity unit** *[***quantity unit...***]* *[***direction***]*

quantity — это число (возможно, со знаком);
unit — одно из значений: **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.
Либо эти же слова во множественном числе, либо их сокращения;
direction может принимать значение **ago** (назад) или быть пустым.
Знак **@** является необязательным.
Все заданные величины различных единиц суммируются вместе с учётом знака чисел.
Указание **ago** меняет знак всех полей на противоположный.
Количества дней, часов, минут и секунд можно определить, не указывая явно соответствующие единицы, например **'1 12:59:10'** эквивалентно **'1 day 12 hours 59 min 10 sec'**.
Сочетание года и месяца также можно записать через минус, например **'200-10'** означает то, же что и **'200 years 10 months'**.
Эти краткие формы только и разрешены стандартом SQL и они используются postgres при выводе, когда **IntervalStyle** имеет значение **sql_standard**.

Логический тип

Тип **boolean** — стандартный SQL-тип.

Тип **boolean** может иметь следующие состояния — «**true**», «**false**» и третье состояние «**unknown**», которое представляется SQL-значением **NULL**. Занимает 1 байт.

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL — **TRUE**, **FALSE** и **NULL**.

Функция ввода данных типа **boolean** воспринимает следующие строковые представления состояния «**true**» — **true**, **yes**, **on**, **1** и следующие представления состояния «**false**» — **false**, **no**, **off**, **0**.

Принимаются и уникальные префиксы этих строк, например **t** или **n**.

Регистр символов не имеет значения. Пробельные символы в начале и в конце строки игнорируются.

Функция вывода данных типа **boolean** всегда выдает **t** или **f**:

```
CREATE TABLE test1 (
    a boolean,
    b text
);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Геометрические типы

- точки;
- прямые;
- отрезки;
- прямоугольники;
- пути;
- многоугольники;
- окружности.

Геометрические типы данных представляют объекты в двумерном пространстве.

Имя	Размер	Описание	Представление
`point`	16 байт	Точка на плоскости	`(x,y)`
`line`	32 байта	Бесконечная прямая	`{A,B,C}`
`lseg`	32 байта	Отрезок	`((x1,y1),(x2,y2))`
`box`	32 байта	Прямоугольник	`((x1,y1),(x2,y2))`
`path`	16+16n байт	Закрытый путь (подобен многоугольнику)	`((x1,y1),...)`
`path`	16+16n байт	Открытый путь	`[(x1,y1),...]`
`polygon`	40+16n байт	Многоугольник (подобен закрытому пути)	`((x1,y1),...)`
`circle`	24 байта	Окружность	`<(x,y),r>` (центр окружности и радиус)

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, Postgres имеет богатый набор функций и операторов.

Типы, описывающие сетевые адреса

Для хранения сетевых адресов лучше использовать эти специальные типы, а не простые текстовые строки, поскольку Postgres проверяет вводимые значения данных типов и предоставляет специализированные операторы и функции для работы с ними.

Имя	Размер	Описание
`cidr`	7 или 19 байт	Сети IPv4 и IPv6
`inet`	7 или 19 байт	Узлы и сети IPv4 и IPv6
`macaddr`	6 байт	MAC-адреса
`macaddr8`	8 байт	MAC-адреса (в формате EUI-64)

