Базы данных

Лекция 05 – 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, 2024

2024.03.05

Оглавление

Язык SQL

В языке SQL вместо терминов «отношение» и «переменная отношения» используются термины «таблица», а вместо терминов «кортеж» и «атрибут» — «строка» и «столбец».

Эти термины используются в стандарте языка SQL и в поддерживающих его продуктах.

SQL — язык очень большого объема. Его спецификация[1] содержит несколько тысяч страниц.

SQL не является в полной мере реляционным языком.

В языке SQL имеются операции как определения данных, так и манипулирования ими.

SQL (Structured Query Language — «язык структурированных запросов») — ***декларативный язык программирования***, применяемый для создания, модификации и управления данными в реляционной базе данных под управлением СУБД (DBMS).

Предназначен для описания, изменения и извлечения данных, хранимых в реляционных базах данных. «Чистый» SQL без современных расширений считается языком программирования не полным по Тьюрингу, но в стандарте языка есть спецификация SQL/PSM, которая предусматривает возможность его процедурных расширений.

Изначально SQL был основным способом работы пользователя с базой данных и позволял выполнять следующий набор операций:

- создание в БД новой таблицы;
- добавление в таблицу новых записей;
- изменение записей;
- удаление записей;
- выборка записей из одной или нескольких таблиц в соответствии с заданным условием;
- изменение структур таблиц.

---

1) International Organization for Standardization (ISO): Information Technology — Database Languages — SQL, Document ISO/IEC 9075:2023

Современный SQL существенно усложнился — появились возможности описания и управления такими хранимыми объектами, как индексы, представления, триггеры и хранимые процедуры.

В результате стал приобретать черты, свойственные языкам программирования.

В данный момент SQL представляет собой совокупность операторов, инструкций и вычисляемых функций. Операторы SQL делятся на:

операторы определения данных (Data Definition Language, DDL):

CREATE создаёт объект базы данных (саму базу, таблицу, представление, пользователя и т.п.);
ALTER изменяет объект;
DROP удаляет объект.

операторы манипуляции данными (Data Manipulation Language, DML):

SELECT выбирает данные, удовлетворяющие заданным условиям;
INSERT добавляет новые данные;
UPDATE изменяет существующие данные;
DELETE удаляет данные.

операторы определения доступа к данным (Data Control Language, DCL):

GRANT предоставляет пользователю (группе) разрешения на определённые операции с объектом;
REVOKE отзывает ранее выданные разрешения;
DENY задаёт запрет, имеющий приоритет над разрешением.

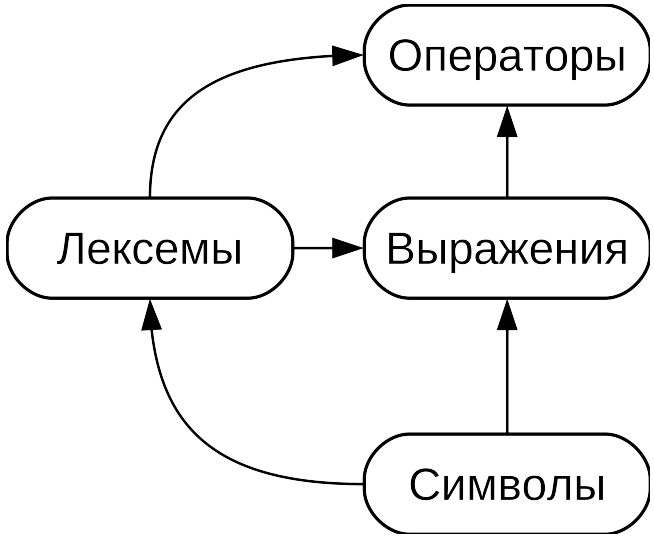
операторы управления транзакциями (Transaction Control Language, TCL):

COMMIT регистрирует транзакцию;
ROLLBACK откатывает все изменения, сделанные в контексте текущей транзакции;
SAVEPOINT делит транзакцию на более мелкие участки.

Синтаксис

Состав языка программирования

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

SQL-программа состоит из последовательности команд.
Команда — последовательность компонентов, оканчивающуюся точкой с запятой '**;**'.
Конец входного потока также считается концом команды.
Какие именно компоненты допустимы для конкретной команды, зависит от её синтаксиса.
Компонентом команды может быть:
1) ключевое слово;
2) идентификатор;
3) идентификатор в кавычках;
4) строка (или константа);
5) специальный символ.

