Базы данных

Лекция 06 – Основы SQL

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, 2023

2022.10.06

Оглавление

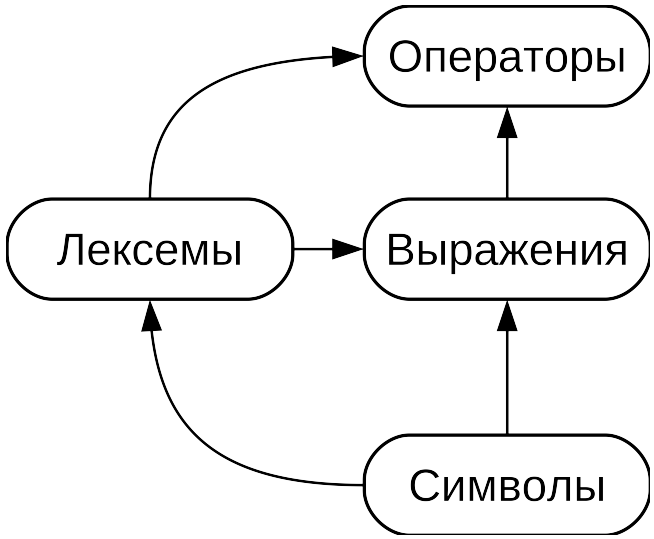
Язык SQL ... 3
- Выражения ... 3
  - Константа или непосредственное значение ... 5
  - Ссылка на столбец ... 5
  - Позиционные параметры ... 5
  - Выражение с индексом ... 6
  - Выражение выбора поля ... 7
  - Применение оператора ... 8
  - Вызов функции ... 9
  - Агрегатное выражение ... 13
  - Приведение типов ... 14
  - Применение правил сортировки ... 16
  - Скалярный подзапрос ... 20
  - Конструктор массива ... 21
  - Конструктор табличной строки ... 24
  - Правила вычисления выражений ... 25
- Типы ... 27
  - Числовые типы ... 28
  - Денежные типы ... 34
  - LC_NUMERIC & LC_MONETARY ... 36
  - Символьные типы ... 38
  - Двоичные типы данных ... 42

Язык SQL

Выражения

Состав языка программирования

Естественный язык	Алгоритмический язык
Предложения Словосочетания Слова Символы	Операторы Выражения Лексемы Символы



Алфавит языка – набор неделимых знаков (символов), с помощью которых пишутся все тексты на языке.
Лексема – элементарная конструкция, минимальная единица языка, имеющая самостоятельный смысл.
Выражение задает правила **вычисления некоторого значения**.
Оператор задает законченное описание **некоторого действия**.

В SQL используется два вида выражений:
- выражения величины (value expresions);
- табличные выражения.

Результатом табличного выражения является таблица, а результатом выражения величины (value) является некоторое значение, которое иногда называют скаляром. Выражения величины часто называют скалярными или просто выражениями. Синтаксис таких выражений позволяет вычислять значения из примитивных частей, используя арифметические, логические и другие операции.

Выражения величины применяются в SQL самых разных контекстах, например в списке результатов команды **SELECT**, в значениях столбцов в **INSERT** или **UPDATE**, в условиях поиска во многих командах.

Выражениями величины являются:

- константа или непосредственное значение;
- ссылка на столбец;
- ссылка на позиционный параметр;
- выражение с индексом;
- выражение выбора поля;
- применение оператора;
- вызов функции;
- агрегатное выражение;
- вызов оконной функции;
- приведение типов;
- применение правил сортировки;
- скалярный подзапрос;
- конструктор массива;
- конструктор табличной строки.

Кроме того, выражением значения являются скобки, которые используются для группировки подвыражений и переопределения приоритета операторов.

Также существует ещё несколько конструкций, которые можно классифицировать как выражения, хотя они не соответствуют общим синтаксическим правилам. Они обычно имеют вид функции или оператора. Пример такой конструкции — предложение **IS NULL**.

Константа или непосредственное значение

Строки, битовые строки и числа (см. LK05)

Ссылка на столбец

Ссылку на столбец можно записать в форме:

tablename **.** ***columnname***

tablename — имя таблицы или её псевдоним (определённый в предложении **FROM**).
Имя таблицы и разделяющую точку ***можно опустить***, если имя столбца уникально среди всех таблиц, упомянутых в текущем запросе, ***но не нужно***.