```
$ ifconfig
wlp0s20f3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.21  netmask 255.255.255.0  broadcast 192.168.100.255
        inet6 fe80::8bde:9f77:3315:fc03  prefixlen 64  scopeid 0x20<link>
        ether f4:4e:e3:91:75:35  txqueuelen 1000  (Ethernet)
```

inet

Тип **`inet`** содержит IPv4- или IPv6-адрес узла и может в этом же поле содержать также и его подсеть. Подсеть представляется числом бит, определяющих адрес сети в адресе узла (маска сети).

Если маска для сетевого адреса IPv4 равна 32, такое значение представляет не подсеть, а определённый узел.

Адреса IPv6 имеют длину 128 бит, поэтому уникальный адрес узла задаётся с маской 128 бит.

Если требуются только адреса сетей, следует использовать тип **`cidr`**, а не **`inet`**.

Формат:

`address/y`

`y` — число бит в маске сети.

cidr

Тип **cidr** содержит определение сети IPv4 или IPv6.

Входные и выходные форматы соответствуют соглашениям CIDR (Classless Internet Domain Routing).

Определение сети записывается в формате

address/y

где **address** — минимальный адрес в сети, представленный в виде адреса IPv4 или IPv6, а **y** — число бит в маске сети.

Если **y** не указывается, это значение вычисляется по старой классовой схеме нумерации сетей, но при этом оно может быть увеличено, чтобы в него вошли все байты введённого адреса.

Если в сетевом адресе справа от маски сети окажутся биты со значением 1, он будет считаться ошибочным.

Различия inet и cidr

inet принимает значения с ненулевыми битами справа от маски сети, а **cidr** — нет.

Например, значение `192.168.0.1/24` является допустимым для типа **inet**, но не для **cidr**.

macaddr

Тип **macaddr** предназначен для хранения MAC-адреса, примером которого является адрес сетевого интерфейса.

Вводимые значения могут задаваться в следующих форматах:

```
'08:00:2b:01:02:03' (IEEE 802-2001 MSB)
'08-00-2b-01-02-03' (IEEE 802-2001 LSB)
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

macaddr8

Тип **macaddr8** хранит MAC-адреса в формате EUI-64, применяющиеся, например, для аппаратных адресов сетевых и прочих коммуникационных интерфейсов.

Этот тип может принять и 6-байтовые, и 8-байтовые MAC-адреса и сохраняет их в 8 байтах.

MAC-адреса, заданные в 6-байтовом формате, хранятся в формате 8 байт, а 4-ый и 5-ый байт содержат FF и FE, соответственно. Ниже показаны примеры допустимых входных строк в стандарте:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
```

Типы даты/времени

- ввод даты/времени
- вывод даты/времени
- часовые пояса
- ввод интервалов
- вывод интервалов

Юлианская дата — число суток, прошедших начиная с полудня понедельника, 1 января 4713 года до н. э. *пролептического* юлианского календаря или, что то же самое, 24 ноября 4714 года до н. э. *пролептического* григорианского календаря (соответственно, −4712 год и −4713 год по астрономическому счёту лет).

Даты сменяются в полдень.

В гражданском счёте первому году н. э. предшествует первый год до н. э.

В астрономическом счёте первому году н. э. предшествует нулевой год.

Юлианский календарь введен 1 января 45 года до н. э.

Григорианский календарь уточняет юлианский. Введен 4 октября 1582 года взамен юлианского, при этом следующим днём после четверга 4 октября стала пятница 15 октября (+11).

