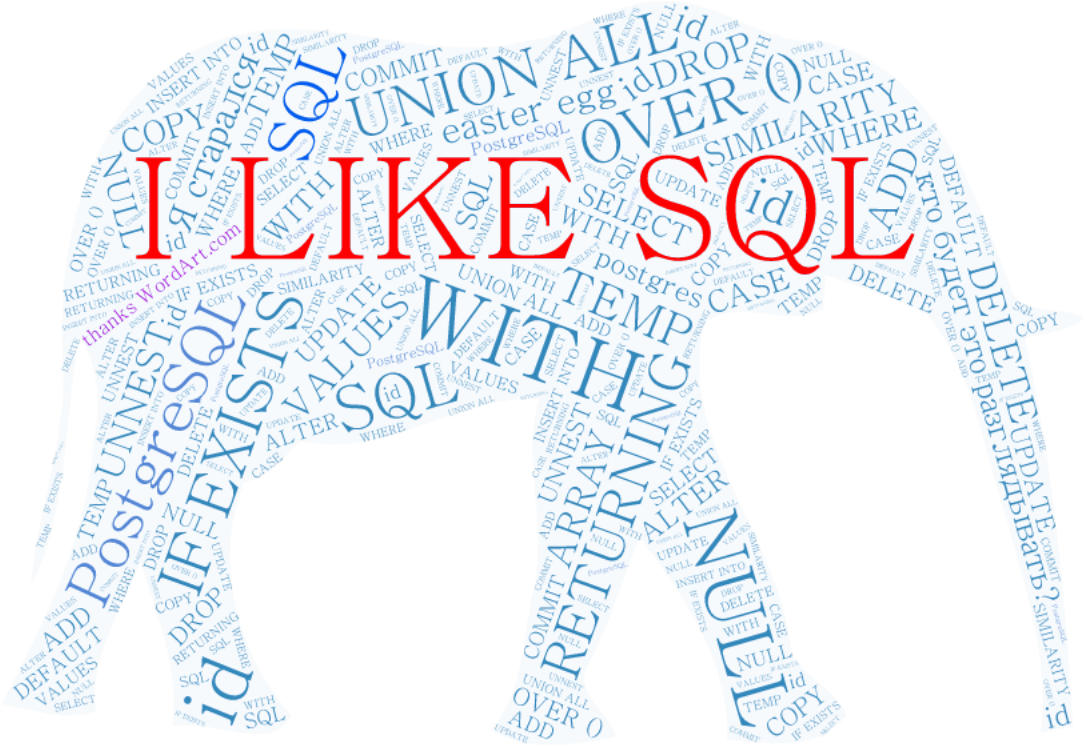
postgres 24 октября в 11:13

Курс молодого бойца PostgreSQL

Программирование, Администрирование баз данных, SQL, PostgreSQL

Tutorial



Хочу поделиться полезными приемами работы с PostgreSQL (другие СУБД имеют схожий функционал, но могут иметь иной синтак

Постараюсь охватить множество тем и приемов, которые помогут при работе с данными, стараясь не углубляться в подробное опи того или иного функционала. Я любил подобные статьи, когда обучался самостоятельно. Пришло время ~~отдать должное бесплатно интернет самообразованию и~~ написать собственную статью.

Данный материал будет полезен тем, кто полностью освоил базовые навыки SQL и желает учиться дальше. Советую выполнять и экспериментировать с примерами в **pgAdmin**'е, я сделал все SQL-запросы выполнимыми без разворачивания каких-либо дампов.

Поехали!

1. Использование временных таблиц

При решении сложных задач трудно поместить решение в один запрос (хотя, многие стараются так сделать). В таких случаях удоб помещать какие-либо промежуточные данные во временную таблицу, для использования их в дальнейшем.

Такие таблицы создаются как обычные, но с ключевым словом **TEMP**, и автоматически удаляются после завершения сессии.

Ключ **ON COMMIT DROP** автоматически удаляет таблицу (и все связанные с ней объекты) при завершении транзакции.

Пример:

```
ROLLBACK;
BEGIN;
CREATE TEMP TABLE my_fist_temp_table -- стоит использовать наиболее уникальное имя
ON COMMIT DROP -- удаляем таблицу при завершении транзакции
AS
SELECT 1 AS id, CAST ('какие-то значения' AS TEXT) AS val;
```

```
------------ Дополнительные манипуляции с таблицей: ------------------

 -- изменим таблицу, добавив столбец. Буду частенько затрагивать смежные темы
ALTER TABLE my_fist_temp_table
ADD COLUMN is_deleted BOOLEAN NOT NULL DEFAULT FALSE;
 -- для тех, кто не в курсе, чаще всего данные в таблицах не удаляются, а помечаются как удаленные подобным флагом

CREATE UNIQUE INDEX ON my_fist_temp_table (lower(val))
WHERE is_deleted = FALSE; -- можно даже создать индекс/ограничение, если это необходимо
-- данный индекс не позволит вставить дубликат(не зависимо от регистра) для столбца VAL, для не удаленных строк

-- манипулируем данными таблицы
UPDATE my_fist_temp_table
 SET id=id+3;

 -- проверяем/используем содержание таблицы
SELECT * FROM my_fist_temp_table;
--COMMIT;
```

2. Часто используемый сокращенный синтаксис Postgres

- Преобразование типов данных.

Выражение:

```
SELECT CAST ('365' AS INT);
```

можно записать менее громоздко:

```
SELECT '365'::INT;
```

- Сокращенная запись конструкции **(I)LIKE '%text%'**

LIKE воспринимает шаблонные выражения. Подробности в мануале
оператор **LIKE** можно заменить на **~~** (две тильды)
оператор **ILIKE** можно заменить на **~~*** (две тильды со звездочкой)

Поиск регулярными выражениями (имеет отличный от LIKE синтаксис)
оператор **~** (одна тильда) воспринимает регулярные выражения
оператор **~*** (одна тильда и звездочка) регистронезависимая версия **~**

Приведу пример поиска разными способами строк, которые содержат слово *text*

Сокращенный синтаксис	Описание	Аналог (I)LIKE
~ ‘text’ or ~~ ‘%text%’	Проверяет соответствие выражению с учётом регистра	LIKE '%text%'
~* ‘text’ ~~* ‘%text%’	Проверяет соответствие выражению без учёта регистра	ILIKE '%text%'
!~ ‘text’ !~~ ‘%text%’	Проверяет несоответствие выражению с учётом регистра	NOT LIKE '%text%'
!~* ‘text’ !~~* ‘%text%’	Проверяет несоответствие выражению без учёта регистра	NOT ILIKE '%text%'

3. Общие табличные выражения (CTE). Конструкция WITH

Очень удобная конструкция, позволяет поместить результат запроса во временную таблицу и тут же использовать ее.