Позиционные параметры

Ссылка на позиционный параметр может появляться в теле определения функции или оператора.
Ссылка на позиционный параметр применяется для обращения к значению, которое передается в SQL-оператор извне.
Параметры используются в определениях SQL-функций и операторов, составляемых пользователями. Некоторые клиентские библиотеки также поддерживают передачу значений данных отдельно от самой SQL-команды, и в этом случае параметры позволяют ссылаться на такие значения. Ссылка на параметр записывается в следующей форме:

$*number*

Например, определение некоторой функции **dept**:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$   -- $1 ссылается на значение
    LANGUAGE SQL;                                 -- первого аргумента функции
```

Выражение с индексом

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

Если результат выражения является массивом, то можно извлечь определённый его элемент:

***expression* [*subscript*]**

или несколько соседних элементов («срез массива»):

***expression* [*lower_subscript* : *upper_subscript*]**

Каждый индекс сам по себе является выражением, результат которого округляется к ближайшему целому.

В общем случае выражение, результатом которого является массив, должно заключаться в круглые скобки, но их можно опустить, если выражение масива используется с индексом — это просто ссылка на столбец или позиционный параметр.

Если исходный массив многомерный, можно соединить несколько индексов.

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]                    -- позиционный параметр
(arrayfunction(a,b))[42]     -- функция, возвращающая массив
```

В последней строке круглые скобки необходимы.

Выражение выбора поля

Если результат выражения — значение составного типа, например, строка таблицы, то определённое поле этой строки можно извлечь следующим образом:

expression . fieldname

В общем случае выражение такого типа должно заключаться в круглые скобки, но их можно опустить, если выражение — это ссылка на таблицу или позиционный параметр:

```
anytable.anycolumn      -- ссылка на столбец в таблицу
$1.somecolumn           -- позиционный параметр
(rowfunction(a,b)).col3 -- функция, возвращающая строку
```

Полная ссылка на столбец — это частный случай выбора поля.
Важный особый случай — извлечение поля из столбца составного типа:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

Скобки требуются, чтобы показать, что **compositecol** — это имя столбца, а не таблицы, и что **mytable** — имя таблицы, а не схемы.
Все поля составного столбца запрашиваются с помощью *:

(compositecol) . *

Применение оператора

Существуют два возможных синтаксиса применения операторов:

```
expression operator expression  -- бинарный инфиксный оператор
operator expression             -- унарный префиксный оператор
```

где ***оператор*** соответствует синтаксическим правилам, описанным в LK05, либо это одно из ключевых слов **AND**, **OR** и **NOT**, либо полное имя оператора в форме:

```
OPERATOR ( schema . operatorname )
```

Существуют встроенные операторы и операторы, определенные системой и пользователем.
Унарный или бинарный некоторый конкретный оператор, зависит от типа оператора.

Вызов функции

Вызов функции записывается просто как имя функции (возможно, дополненное именем схемы) и список аргументов в скобках:

function_name **(** ***[expression [,*** ***expression ...]]*** **)**

Например, вычисление квадратного корня из двух:

```
sqrt(2)
```

Помимо встроенных функций пользователь может определить и другие функции. Аргументам при этом могут быть присвоены необязательные имена.

```
CREATE FUNCTION concat_lower_or_upper(a text,
                                      b text,
                                      uppercase boolean DEFAULT false)
RETURNS text AS $$
    SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Позиционная передача параметров – аргументы указываются в заданном порядке

```
CREATE FUNCTION concat_lower_or_upper(a text,
                                      b text,
                                      uppercase boolean DEFAULT false)
```

Вызовы

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
```

```
SELECT concat_lower_or_upper('Hello', 'World');  -- $3 опущен и исп. знач. по-умолчан.
 concat_lower_or_upper
-----------------------
 HELLO WORLD
```

В позиционной записи любые аргументы с определённым значением по умолчанию можно опускать справа налево.

