

Сборник заметок по наиболее полезным конструкциям PostgreSQL

Содержание

1	Общие сведения	2
2	Сброс пароля для psql и pgAdmin4	2
3	Логический порядок обработки инструкции SELECT	2
4	Смена схемы базы данных	3
5	Создание таблицы	3
5.1	Базовые синтаксис создания таблицы	3
5.2	Создать таблицу по образу другой таблицы	4
6	Копирование данных между файлом и таблицей	4
7	Обновление записей. Команда UPDATE	5
8	Общие табличные выражения	5
8.1	Конструкция WITH	5
8.2	Конструкция WITH RECURSIVE	7
8.3	Изменение данных в WITH	9
9	Работа со строками и регулярные выражения	10
10	Приемы работы в pgAdmin 4	12
11	Приемы работы в psql	12
11.1	Конфигурационный файл	13
11.2	Метакоманды psql	13
11.3	Примеры использования	14
12	Функции, возвращающие множества	14
12.1	Функции, генерирующие ряды значений	14
	Список литературы	15

1. Общие сведения

2. Сброс пароля для psql и pgAdmin4

Для того чтобы доступ к базам данных через терминальный клиент `psql` или через web-интерфейс `pgAdmin4` можно было выполнять без ввода пароля, нужно сделать следующее:

- о найти файл `pg_hba.conf`; на ОС Windows он располагается по адресу `C:\Program Files\PostgreSQL\11\data`,
- о заменить в этом файле метод `md5` (в нижней части файла) на `trust`.

После исправлений файл `pg_hba.conf` должен выглядеть приблизительно так

pg_hba.conf

```
...
# TYPE DATABASE USER ADDRESS METHOD
# IPv4 local connections:
host all all 127.0.0.1/32 trust
host all all 0.0.0.0/0 trust
# IPv6 local connections:
host all all ::1/128 trust
host all all ::0/0 trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
host replication all 127.0.0.1/32 trust
host replication all ::1/128 trust
host appdb app all trust
```

3. Логический порядок обработки инструкции SELECT

Порядок обработки инструкции `SELECT` определяет, когда объекты, определенные в одном шаге, становятся доступными для предложений в последующих шагах. Например, если обработчик запросов можно привязать к таблицам или представлениям, определенным в предложении `FROM`, эти объекты и их столбцы становятся доступными для всех последующих шагов.

Общая процедура выполнения `SELECT` следующая (подробности см. в документации [SELECT](#)):

1. `WITH`: выполняются все запросы в списке `WITH`; по сути они формируют временные таблицы, к которым затем можно обращаться в списке `FROM`; запрос в `WITH` выполняется только один раз, даже если он фигурирует в списке `FROM` неоднократно,
2. `FROM`: вычисляются все элементы в списке `FROM` (каждый элемент в списке `FROM` представляет собой реальную или виртуальную таблицу); другими словами конструируются таблицы из списка `FROM`,
3. `ON`: выбираются строки, удовлетворяющие заданному условию,
4. `JOIN`: выполняется объединение таблиц,
5. `WHERE`: исключаются строки, не удовлетворяющие заданному условию,
6. `GROUP BY`: вывод разделяется по группам строк, соответствующим одному или нескольким значениям, а затем вычисляются результаты агрегатных функций,
7. `HAVING`: исключаются группы, не удовлетворяющие заданному условию,
8. `SELECT`,

9. **DISTINCT**: исключаются *повторяющиеся* строки; **SELECT DISTINCT ON** исключает строки, совпадающие по всем указанным выражениям; **SELECT ALL** (по умолчанию) возвращает все строки результата, включая дубликаты,
10. **UNION**, **INTERSECT** и **EXCEPT**: объединяется вывод нескольких команд **SELECT** в один результирующий набор.
11. **ORDER BY**: строки сортируются в указанном порядке; в отсутствие **ORDER BY** строки возвращаются в том порядке, в каком системе будет проще их выдавать,
12. **LIMIT** (или **FETCH FIRST**), либо **OFFSET**: возвращается только подмножество строк результата.
13. Если указано **FOR UPDATE**, **FOR NO KEY UPDATE**, **FOR SHARE** или **FOR KEY SHARE**, оператор **SELECT** блокирует выбранные строки, защищая их от одновременных изменений.