Примеры будут примитивны, чтобы уловить суть.

a) Простой *SELECT*

```
WITH cte_table_name AS ( -- задаем удобное нам имя таблицы
SELECT schemaname, tablename -- наш любой запрос
FROM pg_catalog.pg_tables -- к примеру, системная таблица с таблицами базы
ORDER BY 1,2
)
SELECT * FROM cte_table_name; -- указываем нашу таблицу
--по факту получим результат выполнения запроса в скобках
```

Таким способом можно 'оборачивать' какие-либо запросы (даже *UPDATE, DELETE* и *INSERT*, об этом будет ниже) и использовать их результаты в дальнейшем.

b) Можно создать несколько таблиц, перечисляя их нижеописанным способом

```
WITH
table_1 (col,b) AS (SELECT 1,1), -- первая таблица
table_2 (col,c) AS (SELECT 2,2)  -- вторая таблица
--,table_3 (cool,yah) AS (SELECT 2,2 from table_2)  -- совсем недавно узнал, что можно обращаться к вышестоящей таблице
SELECT * FROM table_1 FULL JOIN table_2 USING (col);
```

c) Можно даже вложить вышеуказанную конструкцию в еще один (и более) WITH

```
WITH super_with (col,b,c) AS ( /* можем задать имена столбцов в скобках после имени таблицы */
        WITH
                table_1 (col,b) AS (SELECT 1,1),
                table_2 (col,c) AS (SELECT 2,2)
        SELECT * FROM table_1 FULL JOIN table_2 USING (col)-- указываем нашу таблицу
)
SELECT col, b*20, c*30 FROM super_with;
```

По производительности следует сказать, что не стоит помещать в секцию *WITH* данные, которые будут в значительной степени фильтроваться последующими внешними условиями (за пределами скобок запроса), ибо оптимизатор не сможет построить эффективный запрос. Удобнее всего положить в *CTE* результаты, к которым требуется несколько раз обращаться.

4. Функция array_agg(MyColumn).

Значения в реляционной базе хранятся разрозненно (атрибуты по одному объекту могут быть представлены в нескольких строках) передачи данных какому-либо приложению часто возникает необходимость собрать данные в одну строку (ячейку) или массив.

В PostgreSQL для этого существует функция **array_agg()**, она позволяет собрать в массив данные всего столбца (если выборка из о столбца).

При использовании **GROUP BY** в массив попадут данные какого-либо столбца относительно каждой группы.

Сразу опишу еще одну функцию и перейдем к примеру.

array_to_string(array[], ';') позволяет преобразовать массив в строку: первым параметром указывается массив, вторым — удобный разделитель в одинарных кавычках (апострофах). В качестве разделителя можно использовать

▾ спецсимволы

Табуляция **\t** — к примеру, позволит при вставки ячейки в EXCEL без усилий разбить значения на столбцы (использовать так: **array_to_string(array[], E'\t')**)
Перевод строки **\n** — разложит значения массива по строкам в одной ячейке (использовать так: **array_to_string(array[], E'\n')** — объясню ниже почему)

Пример:

```
-- создадим и наполним данными таблицу вышеописанным способом
WITH my_table (ID, year, any_val) AS
(
VALUES (1, 2017,56)
,(2, 2017,67)
,(3, 2017,12)
,(4, 2017,30)
,(5, 2020,8)
,(6, 2030,17)
,(7, 2030,50)
)
SELECT year
,array_agg(any_val) -- собираю данные (по каждому году) в массив
,array_agg(any_val ORDER BY any_val) AS sort_array_agg -- порядок элементов можно отсортировать (с  9+ версии Postgres)
,array_to_string(array_agg(any_val),';') -- преобразовываю массив в строку
,ARRAY['This', 'is', 'my' , 'array'] AS my_simple_array -- способ создания массива
FROM my_table
GROUP BY year; -- группируем данные по каждому году
```

Выдаст результат:

	year integer	array_agg integer[]	sort_array_agg integer[]	array_to_string text	my_simple_array text[]
1	2017	{56,67,12,30}	{12,30,56,67}	56;67;12;30	{This,is,my,array}
2	2020	{8}	{8}	8	{This,is,my,array}
3	2030	{17,50}	{17,50}	17;50	{This,is,my,array}

Выполним обратное действие. Разложим массив в строки при помощи функции **UNNEST**, заодно продемонстрирую конструкцию ***SELECT*** *columns* ***INTO*** *table_name*. Помещу это в спойлер, чтобы статья не сильно разбухала.

▾ UNNEST запрос

```
-- 1 Подготовительный этап
-- в процессе запроса будет создана таблица tst_unnest_for_del, с помощью конструкции SELECT INTO
-- чтобы запрос не приводил к ошибке, в случае если вы будете несколько раз прогонять этот скрипт, начну этот скрипт с уд
ения таблицы.
-- я также надеюсь, что вы запускаете это не на production сервере какого-либо проекта, где есть такая таблица

DROP TABLE IF EXISTS tst_unnest_for_del; /* IF EXISTS не вызовет ошибки, если таблицы для удаления не существует */

WITH
my_table (ID, year, any_val) AS (
VALUES (1, 2017,56)
,(2, 2017,67)
,(3, 2017,12)
,(4, 2017,30)
,(5, 2020,8)
,(6, 2030,17)
,(7, 2030,50)
)
SELECT year
,array_agg(id) AS arr_id -- собираю данные(id) по каждому году в массив
,array_agg(any_val) AS arr_any_val -- собираю данные(any_val) по каждому году в массив
INTO tst_unnest_for_del  -- !! способ создания и заполнения таблицы из полученного результата
FROM my_table
GROUP BY year;

--2 Демонстрирование функции Unnest
SELECT  unnest(arr_id) unnest_id -- разбираем столбец id
,year
,unnest(arr_any_val) unnest_any_val -- разбираем столбец any_val
FROM tst_unnest_for_del
ORDER BY 1 -- восстанавливаем сортировку по id, без принудительной сортировки данные могут быть расположены хаотично
```

Результат:

	unnest_id integer	year integer	unnest_any_val integer
1	1	2017	56
2	2	2017	67
3	3	2017	12
4	4	2017	30
5	5	2020	8
6	6	2030	17
7	7	2030	50

5. Ключевое слово RETURNIG *

указанное после запросов *INSERT, UPDATE* или *DELETE* позволяет увидеть строки, которых коснулась модификация (обычно сервер сообщает лишь количество модифицированных строк).