Именная передача — для указания аргумента используется имя, которое отделяется от выражения значения символами **=>**:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World'); -- uppercase опущен
 concat_lower_or_upper
-----------------------
 hello world
```

Преимуществом такой записи является возможность записывать аргументы в любом порядке:

```
SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
```

Для обратной совместимости поддерживается и старый синтаксис с «:=»:

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
```

Смешанная передача — параметры передаются и по именам, и позиционно. При этом именованные аргументы не могут стоять перед позиционными:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
```

Аргументы **a** и **b** передаются позиционно, а **uppercase** — по имени.

Q: Зачем?

A1: Вызов стал чуть более читаемым.

A2: Для сложных функций с множеством аргументов, часть из которых имеют значения по умолчанию, именная или смешанная передача позволяет записать вызов эффективнее и уменьшить вероятность ошибок.

Агрегатное выражение

Агрегатное выражение представляет собой применение агрегатной функции к строкам, которые выбраны запросом.

Агрегатная функция сводит множество входных значений к одному выходному, например, к сумме или среднему.

Обычно строки данных передаются агрегатной функции в неопределённом порядке и во многих случаях это не имеет значения, например функция **`min`** выдаёт один и тот же результат независимо от порядка поступающих данных.

Есть некоторые агрегатные функции (**`array_agg`**, **`string_agg`**), которые выдают результаты, зависящие от порядка данных. Для таких агрегатных функций в список аргументов можно добавить предложение **`ORDER BY`**, чтобы задать нужный порядок.

Приведение типов

Приведение типа определяет преобразование данных из одного типа в другой.
Postgres воспринимает две эквивалентные формы приведения типов:

```
CAST ( expression AS type )
expression :: type
```

Запись с **CAST** соответствует стандарту SQL.
Вариант с `::` — историческое наследие PostgreSQL.

Когда приведению подвергается значение выражения известного типа, происходит преобразование типа во время выполнения. Это приведение будет успешным, только если определён подходящий оператор преобразования типов.

Явное приведение типа можно опустить, если возможно однозначно определить, какой тип должно иметь выражение, например, когда оно присваивается столбцу таблицы. В таких случаях система преобразует тип автоматически.

Однако автоматическое преобразование выполняется только для приведений с пометкой «OK to apply implicitly» в системных каталогах. Все остальные приведения должны записываться явно. Это ограничение позволяет избежать сюрпризов с неявным преобразованием.

Приведение типа можно записать как вызов функции:

```
typename ( expression )
```

Однако это будет работать только для типов, имена которых являются также допустимыми именами функций. Например, **double precision** нельзя использовать таким образом, а **float8** (альтернативное название того же типа) — можно.

При использовании любой из форм записи внутренне происходит вызов зарегистрированной функции (**CREATE CAST**), выполняющей реальное преобразование. Именем такой функции преобразования является имя выходного типа, и таким образом функциональная форма есть не что иное, как прямой вызов нижележащей функции преобразования.

При создании переносимого приложения на это поведение, конечно, не следует рассчитывать.

Запись приведения типа в функциональной форме лучше не применять, поскольку это может вызвать разного рода несоответствия и, как следствие, трудно обнаруживаемые проблемы.

Применение правил сортировки

Предложение **COLLATE** переопределяет правило сортировки выражения. Оно добавляется после выражения:

expr* COLLATE *collation

где ***collation*** — идентификатор правила, возможно дополненный именем схемы.

Предложение **COLLATE** «связывает» выражение сильнее, чем операторы, поэтому при необходимости следует использовать скобки.

Если правило сортировки явно не определено, система либо выбирает в зависимости от столбцов, которые используются в выражении, либо, если в выражении столбцов нет, переключается на установленное для базы данных правило сортировки по умолчанию.

Предложение **COLLATE** имеет два распространённых применения:

1) переопределение порядка сортировки в предложении **ORDER BY**, например:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

2) переопределение правил сортировки при вызове функций или операторов, возвращающих зависимые от локали результаты, например:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

В последнем случае предложение **COLLATE** добавлено к аргументу оператора, на действие которого нужно повлиять. Не имеет значения, к какому именно аргументу оператора или функции добавляется **COLLATE**, поскольку правило сортировки, применяемое к оператору или функции, выбирается при анализе всех аргументов, а явное предложение **COLLATE** переопределяет правила сортировки.

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

Но будет ошибкой:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

поскольку правило сортировки нельзя применить к результату оператора **>**, который имеет несравниваемый тип данных **boolean**.

Локаль

Локаль — это сочетание языковых и культурных аспектов. Они включают в себя:
- язык сообщений;
- различные наборы символов;
- лексикографические соглашения;
- форматирование чисел;
- и прочее.

В локали существуют различные категории информации, которые программа может использовать – они описаны как макросы.

LC_ALL — локаль целиком
LC_COLLATE — сортировку строк
LC_CTYPE — классы символов
LC_MESSAGES — локализированные сообщения на родном языке
LC_MONETARY — форматирование значений денежных единиц
LC_NUMERIC — форматирование не денежных числовых значений
LC_TIME — форматирование значений дат и времени

LC_COLLATE — эта категория определяет правила сравнения, используемые при сортировке и регулярных выражениях, включая равенство классов символов и сравнение многосимвольных элементов.

Эта категория локали изменяет поведение функций, которые используются для сравнения строк с учётом местного алфавита. Например, немецкая **ßß** эсцет (sharp s) рассматривается как «ss».