```
typedef struct {
    int    year; //
    int    ymon; // месяц года григорианский
    int    mday; // день месяца 1..31
    int    hour; // 0..23
    int    min;  // 0..59
    double sec;  // 0..60 -- (високосная секунда)
    double yday; // день года 1..365
} UTC;
```

PostgreSQL поддерживает полный набор типов даты и времени SQL.

Все даты считаются по Григорианскому календарю, даже для времени до его введения.

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
`timestamp [ (p) ] [ without time zone ]`	8 байт	дата и время (без часового пояса)	4713 до н. э.	294276 н. э.	1 мкс
`timestamp [ (p) ] with time zone`	8 байт	дата и время (с часовым поясом)	4713 до н. э.	294276 н. э.	1 мкс
`date`	4 байта	дата (без времени суток)	4713 до н. э.	5874897 н. э.	1 день
`time [ (p) ] [ without time zone ]`	8 байт	время суток (без даты)	00:00:00	24:00:00	1 мкс
`time [ (p) ] with time zone`	12 байт	время дня (без даты), с часовым поясом	00:00:00+1559	24:00:00-1559	1 мкс
`interval [ поля ][ (p) ]`	16 байт	временной интервал	-178000000 лет	178000000 лет	1 мкс

Типы **time**, **timestamp** и **interval** принимают необязательное значение точности **p**, определяющее, сколько знаков после запятой должно сохраняться в секундах.

По умолчанию точность не ограничивается. Допустимые значения **p** лежат в интервале от 0 до 6.

Тип **time with time zone** определён стандартом SQL, но ценность его сомнительна.

В большинстве случаев сочетание типов **date**, **time**, **timestamp without time zone** и **timestamp with time zone** удовлетворяет все потребности в функционале дат/времени, возникающие в приложениях.

Значения даты и времени принимаются во многих форматах, включая ISO 8601.
SQL предусматривает следующий синтаксис:

type *[* **(p)** *]* **'value'**

p — необязательное указание точности, определяющее число знаков после точки в секундах.
Точность может быть определена для типов **time**, **timestamp** и **interval** в интервале от 0 до 6.
Если в определении константы точность не указана, она считается равной точности значения в строке (но не больше 6 цифр).

Пример

1999-01-08 ISO 8601; рекомендуемый формат даты
1999-01-08 04:05:06
1999-01-08 04:05:06-8:00

Стандарт SQL различает константы типов **timestamp without time zone** и **timestamp with time zone** по знаку «**+**» или «**-**» и смещению часового пояса, добавленному после времени.
Следовательно, согласно стандарту, записи
TIMESTAMP '2004-10-19 10:23:54' — тип **timestamp without time zone**
TIMESTAMP '2004-10-19 10:23:54+02' — тип **timestamp with time zone**.

PostgreSQL никогда не анализирует содержимое текстовой строки, чтобы определить тип значения, и поэтому обе записи будут обработаны как значения типа **timestamp without time zone**. Чтобы текстовая константа обрабатывалась как **timestamp with time zone**, нужно это указать явно:

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

Значения **timestamp with time zone** внутри всегда хранятся в UTC (Universal Coordinated Time — Всемирное скоординированное время). Вводимое значение, в котором часовой пояс указан явно, переводится в UTC с учётом смещения часового пояса. Если часовой пояс не указан, подразумевается заданный системным параметром **timezone** и время пересчитывается в UTC со смещением **timezone**.

```
$ timedatectl show
Timezone=Europe/Minsk
LocalRTC=no
CanNTP=yes
NTP=yes
NTPSynchronized=no
TimeUSec=Sat 2022-10-29 00:22:42 +03
RTCTimeUSec=Sat 2022-10-29 00:22:43 +03
```

Специальные значения

Postgres поддерживает несколько специальных значений даты/времени. Значения **infinity** и **-infinity** имеют особое представление в системе и они отображаются в том же виде. Другие преобразуются в значения даты/времени. Для использования специальных значений в качестве констант в командах SQL, их нужно заключать в апострофы.

Вводимая строка	Допустимые типы	Описание
epoch	date, timestamp	1970-01-01 00:00:00+00 (точка отсчёта времени в Unix)
infinity	date, timestamp	время после максимальной допустимой даты
-infinity	date, timestamp	время до минимальной допустимой даты
now	date, time, timestamp	время начала текущей транзакции
today	date, timestamp	время начала текущих суток (00:00)
tomorrow	date, timestamp	время начала следующих суток (00:00)
yesterday	date, timestamp	время начала предыдущих суток (00:00)
allballs	time	00:00:00.00 UTC

Ввод интервалов

Значения типа **interval** могут быть записаны в следующей расширенной форме:

[@] **quantity unit** *[***quantity unit...***]* *[***direction***]*

quantity — это число (возможно, со знаком);
unit — одно из значений: **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.
Либо эти же слова во множественном числе, либо их сокращения;
direction может принимать значение **ago** (назад) или быть пустым.
Знак **@** является необязательным.
Все заданные величины различных единиц суммируются вместе с учётом знака чисел.
Указание **ago** меняет знак всех полей на противоположный.
Количества дней, часов, минут и секунд можно определить, не указывая явно соответствующие единицы, например **'1 12:59:10'** эквивалентно **'1 day 12 hours 59 min 10 sec'**.
Сочетание года и месяца также можно записать через минус, например **'200-10'** означает то, же что и **'200 years 10 months'**.
Эти краткие формы только и разрешены стандартом SQL и они используются postgres при выводе, когда **IntervalStyle** имеет значение **sql_standard**.

Логический тип

Тип **boolean** — стандартный SQL-тип.

Тип **boolean** может иметь следующие состояния — «**true**», «**false**» и третье состояние «**unknown**», которое представляется SQL-значением **NULL**. Занимает 1 байт.

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL — **TRUE**, **FALSE** и **NULL**.

Функция ввода данных типа **boolean** воспринимает следующие строковые представления состояния «**true**» — **true**, **yes**, **on**, **1** и следующие представления состояния «**false**» — **false**, **no**, **off**, **0**.

Принимаются и уникальные префиксы этих строк, например **t** или **n**.

Регистр символов не имеет значения. Пробельные символы в начале и в конце строки игнорируются.

Функция вывода данных типа **boolean** всегда выдает **t** или **f**:

```
CREATE TABLE test1 (
    a boolean,
    b text
);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Геометрические типы

- точки;
- прямые;
- отрезки;
- прямоугольники;
- пути;
- многоугольники;
- окружности.

Геометрические типы данных представляют объекты в двумерном пространстве.

Имя	Размер	Описание	Представление
`point`	16 байт	Точка на плоскости	`(x,y)`
`line`	32 байта	Бесконечная прямая	`{A,B,C}`
`lseg`	32 байта	Отрезок	`((x1,y1),(x2,y2))`
`box`	32 байта	Прямоугольник	`((x1,y1),(x2,y2))`
`path`	16+16n байт	Закрытый путь (подобен многоугольнику)	`((x1,y1),...)`
`path`	16+16n байт	Открытый путь	`[(x1,y1),...]`
`polygon`	40+16n байт	Многоугольник (подобен закрытому пути)	`((x1,y1),...)`
`circle`	24 байта	Окружность	`<(x,y),r>` (центр окружности и радиус)

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, Postgres имеет богатый набор функций и операторов.

Типы, описывающие сетевые адреса

Для хранения сетевых адресов лучше использовать эти специальные типы, а не простые текстовые строки, поскольку Postgres проверяет вводимые значения данных типов и предоставляет специализированные операторы и функции для работы с ними.

Имя	Размер	Описание
`cidr`	7 или 19 байт	Сети IPv4 и IPv6
`inet`	7 или 19 байт	Узлы и сети IPv4 и IPv6
`macaddr`	6 байт	MAC-адреса
`macaddr8`	8 байт	MAC-адреса (в формате EUI-64)

```
$ ifconfig
wlp0s20f3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.100.21  netmask 255.255.255.0  broadcast 192.168.100.255
        inet6 fe80::8bde:9f77:3315:fc03  prefixlen 64  scopeid 0x20<link>
        ether f4:4e:e3:91:75:35  txqueuelen 1000  (Ethernet)
```

inet

Тип **`inet`** содержит IPv4- или IPv6-адрес узла и может в этом же поле содержать также и его подсеть. Подсеть представляется числом бит, определяющих адрес сети в адресе узла (маска сети).

Если маска для сетевого адреса IPv4 равна 32, такое значение представляет не подсеть, а определённый узел.

Адреса IPv6 имеют длину 128 бит, поэтому уникальный адрес узла задаётся с маской 128 бит.

Если требуются только адреса сетей, следует использовать тип **`cidr`**, а не **`inet`**.

Формат:

`address/y`

`y` — число бит в маске сети.

cidr

Тип **cidr** содержит определение сети IPv4 или IPv6.

Входные и выходные форматы соответствуют соглашениям CIDR (Classless Internet Domain Routing).

Определение сети записывается в формате

address/y

где **address** — минимальный адрес в сети, представленный в виде адреса IPv4 или IPv6, а **y** — число бит в маске сети.

Если **y** не указывается, это значение вычисляется по старой классовой схеме нумерации сетей, но при этом оно может быть увеличено, чтобы в него вошли все байты введённого адреса.

Если в сетевом адресе справа от маски сети окажутся биты со значением 1, он будет считаться ошибочным.

Различия inet и cidr

inet принимает значения с ненулевыми битами справа от маски сети, а **cidr** — нет.