Удобно в связке с *BEGIN* посмотреть на что именно повлияет запрос, в случае неуверенности в результате или для передачи каких *id* на следующий шаг.

Пример:

```
--1
DROP TABLE IF EXISTS for_del_tmp; /* IF EXISTS не вызовет ошибки, если таблицы для удаления не существует */
CREATE TABLE for_del_tmp -- Создаем таблицу
AS       --Наполняем сгенерированными данными из запроса ниже
SELECT generate_series(1,1000) AS id, -- Генерируем 1000 пронумерованных строк
random() AS values; -- Наполняем случайными числами

--2
DELETE FROM for_del_tmp
WHERE id > 500
RETURNING *;
/*Покажет все удаленные строки данной командой,
RETURNING * - вернет все столбцы таблицы test,
так же можно перечислить столбцы как в SELECT (прим. RETURNING id,name)*/
```

Можно использовать в связке с *CTE*, организую ~~безумный~~ пример.

P.S.

Я весьма заморочился, боюсь, что вышло сложно, но я постарался все прокомментировать.

```
--1
DROP TABLE IF EXISTS for_del_tmp; /* IF EXISTS не вызовет ошибки, если таблицы для удаления не существует */
CREATE TABLE for_del_tmp -- Создаем таблицу
AS       --Наполняем сгенерированными данными из запроса ниже
SELECT generate_series(1,1000) AS id, -- Генерируем 1000 пронумерованных строк
((random()*1000)::INTEGER)::text as values; /* Наполняем случайными числами. P.S. У меня Postgre 9.2 Random() возвращает д
ное число меньше единицы, умножаю на 1000, чтобы получить целую часть, затем преобразовываю к INTEGER для избавления от др
ой части, и преобразовываю к тексту, т.к. хочу, чтобы тип данных созданного столбца был TEXT*/

--2
DELETE FROM for_del_tmp
WHERE id > 500
RETURNING *; -- Данный запрос просто удалит записи, вернув удаленные строки на экран

--3
WITH deleted_id (id) AS
        (
        DELETE FROM for_del_tmp
        WHERE id > 25
        RETURNING id -- удаляем еще часть данных, записывая id в наше CTE "deleted_id"
        )
INSERT INTO for_del_tmp -- инициируем INSERT
SELECT id, 'Удаленная строка в ' || now()::TIME || ' а если быть точным, то ' || timeofday()::TIMESTAMP /* здесь можно про
дить за тем, как отличается время возвращаемое функциями (зависит от описания функции, углубляться не буду, и так далеко з
л)*/
FROM deleted_id -- вставляем удаленные данные из "for_del_tmp" в нее же
RETURNING *; -- сразу видим что проинсертилось
--весь блок можно выполнять бесконечно, мы будем вставлять удаляемые данные в эту же таблицу.
```

```
--4
SELECT * FROM for_del_tmp; -- проверяем, что вышло в итоге
```

Таким образом, выполнится удаление данных, и удаленные значения передадутся на следующий этап. Все зависит от вашей фанта целей. Перед применением сложных конструкций обязательно изучите документацию вашей версии СУБД! (при параллельном комбинировании INSERT, UPDATE или DELETE существуют тонкости)

6. Сохранение результата запроса в файл

У команды **COPY** много разных параметров и назначений, опишу самое простое применение для ознакомления.

```
COPY (
SELECT * FROM pg_stat_activity /* Наш запрос. Для примера: системная таблица выполняемых процессов БД */
--) TO 'C:/TEMP/my_proc_tst.csv' -- Запись результата запроса в файл. Пример для Windows
) TO '/tmp/my_proc_tst.csv' -- Запись результата запроса в файл. Пример для LINUX
--) TO STDOUT -- выведет данные в консоль или лог pgAdmin
WITH CSV HEADER -- Необязательная строка. Передает название столбцов таблицы в файл
```

7. Выполнение запроса на другой базе

Не так давно узнал, что можно адресовать запрос к другой базе, для этого есть функция dblink (все подробности в мануале)

Пример:

```
SELECT * FROM dblink(
'host=localhost user=postgres dbname=postgres', /* host и user можно не указывать, если вы хотите использовать текущие */
'SELECT ''Удаленная база: '' || current_database()' /* есть свои нюансы и ограничения. Как пример, запрос передается в оди
ных кавычках, поэтому кавычки внутри запроса должны быть экранированы (в данном примере для экранирования использую две од
рных кавычки подряд). */
)
RETURNS (col_name TEXT)
UNION ALL
SELECT 'Текущая база: ' || current_database();
```

	col_name text
1	Удаленная база: postgres
2	Текущая база: test

Если возникает ошибка:

> «ERROR: function dblink(unknown, unknown) does not exist»

необходимо выполнить установку расширения следующей командой:

```
CREATE EXTENSION dblink;
```

8. Функция similarity

Функция определения схожести одного значения к другому.