4. Смена схемы базы данных

Вывести список доступных схем

```
SHOW search_path;
```

Задать схему

```
SET search_path TO new_schema;
```

или, если требуется доступ к нескольким схемам

```
SET search_path TO new_schema1, new_schema2, public;
```

5. Создание таблицы

5.1. Базовые синтаксис создания таблицы

Для создания таблиц в языке SQL служит команда **CREATE TABLE**. Упрощенный синтаксис таков

```
CREATE TABLE table_name(  
    field_name data_type [constraint],  
    field_name data_type [constraint],  
    ...  
    [constraint],  
    [primary key],  
    [foreign key]  
);
```

Пример

```
CREATE TABLE aircrafts(  
    aircraft_code CHAR(3) NOT NULL,  
    model TEXT NOT NULL,  
    range INTEGER NOT NULL,  
    CHECK (range > 0),           -- ограничение  
    PRIMARY KEY (aircraft_code) -- первичный ключ  
);
```

5.2. Создать таблицу по образу другой таблицы

Создать таблицу со структурой данных, аналогичной другой таблице (но без ограничений базовой таблицы) можно так

```
CREATE TABLE tbl_name AS
  SELECT * FROM base_tbl LIMIT 0;
```

или более короткий вариант

```
CREATE TABLE tbl_name(LIKE base_tbl);
```

6. Копирование данных между файлом и таблицей

Скопировать данные из внешнего файла в таблицу (по сути загрузить данные) можно с помощью команды COPY. С помощью этой же команды можно записать данные из таблицы в файл.

Общий синтаксис команды выглядит так

```
-- копирует содержимое файла в таблицу
COPY имя_таблицы [ ( имя_столбца [, ...] ) ]
  FROM { 'имя_файла' | PROGRAM 'команда' | STDIN }
  [ [ WITH ] ( параметр [, ...] ) ]

-- копирует содержимое таблицы в файл
COPY { имя_таблицы [ ( имя_столбца [, ...] ) ] | ( запрос ) }
  TO { 'имя_файла' | PROGRAM 'команда' | STDOUT }
  [ [ WITH ] ( параметр [, ...] ) ]
```

Здесь допускается параметр:

```
FORMAT имя_формата
OIDS [ boolean ]
FREEZE [ boolean ]
DELIMITER 'символ_разделитель'
NULL 'маркер_NULL'
HEADER [ boolean ]
QUOTE 'символ_кавычек'
ESCAPE 'символ_экранирования'
FORCE_QUOTE { ( имя_столбца [, ...] ) | * }
FORCE_NOT_NULL ( имя_столбца [, ...] )
FORCE_NULL ( имя_столбца [, ...] )
ENCODING 'имя_кодировки'
```

Примеры

```
-- сохранить данные из таблицы 'family' в файл 'family.csv'
postgres=# COPY family TO 'E:/[WorkDirectory]/GARBAGE/family.csv' DELIMITER ',';

-- загрузить в таблицу 'family' данные из файла 'family.csv'
postgres=# CREATE TABLE family (
    person TEXT PRIMARY KEY,
    parent TEXT REFERENCES family -- создать внешний ключ на person
);
postgres=# COPY family FROM 'E:/[WorkDirectory]/GARBAGE/family.csv' DELIMITER ',';
```

7. Обновление записей. Команда UPDATE

Изменить слово Drama на Dramatic в столбце kind таблицы films

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Изменить значение температуры и сбросить уровень осадков к значению по умолчанию в одной строке таблицы weather

```
UPDATE
  weather
SET
  temp_lo = temp_lo + 1,
  temp_hi = temp_lo + 15,
  prcp = DEFAULT
WHERE
  city = 'San Francisco' AND
  dates = '2003-07-03';
```

8. Общие табличные выражения

В конструкциях общих табличных выражений с WITH имена временных таблиц указываются без перечисления имен столбцов, а в конструкциях с WITH RECURSIVE – с перечислением, например, `WITH RECURSIVE tab(col1, col2, ...) AS (...)`.