Например, значение `192.168.0.1/24` является допустимым для типа **inet**, но не для **cidr**.

macaddr

Тип **macaddr** предназначен для хранения MAC-адреса, примером которого является адрес сетевого интерфейса.

Вводимые значения могут задаваться в следующих форматах:

```
'08:00:2b:01:02:03' (IEEE 802-2001 MSB)
'08-00-2b-01-02-03' (IEEE 802-2001 LSB)
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

macaddr8

Тип **macaddr8** хранит MAC-адреса в формате EUI-64, применяющиеся, например, для аппаратных адресов сетевых и прочих коммуникационных интерфейсов.

Этот тип может принять и 6-байтовые, и 8-байтовые MAC-адреса и сохраняет их в 8 байтах.

MAC-адреса, заданные в 6-байтовом формате, хранятся в формате 8 байт, а 4-ый и 5-ый байт содержат FF и FE, соответственно. Ниже показаны примеры допустимых входных строк в стандарте:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
```

Двоичные типы данных

- шестнадцатеричный формат **bytea**
- формат спецпоследовательностей **bytea**

Для хранения двоичных данных предназначен тип **bytea**

Имя	**Размер**	**Описание**
bytea	1 или 4 байта плюс сама двоичная строка	двоичная строка переменной длины

Двоичные строки представляют собой последовательность октетов (ASN.1) и имеют два отличия от текстовых строк.

1) в двоичных строках можно хранить байты с кодом 0 и другими «непечатаемыми» значениями (обычно это значения вне десятичного диапазона 32..126).

В текстовых строках нельзя сохранять нулевые байты, а также значения и последовательности значений, не соответствующие выбранной кодировке базы данных.

2) в операциях с двоичными строками обрабатываются байты в чистом виде, тогда как текстовые строки обрабатываются в зависимости от языковых стандартов. То есть, двоичные строки больше подходят для данных, которые программист видит как «просто байты», а символьные строки — для хранения текста.

Битовые строки

Битовые строки представляют собой последовательности из 1 и 0. Их можно использовать для хранения или отображения битовых масок. В SQL есть два битовых типа:

- **bit(n)**
- **bit varying(n)**

где **n** — положительное целое число.

Длина значения типа **bit** должна в точности равняться **n** — при попытке сохранить более длинные или более короткие данные произойдёт ошибка.

Данные типа **bit varying** могут иметь переменную длину, но не превышающую **n** — строки большей длины не будут приняты.

Запись **bit** без указания длины равнозначна записи **bit(1)**.

Запись **bit varying** без указания длины подразумевает строку неограниченной длины.

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ОШИБКА:  длина битовой строки (2) не соответствует типу bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Тип UUID

Тип данных **uuid** сохраняет универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID), определённые в RFC 4122, ISO/IEC 9834-8:2005 и связанных стандартах.

В некоторых системах это называется GUID, глобальным уникальным идентификатором.

Этот идентификатор представляет собой 128-битное значение, генерируемое специальным алгоритмом, практически гарантирующим, что этим же алгоритмом оно не будет получено больше нигде в мире.

Таким образом, эти идентификаторы будут уникальными и в распределённых системах, а не только в единственной базе данных, как значения генераторов последовательностей.

```
$ cat /etc/fstab
UUID=77b34e2d-e9f9-42ba-9cab-458fcfca267e /         ext4 defaults                  1 1
UUID=66c3886f-c106-4439-a204-f36b01814591 /boot     ext4 defaults                  1 2
UUID=400C-7ED1                            /boot/efi vfat umask=0077,shortname=winnt 0 2
UUID=5bcb7f89-0109-42cd-83ab-478eac528f4f /home     ext4 defaults                  1 2
```

Пример UUID в стандартном текстовом виде:

a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

```
$ uuidgen
df8ae888-6076-4f55-b3fb-677877e5b90d
$ uuidgen -t --sha1 --namespace @dns --name "lsi.bas-net.by"
770990c7-4f1a-5242-8565-d32d339c066d
```

uuid_t

Массивы

Postgres позволяет определять столбцы таблицы как **многомерные массивы переменной длины**.