Использовал для сопоставления текстовых данных, которые были похожи, но не равны друг другу (имелись опечатки). Сэкономил у времени и нервов, сведя к минимуму ручную привязку.
similarity(a, b) выдает дробное число от 0 до 1, чем ближе к 1, тем точнее совпадение.
Перейдем к примеру. С помощью *WITH* организуем временную таблицу с вымышленными данными (и специально исковерканными демонстрации функции), и будем сравнивать каждую строку с нашим текстом. В примере ниже будем искать то, что больше похож ООО «РОМАШКА» (подставим во второй параметр функции).

```
WITH company (id,c_name) AS (
VALUES (1, 'ООО РОМАШка')
UNION ALL
/* P.S. UNION ALL работает быстрее, чем UNION, т.к. отсутствует принудительная сортировка для устранения дубликатов, котор
```

```
 нам не требуется в данном случае */
VALUES (2, 'ООО "РОМАШКА"')
UNION ALL
VALUES (3, 'ООО РаМАШКА')
UNION ALL
VALUES (4, 'ОАО "РОМАКША"')
UNION ALL
VALUES (5, 'ЗАО РОМАШКА')
UNION ALL
VALUES (6, 'ООО РО МАШКА')
UNION ALL
VALUES (7, 'ООО РОГА И КОПЫТА')
UNION ALL
VALUES (8, 'ZAO РОМАШКА')
UNION ALL
VALUES (9, 'Как это сюда попало?')
UNION ALL
VALUES (10, 'Ромашка 33')
UNION ALL
VALUES (11, 'ИП "РомаШкович"')
UNION ALL
VALUES (12, 'ООО "Рома Шкович"')
UNION ALL
VALUES (13, 'ИП "Рома Шкович"')
)
SELECT *, similarity(c_name, 'ООО "РОМАШКА"')
,dense_rank() OVER (ORDER BY similarity(c_name, 'ООО "РОМАШКА"') DESC)
AS "Ранжирование результатов" -- оконная функций, о ней будет сказано ниже
FROM company
WHERE similarity(c_name, 'ООО "РОМАШКА"') >0.25 -- значения от 0 до 1, чем ближе к 1, тем точнее совпадение
ORDER BY similarity DESC;
```

Получим следующий результат:

	id integer	c_name text	similarity real	Ранжирование результатов bigint
1	1	ООО РОМАШка	1	1
2	2	ООО "РОМАШКА"	1	1
3	6	ООО РО МАШКА	0.666667	2
4	3	ООО РаМАШКА	0.6	3
5	10	Ромашка 33	0.533333	4
6	8	ZAO РОМАШКА	0.5	5
7	5	ЗАО РОМАШКА	0.5	5
8	12	ООО "Рома Шкович"	0.4	6
9	11	ИП "РомаШкович"	0.3	7
10	4	ОАО "РОМАКША"	0.263158	8

Если возникает ошибка

> «ERROR: function similarity(unknown, unknown) does not exist»

необходимо выполнить установку расширения следующей командой:

```
CREATE EXTENSION pg_trgm;
```

▾ Пример посложнее

```
WITH company (id,c_name) AS ( -- входная таблица с данными
VALUES (1, 'ООО РОМАШка')
UNION ALL
VALUES (2, 'ООО "РОМАШКА"')
UNION ALL
VALUES (3, 'ООО РаМАШКА')
UNION ALL
VALUES (4, 'ОАО "РОМАКША"')
UNION ALL
VALUES (5, 'ЗАО РОМАШКА')
UNION ALL
VALUES (6, 'ООО РО МАШКА')
UNION ALL
```

```
VALUES (7, 'ООО РОГА И КОПЫТА')
UNION ALL
VALUES (8, 'ZAO РОМАШКА')
UNION ALL
VALUES (9, 'Как это сюда попало?')
UNION ALL
VALUES (10, 'Ромашка 33')
UNION ALL
VALUES (11, 'ИП "РомаШкович"')
UNION ALL
VALUES (12, 'ООО "Рома Шкович"')
UNION ALL
VALUES (13, 'ИП "Рома Шкович"')
UNION ALL
VALUES (13, 'ООО РАГА И КАПЫТА')
),
compare (id, need) AS -- наша база для сопоставления
(VALUES (100500, 'ООО "РОМАШКА"')
UNION ALL
VALUES (9999, 'ООО "РОГА И КОПЫТА"')
)

SELECT c1.id, c1.c_name, 'сравниваем с ' || c2.need, similarity(c1.c_name, c2.need)
,dense_rank() OVER (PARTITION BY c2.need ORDER BY similarity(c1.c_name, c2.need) DESC)
AS "Ранжирование результатов" -- оконная функций, о ней будет сказано ниже
FROM company c1 CROSS JOIN compare c2
WHERE similarity(c_name, c2.need) >0.25 -- значения от 0 до 1, чем ближе к 1, тем точнее совпадение
ORDER BY similarity DESC;
```

Получим такой результат:

	id integer	c_name text	compare text	similarity real	Ранжирование результатов bigint
1	7	ООО РОГА И КОПЫТА	сравниваем с ООО "РОГА И КОПЫТА"(id 9999)	1	1
2	1	ООО РОМАШка	сравниваем с ООО "РОМАШКА"(id 100500)	1	1
3	2	ООО "РОМАШКА"	сравниваем с ООО "РОМАШКА"(id 100500)	1	1
4	6	ООО РО МАШКА	сравниваем с ООО "РОМАШКА"(id 100500)	0.666667	2
5	3	ООО РаМАШКА	сравниваем с ООО "РОМАШКА"(id 100500)	0.6	3
6	10	Ромашка 33	сравниваем с ООО "РОМАШКА"(id 100500)	0.533333	4
7	13	ООО РАГА И КПЫТА	сравниваем с ООО "РОГА И КОПЫТА"(id 9999)	0.521739	2
8	5	ЗАО РОМАШКА	сравниваем с ООО "РОМАШКА"(id 100500)	0.5	5
9	8	ZAO РОМАШКА	сравниваем с ООО "РОМАШКА"(id 100500)	0.5	5
10	12	ООО "Рома Шкович"	сравниваем с ООО "РОМАШКА"(id 100500)	0.4	6
11	11	ИП "РомаШкович"	сравниваем с ООО "РОМАШКА"(id 100500)	0.3	7
12	4	ОАО "РОМАКША"	сравниваем с ООО "РОМАШКА"(id 100500)	0.263158	8

Сортируем по *similarity DESC*. Первыми результатами видим наиболее похожие строки (1— полное сходство).

Необязательно выводить значение *similarity* в *SELECT*, можно просто использовать его в условии WHERE similarity(c_name, 'ООО «РОМАШКА»') >0.7
и самим задавать устраивающий нас параметр.

P.S. Буду признателен, если подскажете какие еще есть способы сопоставления текстовых данных. Пробовал убирать регулярными выражениями все кроме букв/цифр, и сопоставлять по равенству, но такой вариант не срабатывает, если присутствуют опечатки.