Компоненты обычно разделяются пробельными символами (пробел, табуляция, перевод строки), но это не требуется, если нет неоднозначности, например, когда рядом с компонентом другого типа оказывается спецсимвол.

Пример

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

SQL-программы могут содержать ***комментарии*** – они не являются компонентами команд и пропускаются при интерпретации.

Идентификаторы – идентифицируют имена таблиц, столбцов или других объектов БД, в зависимости от того, где они используются. Иногда их называют «именами».

Идентификаторы и ключевые слова

Ключевые слова и идентификаторы имеют одинаковую лексическую структуру, то есть, не зная языка, нельзя определить, является ли некоторый компонент ключевым словом или идентификатором. Ключевых слов около тысячи.

Идентификаторы и ключевые слова SQL должны начинаться с буквы (**a-z**). Допускаются также не латинские буквы и буквы с диакритическими знаками или подчёркивания (_). Последующими символами в идентификаторе или ключевом слове могут быть буквы, цифры (**0-9**), знаки доллара (**$**) или подчёркивания.

Стандарт SQL не включает использование знака доллара в идентификаторах, поэтому их использование влияет на переносимость приложений.

В стандарте SQL гарантированно не будет ключевых слов с цифрами и начинающихся или заканчивающихся подчёркиванием.

Система выделяет для идентификатора не более **NAMEDATALEN-1** байт, а более длинные имена усекаются. По умолчанию **NAMEDATALEN** равно 64, так что максимальная длина идентификатора равна 63 байта. Если этого недостаточно, этот предел можно увеличить, изменив константу **NAMEDATALEN** в файле **src/include/pg_config_manual.h**.

Ключевые слова и идентификаторы без кавычек воспринимаются системой без учёта регистра.

```
UPDATE MY_TABLE SET A = 5;
update my_table set a = 5;
```

Обычно используется неформальное соглашение записывать ключевые слова заглавными буквами, а имена строчными:

```
UPDATE my_table SET a = 5;
```

Выделенные идентификаторы или идентификаторы в кавычках

Это обычный набор символов, заключенный в двойные кавычки (**"**).

Такие идентификаторы всегда будут считаться идентификаторами, но не ключевыми словами.

Таким образом **"update"** можно использовать для обозначения столбца или таблицы, в то время как **upadte** без кавычек будет воспринят как ключевое слово.

```
UPDATE "my_table" SET "a" = 5;
```

Идентификаторы в кавычках могут содержать любые символы, за исключением символа **'\0'**.

Чтобы включить в такой идентификатор кавычки, их нужно продублировать.

Кавычки позволяют создавать таблицы и столбцы с именами, которые иначе были бы невозможны, например, с пробелами или амперсандами. Ограничение длины при этом сохраняется.

Идентификатор, заключённый в кавычки, становится зависимым от регистра.

Идентификаторы без кавычек в Postgres всегда переводятся в нижний регистр, что несовместимо со стандартом SQL, где говорится о том, что имена должны приводиться к верхнему регистру.

То есть, согласно стандарту **foo** должно быть эквивалентно **"FOO"**, но не **"foo"**. Поэтому при создании переносимых приложений рекомендуется либо всегда заключать определённое имя в кавычки, либо не заключать никогда.

Идентификаторы в кавычках позволяет использовать символы Unicode по их кодам.

Такой идентификатор начинается с **U&**, а затем сразу без пробелов идёт двойная кавычка, например **U&"foo"**.

В кавычках можно записывать символы Unicode двумя способами — обратная косая черта, а за ней код символа из четырёх шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, идентификатор "**data**" можно записать так:

```
U&"d\0061t\+000061"
```

Константы

В PostgreSQL есть три типа констант подразумеваемых типов — строки, битовые строки и числа. Константы можно также записывать, указывая типы явно.

Строковые константы

Строковая константа в SQL — это обычная последовательность символов, заключённая в апострофы (**'**), например:

```
'Это строка'
```

Чтобы включить апостроф в строку, нужно использовать два апострофа рядом:

```
'Жанна д''Арк'
```

Две строковые константы, разделённые пробельными символами и *минимум* одним переводом строки, объединяются в одну и обрабатываются, как если бы строка была записана в одной константе:

```
SELECT 'foo'
'bar';
```

эквивалентно:

```
SELECT 'foobar';
```

но будет синтаксической ошибкой:

```
SELECT 'foo'      'bar';
```

Строковые константы со спецпоследовательностями в стиле C

Postgres принимает «спецпоследовательности», что является расширением стандарта SQL.