Элементами массивов могут быть любые встроенные или определённые пользователями базовые типы, перечисления, составные типы, типы-диапазоны или домены.

Объявления типов массивов

Для объявления типа массива к названию типа элементов добавляются квадратные скобки **[]**.

Приведенная ниже команда создает таблицу **sal_emp** со столбцами типов **text**, одномерный массив с элементами типа **integer** — квартальная зарплата работников, и двухмерный массив с элементами типа **text** — недельный график работника.

```
CREATE TABLE sal_emp (
    name            text,       -- ФИО работника
    pay_by_quarter  integer[4], -- квартальная зарплата
    schedule        text[][]    -- недельный график
);
```

Ограничений на фактический размер массива не накладывается, соответственно, указанная размерность для **pay_by_quarter** игнорируется.

Также существует и альтернативный синтаксис с ключевым словом **ARRAY** (стандарт SQL).

Столбец **pay_by_quarter** можно было бы определить так:

```
pay_by_quarter  integer ARRAY[4]
```

Или без указания размера массива:

```
pay_by_quarter  integer ARRAY,
```

Ввод значения массива

Чтобы записать значение массива в виде литеральной константы, следует заключить значения элементов в фигурные скобки и разделить их запятыми.

Если элемент содержит запятые или фигурные скобки, его заключают в двойные кавычки.

Общий формат константы массива:

```
'{ val_1 delim val_2 delim ... val_N}'
```

delim — символ, указанный в качестве разделителя в соответствующей записи в системной таблице **pg_type**. Обычно это запятая (**,**) или, в случае типа **box**, точка с запятой (**;**).

Каждое **val** — либо константа типа элемента массива, либо вложенный массив.

```
'{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}'
```

Пример

```
INSERT INTO sal_emp
    VALUES ('Virt N.',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Knuth D.',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

И результат запроса:

```
SELECT * FROM sal_emp;
name |     pay_by_quarter      |                  schedule
-----+-------------------------+-----------------------------------
Bill |{10000,10000,10000,10000}|{{meeting,lunch},{training,presentation}}
Carol|{20000,25000,25000,25000}|{{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

В многомерных массивов число элементов в каждой размерности должно быть одинаковым.
В противном случае возникает ошибка. Например:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ОШИБКА:  ошибочный литерал массива: "{{"meeting", "lunch"}, {"meeting"}}"
СТРОКА 4:     '{{"meeting", "lunch"}, {"meeting"}}');
              ^
ПОДРОБНОСТИ:  Для многомерных массивов должны задаваться вложенные массивы с
соответствующими размерностями.
```

Обращение к массивам

Указывается индекс элемента в массиве.
Следующий запрос получает имена сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Knuth D.
```

Индексы элементов массива записываются в квадратных скобках.
По умолчанию в Postgres действует соглашение о нумерации элементов массива с 1. Т.е. в массиве из **n** элементов первым считается **array[1]**, а последним — **array[n]**.

Следующий запрос выдаёт зарплату всех сотрудников в третьем квартале:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
```

Прямоугольные срезы массива (блоки)

Срез массива (подмассив) для одной или нескольких размерностей обозначается как

lower-bound:upper-bound

Следующий запрос получает первые пункты в графике Дональда Кнута в первые два дня недели:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Knuth D.';
        schedule
------------------------
 {{breakfast},{meeting}}
```

Если одна из размерностей записана в виде среза, то есть содержит двоеточие, тогда срез распространяется на все размерности.

Если при этом для размерности указывается только одно число (без двоеточия), в срез войдут элемент от 1 до заданного номера. Например, в следующем примере **[2]** эквивалентно **[1:2]**:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Knuth D.';
                 schedule
-----------------------------------------
 {{breakfast,consulting},{meeting,lunch}}
```

Поэтому при обращении к одному элементу размерности, срезы нужно записывать явно для всех измерений, например **[1:2][2:2]** вместо **[2][2:2]**.