9. Оконные функции OVER() (PARTITION BY __ ORDER BY __)

Почти описав в своем черновике этот очень мощный инструмент, обнаружил ~~(с грустью и радостью)~~, что подобная качественная ст на эту тему уже существует. Не вижу смысла дублировать информацию, поэтому рекомендую обязательно ознакомиться с данной статьей (ссылка — habrahabr.ru/post/268983/, автору низкий поклон) тем, кто еще не умеет пользоваться оконными функциями SQ

10. Множественный шаблон для LIKE

Задача. Необходимо отфильтровать список пользователей, имена которых должны соответствовать определенным шаблонам.

Как всегда, представлю простейший пример:

```
-- Создаем таблицу с данными
CREATE TEMP TABLE users_tst (id, u_name)
AS (VALUES (1::INT, NULL::VARCHAR(50))
```

```
,(2, 'Ульяна Х.')
,(3, 'Семён И.')
,(4, 'Виктория Т.')
,(5, 'Ольга С.')
,(6, 'Елизавета И.')
,(7, 'Николай Х.')
,(8, 'Исаак Р.')
,(9, 'Елисей А.')
);
```

Имеем запрос, который выполняет свою функцию, но становится громоздким при большом количестве фильтров.

```
SELECT * FROM users_tst
WHERE u_name LIKE 'В%'
        OR u_name LIKE '%аа%'
        OR u_name LIKE 'Ульяна Х.'
        OR u_name LIKE 'Елисей%'
        -- и т.д.
```

Продемонстрирую, как сделать его более компактным:

```
SELECT * FROM users_tst
WHERE u_name LIKE ANY (ARRAY['В%', '%аа%', 'Ульяна Х.', 'Елисей%'])
```

Можно проделать интересные трюки, используя подобный подход.
Напишите в комментариях, если есть мысли, как еще можно переписать исходный запрос.

11. Несколько полезных функций

NULLIF(a,b)
Возникают ситуации, когда определенное значение нужно трактовать как NULL.
Например, строки нулевой длины ('' — пустые строки) или ноль(0).
Можно написать CASE, но лаконичнее использовать функцию NULLIF, которая имеет 2 параметра, при равенстве которых возвраща
NULL, иначе выводит исходное значение.

```
SELECT id
,param
,CASE WHEN param = 0 THEN NULL ELSE param END -- решение через CASE
,NULLIF(param,0) -- решение через NULLIF
,val FROM(
VALUES( 1, 0, 'В столбце слева был 0' )
) AS tst (id,param,val);
```

COALESCE выбирает первое не NULL значение

```
SELECT COALESCE(NULL,NULL,-20,1,NULL,-7); --выберет -20
```

GREATEST выбирает наибольшее значение из перечисленных

```
SELECT GREATEST(2,1,NULL,5,7,4,-9); --выберет 7
```

LEAST выбирает наименьшее значение из перечисленных

```
SELECT LEAST(2,1,NULL,5,7,4,-9); -- выберет -9
```

PG_TYPEOF показывает тип данных столбца

```
SELECT pg_typeof(id), pg_typeof(arr), pg_typeof(NULL)
FROM (VALUES ('1'::SMALLINT, array[1,2,'3',3.5])) AS x(id,arr);
-- покажет smallint, numeric[] и unknown соответственно
```

PG_CANCEL_BACKEND останавливаем нежелательные процессы в базе

```
SELECT pid, query, * FROM pg_stat_activity -- таблица с процессами БД. В старых версиях postgres столбец PID назывался PRO
D
WHERE state <> 'idle' and pid <> pg_backend_pid(); -- исключаем подключения и свой только что вызванный процесс

SELECT pg_terminate_backend(PID); /* подставляем сюда PID процесса который мы хотим остановить, в отличие от нижеприведенн
 команды, посылает более щадящий сигнал о завершении, который не всегда может убить процесс*/
SELECT pg_cancel_backend(PID); /* подставляем сюда PID процесса который мы хотим остановить. Практически гарантированно уб
ет запрос, что-то вроде KILL -9 в LINUX */
```

Подробнее в мануале

P.S.

```
SELECT pg_cancel_backend(pid) FROM pg_stat_activity -- примера ради убиваем все процессы
WHERE state <> 'idle' and pid <> pg_backend_pid();
```

Внимание! Ни в коем случае не убивайте зависший процесс через консоль KILL -9 или диспетчер задач.
Это может привести к краху БД, потере данных и долгому автоматическому восстановлению базы.

12. Экранирование символов

Начну с основ.
В SQL строковые значения обрамляются ' апострофом (одинарной кавычкой).
Числовые значения можно не обрамлять апострофами, а для разделения дробной части нужно использовать точку, т.к. запятая буд
воспринята как разделитель

```
SELECT 'Мой текст', 365, 567.6,  567,6
```

результат:

	?column? unknown	?column? integer	?column? numeric	?column? integer	?column? integer
1	Мой текст	365	567.6	567	6

Все хорошо, до тех пор пока не требуется выводить сам знак апострофа '
Для этого существуют два способа экранирования (известных мне)

```
SELECT 1, 'Апостроф '' и два апострофа подряд '''' ' -- Экранирование двойным написанием ''
UNION ALL
SELECT 2, E'Апостроф \' и два апострофа подряд \'\' ' -- экранирование обратным слешем, , английская буква E перед первой
ычкой необходима, чтобы символ \ воспринимался как символ экранирования
```

результат одинаковый:

	?column? integer	?column? text
1	1	Апостроф ' и два апострофа подряд ''
2	2	Апостроф ' и два апострофа подряд ''

В *PostgreSQL* существуют более удобный способ использовать данные, без экранирования символов. В обрамленной двумя знакам
доллара **$$** строке можно использовать практически любые символы.

Пример:

```
select $$необязательно писать '' чтобы просто вывести апостроф ', или заморачиваться с E'\' $$
```

получаю данные в первозданном виде:

	?column? unknown
1	необязательно писать '' чтобы просто вывести апостроф ', или заморачиваться с E'\'

Если этого мало, и внутри требуется использовать два символа доллара подряд **$$**, то Postgres позволяет задать свой «ограничите
Стоит лишь между двумя долларами написать свой текст, например:

```
select  $uniq_tAg$ необязательно писать '' чтобы просто вывести апостроф ', или заморачиваться с E'\', обрамляйте в $$ или
ny_text$  $uniq_tAg$
```

Увидим наш текст:

	?column? unknown
1	необязательно писать '' чтобы вывести апостроф ', или заморачиваться с E'\', обрамляйте в $$ или $any text$

Для себя этот способ открыл не так давно, когда начал изучать написание функций.