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

```
E'foo'
```

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

Спецпоследовательность	Интерпретация
\b	символ «забой»
\f	подача формы
\n	новая строка
\r	возврат каретки
\t	табуляция
\o, \oo, \ooo (o = 0–7)	восьмеричное значение байта
\xh, \xhh (h = 0–9, A–F)	шестнадцатеричное значение байта
\uxxxx, \Uxxxxxxxx (x = 0–9, A–F)	16- или 32-битный шестнадцатеричный код символа Unicode

Любой другой символ, идущий после обратной косой черты, воспринимается буквально.

Таким образом, чтобы включить в строку обратную косую черту, нужно написать две косых черты (\\). Так же можно включить в строку апостроф, написав \', в дополнение к ''.

Строковые константы, заключённые в доллары

Стандартный синтаксис для строковых констант может плохо читаться, когда строка содержит много апострофов или обратных косых черт, поскольку каждый такой символ приходится дублировать.

Чтобы и в таких случаях запросы оставались читаемыми, Postgres предлагает способ записи строковых констант — «заключение строк в доллары».

Строковая константа, заключённая в доллары, начинается со знака доллара '**$**', необязательного «*тега*» из нескольких символов и ещё одного знака доллара, затем содержит обычную последовательность символов, составляющую строку, и оканчивается знаком доллара, тем же тегом и замыкающим знаком доллара.

Например, строку «Жанна д'Арк» можно записать в долларах двумя способами:

```
$$Жанна д'Арк$$
$SomeTag$Жанна д'Арк$SomeTag$
```

Битовые строковые константы

Битовые строковые константы похожи на обычные с дополнительной буквой **B** (заглавной или строчной), добавленной непосредственно перед открывающим апострофом без пробелов:

```
B'1001'.
```

В битовых строковых константах допускаются лишь символы 0 и 1.

Битовые константы могут быть записаны в шестнадцатеричном виде, с начальной буквой **X** (заглавной или строчной):

```
X'1FF'
```

Числовые константы

<цифры>
<цифры>**.**[<цифры>][**e**[**+-**]<цифры>]
[<цифры>]**.**<цифры>[**e**[**+-**]<цифры>]
<цифры>**e**[**+-**]<цифры>

где <цифры> — это одна или несколько десятичных цифр (**0..9**).

До или после десятичной точки (при её наличии) должна быть минимум одна цифра.

Как минимум одна цифра должна следовать за обозначением экспоненты (**e**), если она присутствует.

В числовой константе не может быть пробелов или других символов.

Любой знак минус или плюс в начале строки не считается частью числа — это оператор, применённый к константе.

```
42
3.5
4.
.001
5e2
1.925e-3
```

Числовая константа, не содержащая точки и экспоненты, изначально рассматривается как константа типа **integer**, если её значение умещается в 32-битный тип **integer**, затем как константа типа **bigint**, если её значение умещается в 64-битный **bigint**, в противном случае она принимает тип **numeric**.

Константы, содержащие десятичные точки и/или экспоненты, всегда считаются константами типа **numeric**.

Константы других типов

Константу обычного типа можно ввести одним из следующих способов:

```
type 'string'               -- REAL '123.45'
'string'::type              -- '123.45'::REAL
CAST ( 'string' AS type ) -- CAST ( '123.45' AS REAL )
```

Явное приведение типа можно опустить, если нужный тип константы определяется однозначно (например, когда она присваивается непосредственно столбцу таблицы), так как в этом случае приведение происходит автоматически.

Операторы

Имя оператора представляет собой последовательность не более чем **NAMEDATALEN-1** (по умолчанию 63) символов из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ? **{[$()]}\,;:**

Однако для имён операторов есть ещё несколько ограничений:

- Сочетания символов **--** и **/*** не могут присутствовать в имени оператора (начало комментария).
- Многосимвольное имя оператора не может заканчиваться знаком **+** или **-**, если только оно не содержит также один из этих символов:

~ ! @ # % ^ & | ` ?

@- — допустимое имя оператора

***-** — недопустимое имя оператора.

Благодаря этому ограничению, Postgres может разбирать корректные SQL-запросы без пробелов между компонентами.

```
CREATE OPERATOR имя (
    {FUNCTION|PROCEDURE} = имя_функции
    [, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]
    [, COMMUTATOR = коммут_оператор ] [, NEGATOR = обратный_оператор ]
    [, RESTRICT = процедура_ограничения ] [, JOIN = процедура_соединения ]
    [, HASHES ] [, MERGES ]
)
```

$[A, B] = AB - BA, \; AB = 1.$

Пользовательские операторы

Любой оператор представляет собой просто «синтаксический сахар» для вызова нижележащей функции, которая выполняет реальную работу.