```
strcoll("Das Große Eszett", "Das Grosse Eszett"); //
```

LC_CTYPE — эта категория определяет интерпретацию последовательности байт в символы (например, одиночный или многобайтовый символ), классификацию символов (например, буквенный или цифровой) и поведение классов символов.

LC_MONETARY — эта категория определяет форматирования, используемое для денежных значений. Она изменяет информацию, возвращаемую функцией, которая описывает способ отображения числа, например, необходимо ли использовать в качестве десятичного разделителя точку или запятую.

LC_MESSAGES — эта категория изменяет язык отображаемых сообщений и указывает, как должны выглядеть положительный и отрицательный ответы.

LC_NUMERIC — эта категория определяет правила форматирования, используемые для не денежных значений, например, разделительный символ тысяч и дробной части (точка в англоязычных странах, и запятая во многих других).

LC_TIME — эта категория управляет форматированием значений даты и времени. Например, большая часть Европы использует 24-часовой формат, тогда как в США используют 12-часовой.

LC_ALL — всё вышеперечисленное.

Список поддерживаемых категорий можно получить с помощью утилиты оболочки **locale**

```
$ locale
LANG=ru_RU.utf8
LC_CTYPE="ru_RU.utf8"
LC_NUMERIC="ru_RU.utf8"
LC_TIME="ru_RU.utf8"
LC_COLLATE="ru_RU.utf8"
LC_MONETARY="ru_RU.utf8"
LC_MESSAGES="ru_RU.utf8"
LC_PAPER="ru_RU.utf8"
LC_NAME="ru_RU.utf8"
LC_ADDRESS="ru_RU.utf8"
LC_TELEPHONE="ru_RU.utf8"
LC_MEASUREMENT="ru_RU.utf8"
LC_IDENTIFICATION="ru_RU.utf8"
LC_ALL=
```

Скалярный подзапрос

Скалярный подзапрос — это обычный запрос **SELECT** в скобках, который возвращает ровно одну строку и один столбец. Результат такого запроса **SELECT** используется в окружающем его выражении.

В качестве скалярного подзапроса нельзя использовать запросы, возвращающие более одной строки или столбца.

Если в результате выполнения подзапрос не вернёт никаких строк, скалярный результат считается равным **NULL**.

В подзапросе можно ссылаться на переменные из окружающего запроса — в процессе одного вычисления подзапроса они будут считаться константами.

Например, следующий запрос находит самый населённый город в каждом штате:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name) FROM states;
```

Конструктор массива

Конструктор массива — это выражение, которое создаёт массив, определяя значения его элементов. Конструктор простого массива состоит из ключевого слова **ARRAY**, открывающей квадратной скобки **[**, списка выражений (разделённых запятыми), задающих значения элементов массива, и закрывающей квадратной скобки **]**. Например:

```
SELECT ARRAY[ 1, 2, 3 + 4 ];
  array
---------
 {1,2,7}
```

По умолчанию типом элементов массива считается общий тип для всех выражений, определённый по правилам, действующим и для конструкций UNION и CASE.

Можно переопределить тип по умолчанию, указав явно в конструкторе требуемый тип:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
```

Это эквивалентно тому, как привести к нужному типу каждое выражение по отдельности.

Многомерные массивы – их можно создавать, вкладывая конструкторы друг в друга. При этом во внутренних конструкторах слово **ARRAY** можно опускать:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
```

эквивалентно:

```
SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
```

Многомерные массивы должны быть прямоугольными, и поэтому внутренние конструкторы одного уровня должны создавать вложенные массивы одинаковой размерности.

Любое приведение типа, применённое к внешнему конструктору **ARRAY**, автоматически распространяется на все внутренние.