Заключение

Надеюсь, данный материал поможет узнать много нового начинающим и «средничкам». Сам я не являюсь разработчиком, а могу л
назвать себя любителем SQL, поэтому то, как использовать описанные приемы — решать Вам.

Желаю успехов в изучении SQL. Жду комментариев и благодарю за прочтение!

Метки: SQL, Postgres, PostgreSQL, базы данных, СУБД, tips and tricks

26,0 Карма 60,0 Рейтинг 12 Подписчики

Владимир Голяткин @postgres
Ведущий специалист по внедрению ПО

Комментарии 49

ploop 24.10.17 в 12:21

Про similarity не знал. Могло бы сэкономить кучу времени в некоторых задачах, например, со строковыми адресами.

ilyaplot 24.10.17 в 17:14

Как раз решаю сейчас задачи, с которыми similarity справляется на ура. Спасибо, автор! ~~Пошел удалять лишний код.~~
Думаю, с приведением типов любой разработчик, использующий postgres рано или поздно столкнется, а вот про функции, подобные simi
жду следующий пост.

postgres 24.10.17 в 19:42

Рад, что статья популярна не только из-за картинки (хотя, над ней я тоже попотел).
similarity и для меня была 'палочкой выручалочкой', грустно, когда и не предполагаешь, что СУБД может такое 'вытворять'.
Я бы с удовольствием выпустил «Часть 2», знать бы в сторону каких функций и фишек смотреть...

Art_mik 25.10.17 в 11:56

Курсоры — www.postgresql.org/docs/9.6/static/plpgsql-cursors.html
Процедурный язык PL/pgSQL — www.postgresql.org/docs/current/static/plpgsql.html
Я считаю, что на это стоит обратить внимание. Я бы сказал, что без использования этих вещей postgresql и не postgresql вовсе.

dgstudio 24.10.17 в 20:19

Откройте для себя товарища Левенштейна https://postgrespro.ru/docs/postgresql/9.4/fuzzystrmatch.html

postgres 24.10.17 в 21:17

Самое забавное, что раньше (на postgres 8.*) я использовал самописную функцию *similarity* в теле которого была функция *Левенште*
(на 9.2 она затерлась, и я думал, что разница в результатах появилась из-за новой версии СУБД).
Я вырезал участок своего негодования из статьи, т.к. он даже не совпадал с документацией postgres 8, а перепроверив тело функци
понял свою ошибку.
В спойлере можно почерпнуть разницу между **similarity** и **levenshtein**

Вырезанный участок, имеются ошибочные домыслы

P.S. Только жаль, что **SOUNDEX** не очень подходит для русского языка.

geminirff 25.10.17 в 10:16

О, как! Есть готовое. Сам то всегда реализовывал такую задачу методом Q-грамм (Би-грамм если быть точным). Надеюсь, производительность реализации позволит в лоб сопоставить две таблицы хоты бы 10к в каждой.

ploop 25.10.17 в 11:41

«в лоб» будет медленно, т.е. надо для каждой строки из первой таблицы найти по всем записям similarity второй таблицы, сортирнуть ней, и взять с наивысшим соответствием. Вот именно так я вчера и проверял, у меня уходило секунд ~6 на запись (limit 10 всего запро работал минуту).

Правда у меня объём был 230к, зато реальных данных. И связка очень красиво так получилась! Адреса разной структуры — в одном опущена область, в другом индекс, первый структурированный с разделителем, второй ручной ввод.

Естественно некий процент ошибок будет, но это лучше, чем всё лопатить руками.

postgres 25.10.17 в 11:54

А попробуйте создать индекс для столбца по триграммам www.postgresql.org/docs/9.1/static/pgtrgm.html

geminirff 25.10.17 в 12:06

Ну! 6 сек, на 230к строк ... да небось в Unicode, со средней длиной ~50 символов – это круто!

ploop 25.10.17 в 14:27

Может и не 6, а 10… что-то засомневался, доберусь до БД проверю.
Сама функция работает мгновенно, т.е. `select field` и `select similarity(field, 'static text')` работает одинаково практиче

roveo 24.10.17 в 12:35

> Проверяет соответствие регулярному выражению с учётом регистра

LIKE использует не регулярные выражения, а свой синтаксис шаблонов. Кто это знает, понял, что имелось в виду, а кто не знает, мог подума что туда надо писать regexp.

postgres 25.10.17 в 21:35

Действительно, вышло не так как я хотел. Поправил

akhkmed 24.10.17 в 13:09

Про временные таблицы есть ограничения, связанные с хранением версий строк: очень интенсивное их использование приводит к распуха системных таблиц и снижению производительности, аналогично постоянным update обычных таблиц.
Как альтернативу временным таблицам могу предложить переменные типа jsonb или array композитных типов, правда индексы тут уже не применить, в отличие от временных таблиц. К слову, jsonb делает работу с данными в plpgsql гораздо удобнее, рекомендую взять на вооружение.

postgres 24.10.17 в 15:11

Спасибо, не знал о таком! Моя цель была просто продемонстрировать возможности, и подбить профессионалов на написание подобных статей.

dellert 24.10.17 в 15:27

для типа jsonb возможно применить индексы типа GIN

Дока

ilyaplot 24.10.17 в 17:17

Индексы в jsonb? Легко.

```
CREATE INDEX station_synonyms_idx
  ON station
  USING btree
  ((additional_data #>> '{synonyms}'::text[]) COLLATE pg_catalog."default");
```

Это индекс по текстовому массиву. Думаю, нет смысла приводить примеры для простых строковых или числовых индексов.

akhkmed 24.10.17 в 17:40

По колонке в таблице — легко, а тут вместо temp table предлагаю переменную типа jsonb для промежуточного хранения. Если внутри а просто dict, то вытаскивание значения по ключу происходит быстро.

Но если в переменной большой массив dict-ов, то как из него вытащить пару значений по каким то условиям, не перебрав его целико Индексы на переменные типа jsonb не повешать.

shurutov 24.10.17 в 14:31

dblink — не функция, а расширение. Странно, что всплыло именно оно, а не postgres_fdw: https://www.postgresql.org/docs/9.6/static/postgres-fdw.html

ploop 24.10.17 в 15:21

postgres_fdw, если память не изменяет, в 9.3 появился? А в более ранних версиях только dblink.

akhkmed 24.10.17 в 17:55

Если не ошибаюсь, в dblink в плане транзакций довольно странно: транзакциями наверное можно управлять явно на уровне выражени документация этот момент не описывает. В fdw управление транзакциями происходит автоматически.
Для удалённого вызова функций удобнее использовать pl/proxy или plexor вместо fdw. В первом, насколько знаю, также нет автоматического управления транзакциями, во втором оно автоматическое.