Поэтому прежде чем можно создать оператор, необходимо создать нижележащую функцию.

Однако оператор не только синтаксический сахар — он несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором.

Postgres поддерживает префиксные и инфиксные операторы.

A + B — инфиксная запись (требует скобки для смены приоритетов) // A + B * C → A + (B * C)

+ A B — префиксная запись + A * B C

A B + — постфиксная запись B C * A + или A B C * + (стековые компьютеры)

Операторы могут быть перегружены — то есть одно имя оператора могут иметь различные операторы с разным количеством и типами операндов.

Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

Специальные знаки

Некоторые не алфавитно-цифровые символы имеют специальное значение, но при этом не являются операторами.

Знак доллара '**$**', предваряющий число, используется для представления позиционного параметра в теле определения функции или подготовленного оператора.

В других контекстах знак доллара может быть частью идентификатора или строковой константы, заключённой в доллары.

Круглые скобки '**( )**' имеют обычное значение и применяются для группировки выражений и повышения приоритета операций.

В некоторых случаях скобки — это необходимая часть синтаксиса определённых SQL-команд.

Квадратные скобки '**[ ]**' применяются для выделения элементов массива.

Запятые '**,**' используются в некоторых синтаксических конструкциях для разделения элементов списка.

Точка с запятой '**;**' завершает команду SQL. Она не может находиться нигде внутри команды, за исключением строковых констант или идентификаторов в кавычках.

Двоеточие '**:**' применяется для выборки «срезов» массивов (slice).

В некоторых диалектах SQL, например, в Embedded SQL, двоеточие может быть префиксом в имени переменной.

Звёздочка '*****' используется в некоторых контекстах как обозначение всех полей строки или составного значения.

Она также имеет специальное значение, когда используется как аргумент некоторых агрегатных функций, а именно функций, которым не нужны явные параметры.

Точка '**.**' используется в числовых константах, а также для отделения имён схемы, таблицы и столбца.

Комментарии

Комментарий — это последовательность символов, которая начинается с двух минусов и продолжается до конца строки:

```
-- Это стандартный комментарий SQL
```

Кроме этого, можно записывать блочные комментарии в стиле C:

```
/*
 * многострочный комментарий
 * с вложенностью: /* вложенный блок комментария */
 */
```

Здесь комментарий начинается с `/*` и продолжается до соответствующего вхождения `*/`.

Блочные комментарии можно вкладывать друг в друга — это разрешено по стандарту SQL и позволяет комментировать большие блоки кода, которые при этом уже могут содержать блоки комментариев.

Комментарий удаляется из входного потока в начале синтаксического анализа и фактически заменяется пробелом.

Приоритеты операторов

Большинство операторов имеют одинаковый приоритет и вычисляются слева направо.

Приоритет и очерёдность операторов жёстко фиксированы в синтаксическом анализаторе.

Если необходимо, чтобы выражение с несколькими операторами разбиралось не в том порядке, который диктуют приоритеты, следует использовать скобки.

Оператор/элемент	Очерёдность	Описание
`.`	слева-направо	разделитель имён таблицы и столбца
`::`	слева-направо	приведение типов в стиле PostgreSQL
`[ ]`	слева-направо	выбор элемента массива
`+ -`	справа-налево	унарный плюс, унарный минус
`^`	слева-направо	возведение в степень
`* / %`	слева-направо	умножение, деление, остаток от деления
`+ -`	слева-направо	сложение, вычитание
`(любой другой оператор)`	слева-направо	все другие встроенные и пользовательские операторы
`BETWEEN IN LIKE ILIKE SIMILAR`		проверка диапазона, проверка членства, сравнение строк
`< > = <= >= <>`		операторы сравнения
`IS ISNULL NOTNULL`		`IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM` и т. д.
`NOT`	справа-налево	логическое отрицание
`AND`	слева-направо	логическая конъюнкция
`OR`	слева-направо	логическая дизъюнкция

Правила приоритета операторов также применяются к операторам, определённым пользователем с теми же именами, что и вышеперечисленные встроенные операторы.

Выражения

Алфавит языка – набор неделимых знаков (символов), с помощью которых пишутся все тексты на данном 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 (скалярные выражения) 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 — массив и с индексом — ссылка на таблицу 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 таблице
$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 выше, 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 даты и времени;
- 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}
```

В этом случае каждое выражение по отдельности приводится к нужному типу .

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**[2], 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**.

---

2) неизменяемый, фиксированный