```
SELECT schedule[1:2][2:2] FROM sal_emp WHERE name = 'Knuth D.';
        schedule
------------------------
 {{consulting},{lunch}}
```

Значения **lower-bound** и/или **upper-bound** в указании среза можно опустить. В этом случае опущенная граница заменяется нижним или верхним пределом индексов массива. Например:

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Knuth D.';
schedule
------------------------
 {{consulting},{lunch}}
```

Текущие размеры значения массива можно получить с помощью функции **array_dims**:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Knuth D.';
 array_dims
------------
 [1:2][1:2]
```

Изменение массивов

Значение массива

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

можно заменить полностью:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Knuth D.';
```

или используя синтаксис ARRAY:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Knuth D.';
```

Также можно изменить один элемент массива:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000 WHERE name = 'Virt N.';
SELECT * FROM sal_emp WHERE name = 'Virt N.';
  name   |      pay_by_quarter       |                  schedule
---------+---------------------------+-------------------------------------
 Virt N. | {10000,10000,10000,15000} | {{meeting,lunch},{training,presentation}}
```

или срез:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}' WHERE name = 'Virt N.';
SELECT * FROM sal_emp WHERE name = 'Virt N.';
  name   |      pay_by_quarter       |                 schedule
---------+---------------------------+-----------------------------------
 Virt N. | {27000,27000,10000,15000} | {{meeting,lunch},{training,presentation}}
```

Поиск значений в массивах

Чтобы найти значение в массиве, необходимо проверить все его элементы.
Это можно сделать вручную, если размер массива известен:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
  name   |      pay_by_quarter       |                 schedule
---------+---------------------------+-----------------------------------------
 Virt N. | {27000,27000,10000,15000} | {{meeting,lunch},{training,presentation}}
```

или, если размерность велика/неизвестна

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

Можно найти строки, в которых массивы содержат только значения, равные 10000:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Для поиска в массивах postgres предлагает использовать несколько функций.

`array_position(array, val[, pos])` — возвращает позицию первого вхождения `val`
`pos` — поиск с указанной позиции, если задано

`array_positions(array, val)` — возвращает массив позиций всех вхождений `val`

Составные типы

Составной тип представляется структурой табличной строки или записи.
Объявление составного типа :

```
CREATE TYPE type_name AS (
    [ attribute_name data_type  [ COLLATE colation_rule] [, ... ] ]
)
```

Это просто список имён полей и соответствующих типов данных.
Postgres позволяет использовать составные типы во многом так же, как и простые типы, например, в определении таблицы можно объявить столбец составного типа.
Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,    --
    supplier_id     integer, -- поставщик
    price           numeric  --
);
```

Синтаксис похож на **CREATE TABLE**, но допускает только названия полей и их типы.
Какие-либо ограничения (такие как NOT NULL) в настоящее время не поддерживаются.
Ключевое слово AS здесь имеет значение — без него система будет считать, что подразумевается другой тип команды CREATE TYPE, и выдаст синтаксическую ошибку.

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

или в функциях:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Когда создаётся таблица, вместе с ней автоматически создаётся составной тип.
Этот тип представляет тип строки таблицы, и его именем становится имя таблицы.
Например, при выполнении команды:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

в качестве побочного эффекта будет создан составной тип **inventory_item**, в точности соответствующий тому, что был показан выше, и использовать его можно так же.

Конструирование составных значений

Чтобы записать значение составного типа в виде текстовой константы, его поля нужно заключить в круглые скобки и разделить их запятыми. Значение любого поля можно заключить в кавычки, а если оно содержит запятые или скобки, это делать обязательно.

Таким образом, в общем виде константа составного типа записывается так:

'(val1 , val2 , ...)'

Например, запись:

'("fuzzy dice", 42, 1.99)'

будет допустимой для описанного выше типа **inventory_item**.

Чтобы присвоить **NULL** какому-либо из полей, в соответствующем месте в списке нужно оставить пустое место. Например, задаем значение **NULL** для третьего поля:

'("fuzzy dice", 42,)'

Если же требуется вставить пустую строку, нужно записать пару кавычек:

'("", 42,)'

Здесь в первом поле окажется пустая строка, а в третьем — **NULL**.

Значения составных типов также можно конструировать, используя синтаксис выражения **ROW**.

В большинстве случаев это значительно проще, чем записывать значения в строке, так как при этом не нужно беспокоиться о вложенности кавычек:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Обращение к составным типам (.)

Чтобы обратиться к полю столбца составного типа, после имени столбца нужно добавить точку и имя поля, подобно тому, как указывается столбец после имени таблицы.

На самом деле, эти обращения неотличимы, так что часто бывает необходимо использовать скобки, чтобы команда была разобрана правильно.

Например, можно попытаться выбрать поле столбца из тестовой таблицы **on_hand** так:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

Но это не будет работать, так как согласно правилам SQL имя **item** здесь воспринимается как имя таблицы, а не столбца в таблице **on_hand**. Поэтому этот запрос нужно переписать так:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

либо указать и имя таблицы[1]:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

В этом случае объект в скобках будет правильно интерпретирован как ссылка на столбец **item**, из которого выбирается поле.

При выборке поля из значения составного типа также возможны всякие синтаксические проблемы. Например, чтобы выбрать одно поле из результата функции, возвращающей составное значение, требуется написать что-то подобное:

```
SELECT (my_func(...)).field FROM ...
```

Без дополнительных скобок в этом запросе произойдёт синтаксическая ошибка.

---

1) явное всегда лучше неявного

Идентификаторы объектов

Идентификатор объекта (Object Identifier, OID) используется внутри Postgres в качестве первичного ключа различных системных таблиц.

Идентификатор объекта представляется в типе **`oid`**.

Также существуют различные типы-псевдонимы для **`oid`**, с именами **`reg<something>`**.

Имя	Ссылки	Описание	Пример значения
`oid`	`any`	числовой идентификатор объекта	`564182`
`regclass`	`pg_class`	имя отношения	`pg_type`
`regcollation`	`pg_collation`	имя правила сортировки	`"POSIX"`
`regconfig`	`pg_ts_config`	конфигурация текстового поиска	`english`
`regdictionary`	`pg_ts_dict`	словарь текстового поиска	`simple`
`regnamespace`	`pg_namespace`	пространство имён	`pg_catalog`
`regoper`	`pg_operator`	имя оператора	`+`
`regoperator`	`pg_operator`	оператор с типами аргументов	`*(integer,integer)`
`regproc`	`pg_proc`	имя функции	`sum`
`regprocedure`	`pg_proc`	функция с типами аргументов	`sum(int4)`
`regrole`	`pg_authid`	имя роли	`smithee`
`regtype`	`pg_type`	имя типа данных	`integer`

Псевдотипы

В систему типов PostgreSQL включены несколько специальных элементов, которые в совокупности называются псевдотипами.

Псевдотип нельзя использовать в качестве типа данных столбца, но можно объявить функцию с аргументом или результатом такого типа.

Каждый из существующих псевдотипов полезен в ситуациях, когда характер функции не позволяет просто получить или вернуть определённый тип данных SQL.

Имя псевдотипа	Описание
any	Указывает, что функция принимает любой вводимый тип данных.
anyelement	Указывает, что функция принимает любой тип данных.
anyarray	Указывает, что функция принимает любой тип массива.
anynonarray	Указывает, что функция принимает любой тип данных, кроме массивов.
anyenum	Указывает, что функция принимает любое перечисление.
anyrange	Указывает, что функция принимает любой диапазонный тип данных.
anymultirange	Указывает, что функция принимает любой мультидиапазонный тип данных.
anycompatible	Указывает, что функция принимает любой тип данных и может автоматически приводить различные аргументы к общему типу данных.
cstring	Указывает, что функция принимает или возвращает строку в стиле C.
internal	Указывает, что функция принимает или возвращает внутренний серверный тип данных.
record	Указывает, что функция принимает или возвращает неопределённый тип строки.
void	Указывает, что функция не возвращает значение.
unknown	Обозначает ещё не распознанный тип, например простую строковую константу.

Функции, написанные на языке C, могут быть объявлены с параметрами или результатами любого из этих псевдотипов. Ответственность за безопасное поведение функции с аргументами таких типов ложится на разработчика функции.