Taragolis 24.10.17 в 14:51

Еще надо не забывать, что функции GREATEST и LEAST, в отличии от реализации в Oracle и DB2 LUW, игнорируют NULL.

```
SELECT GREATEST(1, 2, 3, NULL, 4, 5) -- 5
```

```
SELECT GREATEST(1, 2, 3, NULL, 4, 5) FROM DUAL; -- NULL
```

```
SELECT GREATEST(1, 2, 3, NULL, 4, 5) FROM SYSIBM.DUAL; -- NULL
```

ploop 27.10.17 в 10:16

Вот, кстати, прямо сейчас наткнулся на проблему с GREATEST и NULL. По задаче надо выбрать все положительные числа, вместо отрицательного — ноль. Но с сохранением NULL, а она не сохраняет его.
На скорую руку можно обойтись таким костылём:

```
SELECT GREATEST(field, 0) * (field::integer::boolean::integer)
```

Пояснение:
Тип поля — numeric, сначала преобразуется в integer, затем в boolean затем опять в integer. В итоге при любом значении поля, отличном о нуля, получим единицу на выходе. Или NULL, если field IS NULL.

Taragolis 27.10.17 в 10:41

Крутое решение, хотя я бы все же сделал через старый добрый CASE .. THEN .. ELSE .. END, тогда результат был такой же, однако т кто потому будет доробатовать это дело не пришлось бы думать, а что вот здесь вот происходит

```
WITH TEST_DATA as (
  SELECT unnest(ARRAY[1,NULL,42,-3,0,2,-15,NULL,55])::int field
)
SELECT
  field
  , GREATEST(field, 0) * (field::integer::boolean::integer) option_1
  , CASE WHEN field < 0 THEN 0 ELSE field END option_2
FROM TEST_DATA
```

ploop 27.10.17 в 12:46

Естественно, если это код будет храниться. Если это разовый наколеночный запрос иногда костыли выручают. Ну и пример ниже придётся довольно сложно раскладывать

```
select field1 * (a>b)::integer + field2 * (c>d)::integer + field3 * (e = f)::integer
```

Собственно, вопрос в том, нет ли что-то похожего на greatest/least с учетом null? Нагуглить сходу не удалось.

Taragolis 27.10.17 в 14:10

Ну... можно написать свои функции, я такие себе собираю, когда нужно быстро мигрировать с ORACLE с сохранением бизнес лог наименьшими потерями

К примеру в данном случае можно использовать что-то навроде такого

```
CREATE OR REPLACE FUNCTION f_array_has_null (ANYARRAY)
  RETURNS bool LANGUAGE sql IMMUTABLE AS
 'SELECT array_position($1, NULL) IS NOT NULL';

CREATE FUNCTION f_least_ora(VARIADIC arr numeric[])
  RETURNS numeric LANGUAGE plpgsql IMMUTABLE AS $$
    BEGIN
      IF f_array_has_null($1) THEN
        RETURN NULL;
      ELSE
        RETURN (SELECT min(x) FROM unnest($1) x);
      END IF;
    END
$$ ;
```

```
CREATE FUNCTION f_greatest_ora(VARIADIC arr numeric[])
  RETURNS numeric LANGUAGE plpgsql IMMUTABLE AS $$
    BEGIN
      IF f_array_has_null($1) THEN
        RETURN NULL;
      ELSE
        RETURN (SELECT max(x) FROM unnest($1) x);
      END IF;
    END
$$ ;
```

Ну и как результат

```
SELECT
    least(1, NULL, 42, -3, 0, -0.5, -15, NULL, 55)          -- -15
  , f_least_ora(1, NULL, 42, -3, 0, -0.5, -15, NULL, 55)    -- NULL
  , least(1, 42, -3, 0, -0.5, -15, 55)                      -- -15
  , f_least_ora(1, 42, -3, 0, -0.5, -15, 55)                -- NULL
  , greatest(1, NULL, 42, -3, 0, -0.5, -15, NULL, 55)       -- 55
  , f_greatest_ora(1, NULL, 42, -3, 0, -0.5, -15, NULL, 55) -- NULL
  , greatest(1, 42, -3, 0, -0.5, -15, 55)                   -- 55
  , f_greatest_ora(1, 42, -3, 0, -0.5, -15, 55)             -- 55
```

darthunix 24.10.17 в 15:33

А вот за абзац про экранирование строки через $$ вам от меня благодарность! Я писал функции и не понимал, что просто описываю тело функции в виде обычного текстового поля в ddl команде `create function .... as $$ ... $$`. По факту я могу смело писать

```
do language plpgsql 'begin select 1; end';
```

вместо идущего в примерах

```
do language plpgsql $$begin select 1; end$$;
```

ведь это одно и то же.

Tishka17 24.10.17 в 16:46

~ — это сокращенная запись LIKE?
Мне казалось, тильда делает поиск по регулярному выражению, а LIKE только учитывает проценты/черточки

postgres 24.10.17 в 17:28

Согласен, неверно выразился. Я имел в виду утверждение, что LIKE '%text%' выдаст такой же результат, что ~ 'text'

ploop 25.10.17 в 08:13

А разве не две тильды соответствует LIKE? То есть LIKE '%text%' = ~~'%text%'
С одной тильдой во-первых не будет работать, во-вторых в исходниках представлений LIKE автоматически заменяется на две тильды, со звёздочкой соответственно.

QuickJoey 24.10.17 в 16:51

Если есть NULLIF, то можно добавить и COALESCE, выбирает первое значение отличное от NULL.

```
SELECT COALESCE(NULL, NULL, 1, 2, 3, NULL, 4, 5); -- 1
```

postgres 25.10.17 в 20:57

Добавил COALESCE и еще парочку

ploop 26.10.17 в 01:04

Ну, думаю, COALESCE знаком всем, кто плотно работает с PostgreSQL. Конструкции вида `WHERE COALESCE(field, 0) = 0` и подобные встречаются повсеместно, если тип поля допускает NULL. А на разовых запросах позволяет не вспоминать, что там за поле что оно допускает.