Элементы многомерного массива можно создавать не только вложенными конструкторами **ARRAY**, но и другими способами, позволяющими получить массивы нужного типа:

```
CREATE TABLE arr(f1 int[], f2 int[]);
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
```

Можно создать и пустой массив, но так как массив не может быть не типизированным, необходимо явно привести пустой массив к нужному типу:

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
```

Также возможно создать массив из результатов подзапроса. В этом случае конструктор массива записывается так же с ключевым словом **ARRAY**, за которым в круглых скобках следует подзапрос:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                              array
------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}
```

```
SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
---------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
```

Индексы массива, созданного конструктором **ARRAY**, всегда начинаются с 1.

Конструктор табличной строки

Конструктор строки таблицы — это выражение, создающее строку (кортеж), также называемую составным значением (composite value) из значений ее полей.

Конструктор строки состоит из ключевого слова **ROW**, открывающей круглой скобки, нуля или нескольких выражений (разделённых запятыми), определяющих значения полей, и закрывающей скобки:

```
SELECT ROW ( 1, 2.5, 'this is a test' ) ;
```

Если в списке более одного выражения, ключевое слово **ROW** можно опустить.

Конструктор строки поддерживает синтаксис **rowvalue.***, при этом данное выражение будет развёрнуто в список элементов.

Например, если таблица **t** содержит столбцы **f1** и **f2**, следующие операторы эквивалентны:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

Используя конструктор строк (кортежей), можно создавать составное значение для сохранения в столбце составного типа или для передачи аргумента в функцию, принимающую составной параметр.

Можно сравнивать два составных значения или проверить их с помощью **IS NULL** или **IS NOT NULL**, например:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

  -- выбрать все строки, содержащие только NULL
SELECT ROW(table.*) IS NULL FROM table;
```

Правила вычисления выражений

Порядок вычисления подвыражений не определён. В частности, аргументы оператора или функции не обязательно вычисляются слева направо или в любом другом фиксированном порядке.

Более того, если результат выражения можно получить, вычисляя только некоторые его части, тогда другие подвыражения не будут вычисляться вовсе. Например, если написать:

```
SELECT true OR somefunc();
```

функция **somefunc()** скоре всего не будет вызываться. То же самое справедливо для записи:

```
SELECT somefunc() OR true;
```

Поэтому в сложных выражениях не стоит использовать функции с побочными эффектами.

Особенно опасно рассчитывать на порядок вычисления или побочные эффекты в предложениях **WHERE** и **HAVING**, так как эти предложения тщательно оптимизируются при построении плана выполнения.

Логические выражения (сочетания **AND/OR/NOT**) в этих предложениях могут быть видоизменены любым способом, допустимым законами Булевой алгебры.

Когда порядок вычисления важен, его можно зафиксировать с помощью конструкции CASE. Например, можно запросто получить деление на ноль в предложении WHERE:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Безопасный вариант:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

Применяемая так конструкция CASE защищает выражение от оптимизации, поэтому использовать её нужно только при необходимости.

В данном случае было бы лучше решить проблему, переписав условие как **`y > 1.5*x`**.

CASE не предотвращает раннее вычисление константных подвыражений — функции и операторы, помеченные как **IMMUTABLE**[1], могут вычисляться при планировании, а не при выполнении запроса. Поэтому в примере

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

скорее всего, произойдёт деление на ноль в процессе упрощения планировщиком константного подвыражения, несмотря на то, что во всех строках в таблице x > 0, и во время выполнения ветвь ELSE никогда бы не выполнилась.

Похожие ситуации, в которых неявно появляются константы, могут возникать и в запросах внутри функций, так как значения аргументов функции и локальных переменных при планировании могут быть заменены константами. Поэтому, для защиты от рискованных вычислений вместо выражения **CASE** безопаснее использовать конструкцию **IF-THEN-ELSE**.

---

1) неизменяемый, фиксированный

Типы

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

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. Таким образом, 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 некоторых 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.

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.

Тип **bytea** поддерживает два формата ввода и вывода — шестнадцатеричный и традиционный для PostgreSQL формат спецпоследовательностей.

Строка со спецпоследовательностями начинается с буквы E (заглавной или строчной), стоящей непосредственно перед апострофом:

```
E'foo'
```

Внутри таких строк символ \' начинает C-подобные спецпоследовательности:

Входные данные принимаются в обоих форматах, а формат выходных данных зависит от параметра конфигурации **bytea_output**; по умолчанию шестнадцатеричный.

Стандарт SQL определяет другой тип двоичных данных — **BLOB (BINARY LARGE OBJECT)** — большой двоичный объект.

Его входной формат отличается от форматов **bytea**, но функции и операторы в основном те же.