QuickJoey 27.10.17 в 17:58

Там же «курс молодого бойца», а не опытного ;-)
Кстати, я стараюсь не допускать идентичности NULL и 0 (NULL и ''). Так удобнее, чтобы пусто было пусто, а 0 может что-то значить. WHERE, соответственно пишу FieldValue IS NULL. А при заполнении полей сначала привожу переменные _variable:=NULLIF(_variable,'' в триггере причёсываю значения полей.

ploop 27.10.17 в 19:37

> Кстати, я стараюсь не допускать идентичности NULL и 0

Да все стараются, как вы сказали, «с опытом». NULL разрешается только там, где он логически необходим, а на практике это не уж и часто.
Но структуры достаются по наследству, либо сам не очень удачно спроектировал и т.д., когда на рефакторинг ресурсов нет, име что имеем...

pumbo 25.10.17 в 08:10

Ещё хотелось бы добавить команду DO — выполнение анонимного блока кода. Бывает полезно когда нужно разово (по-быстрому) выполнит какие-то действия в транзакции, без создания отдельной функции.

```
DO $$
DECLARE
  -- переменные
BEGIN
  -- блок кода
  -- * транзакция запускается автоматически
  -- * для вывода данных удобно использовать RAISE NOTICE 'Data: %', foo;
END$$;
```

pensnarik 25.10.17 в 09:09

Зачем в 8 примере в запросе используются **UNION ALL**? С помощью **VALUES** можно выбрать сразу несколько строк:
▸ SQL

postgres 25.10.17 в 09:29

Напомню, что моя цель была показать как можно больше рабочих моментов.
Хотелось затронуть свойство **UNION ALL**. Согласен, там было бы уместнее SELECT вместо VALUES, но в примере 4 я показал, что можно через запятую VALUES перечислять.

PaulZi 25.10.17 в 10:39

Еще пара полезных штук:
1) Например вы используете составной ключ, и вам надо найти некоторые строки IN может работать с несколькими колонками:

```
SELECT * FROM product_attribute WHERE (product_id, attribute_id) IN ((1, 11), (2, 12), (2, 13))
```

2) Конструкцию VALUES удобно иногда использовать в FROM и JOINах:

```
SELECT tmp.dig, tmp.name FROM (VALUES (1, 'one'), (2, 'two'), '3, 'three')) as tmp(dig, name)
```

RedWolf 25.10.17 в 22:33

R-U-T 26.10.17 в 08:38

Уважаемые знатоки postgres, есть небольшая задачка, подскажите как правильно решить её с помощью postgres(сейчас использую свою коленнописную фурту на python для такого расчета).
Вообщем задачка:
Есть простая таблица с колонками id и price. В данных колонка записываются результаты примерно в таком порядке:
id price
1 45
2 80
3 75
4 125
5 50
6 70

7 22
8 23
9 47
10 20
Необходимо: отобразить id всех колонок, где (price)значение или сума сложения которых(-ой) будут равны, например, = 125
В приведенном случае это будут колонки: 4, 1+2, 3+5, 3+8+9 и так в порядке усложнения.
Подскажите пожалуйста более правильный вариант.
Заранее благодарен сообществу.
Автору спасибо большое за статью.

Yahweh 26.10.17 в 09:21

Что то я сомневаюсь что это задача уровня бд

PS 3+8+9 = 145 :)

R-U-T 27.10.17 в 17:14

Да, немного подошибся:)

vazir 26.10.17 в 09:37

Повесьте триггер на инсерт и пересчитывайте. Результаты можно в отдельнкю таблицу. Вариантов вобщем то миллион

R-U-T 27.10.17 в 17:15

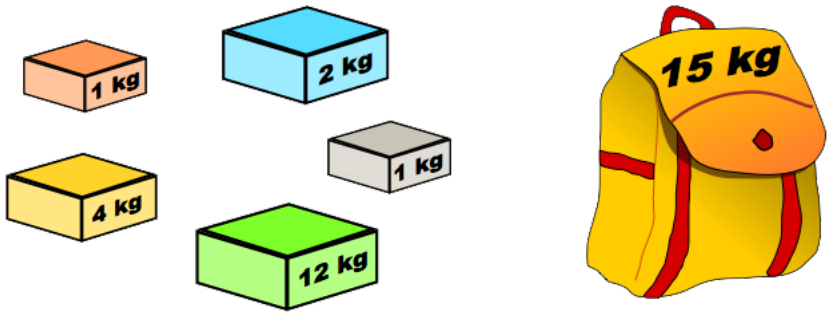
Тригер не подходит. По скольку мне нужно делать данные вычитывания только тогда, когда есть не плановое событие.

ploop 27.10.17 в 17:34

Примерный алгоритм вам ниже подсказали, а так PL/pgSQL поддерживает циклы FOR, WHILE, и даже FOREACH. Вот и вперёд :)

postgres 26.10.17 в 19:49

Очень похоже на одну из вариаций *задачи о Рюкзаке* (*Knapsack problem*) Wiki



zoroda 29.10.17 в 17:00

Решал такое: https://prosql.github.io/summ/

Taragolis 27.10.17 в 11:32

Ох, еще вспомнил, что с `to_date(text, text)` надо быть осторожным, так как PostgreSQL спокойно скушает `to_date('30.02.2017', 'dd.mm.yyyy')` и вернет 2 марта 2017, когда Oracle и DB2 LUW вернет ошибку.

В таком случае, если позволяет задача переключится на другой `datestyle` и сделать простое приведение типов

```
sql> set datestyle to DMY
sql> SELECT '30.02.2017'::date
[22008] ERROR: date/time field value out of range: "30.02.2017" Position: 8
```

> Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Про туалетную бумагу, DevOps и 582 банка

+7 648 3 5

«Иногда приходится заглядывать в код Spark»: Александр Морозов (SEMrush) об использовании Scala, Spark и ClickHouse

+12 495 2 1

Чем хорош (и чем плох) Typescript: опыт UI-разработчиков

+16 1,9k 11 8

Пошаговая настройка Graylog2

+7 779 18 3

Новинка от Aftershokz — обзор новой гарнитуры с костной проводимостью Trekz Air GT

+11 1k 2 1