8.1. Конструкция WITH

Основное предназначение SELECT в предложении WITH (Common Table Expression, Общие Табличные Выражения) заключается в разбиении сложных запросов на простые части. Например, пусть задана некоторая таблица orders¹

```
WITH --part 1, common table expression
  regional_sales AS ( --def temp_table1
    SELECT region, sum(amount) AS total_sales
    FROM orders --base table
    GROUP BY region
  ),
  top_regions AS ( --def temp_table2
    SELECT region
    FROM regional_sales --temp_table1
    WHERE total_sales > (
      SELECT SUM(total_sales)/10
      FROM regional_sales --temp_table2
    )
)
SELECT --part 2
  region,
  product,
  SUM(quantity) AS product_units,
  SUM(amount) AS product_sales
FROM orders
WHERE region IN (
  SELECT region
  FROM top_regions --temp_table2
)
```

¹См. документацию PostgreSQL <https://postgrespro.ru/docs/postgrespro/9.5/queries-with>

```
GROUP BY region, product;
```

Здесь в инструкции WITH объявляются две *временные таблицы* `regional_sales` и `top_regions`. Вторая временная таблица `top_regions` ссылается на временную таблицу `regional_sales`, сформированную в первых строках настоящего запроса. Во второй части запроса также используется временная таблица `top_regions`.

Еще один пример. Пусть задана таблица

```
# SELECT * FROM test_tab;
id |   cae_name   | solver | num_cores
 1 | ANSYS        | Direct |        32
 3 | Comsol       | Direct |        16
 4 | LMS Virtual Lab | Direct |        32
 2 | Nastran      | Iterativ |        16
(4 строки)
```

Требуется выяснить сколько CAE-пакетов имеют прямой, а сколько итерационный решатель. Эту задачу можно решить следующим образом

```
WITH sub_tab AS ( --make temp table
    SELECT solver, 1 AS count
    FROM test_tab
)
SELECT solver, sum(count)
FROM sub_tab --link to temp table
GROUP BY solver;
```

Часть с WITH возвращает

```
# SELECT solver, 1 AS count FROM test_tab;
solver | count
=====
Direct |    1
Direct |    1
Direct |    1
Iterativ |    1
(4 строки)
```

Полезный пример с использованием конструкции CASE...END и WHEN...THEN

```
WITH cte_film AS ( --part 1
    SELECT
        film_id,
        title,
        (CASE --start block
            WHEN length < 30 THEN 'Short'
            WHEN length < 90 THEN 'Medium'
            ELSE 'Long'
        END) length
    FROM
        film
)
SELECT --part 2
    film_id,
    title,
    length
FROM
    cte_film
WHERE
    length = 'Long'
```

```
ORDER BY
    title;
```

Пример с использованием логических операторов

```
WITH cte_films AS (
    SELECT
        film_id,
        title,
        (CASE
            WHEN length < 30 THEN 'Short'
            WHEN length >= 30 AND length < 90 THEN 'Medium'
            WHEN length > 90 THEN 'Long'
        END) length
    FROM
        film
)
```

8.2. Конструкция WITH RECURSIVE

Если к WITH добавить RECURSIVE, то можно будет получить доступ к промежуточному результату. Например,

```
WITH RECURSIVE tbl(n) AS ( --part 1
    SELECT 1 --or VALUES(1). This is nonrecursive part
    UNION ALL
    SELECT n+1 FROM tbl WHERE n < 10 --and this is recursive part
)
SELECT sum(n) from tbl; --part 2
```

На первой итерации в таблице `tbl` в атрибуте `n` находится значение 1. На этом вычисления некурсивной части заканчиваются. Далее переходим к вычислениям в рекурсивной части. Таблица `tbl` ссылается на последнее вычисленное значение, поэтому на второй итерации удастся выполнить `n+1`, после чего новым значением таблицы `tbl` станет 2 (`tbl -> 2`). Проверяем условие `n < 10`, а затем переходим к следующей итерации и т.д.

Удобно представлять, что вычисленные значения хранятся в некоторой промежуточной области в порядке вычисления, а таблица `tbl` всегда ссылается на последнее вычисленное значение.

На последнем этапе 1 объединяется с 2, 3 и т.д., т.е. в итоге получается последовательность от 1 до 10. Во второй части запроса остается лишь просуммировать элементы этой последовательности и вывести на экран.

Рассмотрим еще такой пример

```
WITH RECURSIVE
    included_parts(sub_part, part, quantity) AS (
        SELECT --nonrecursive part
            sub_part,
            part,
            quantity
        FROM parts --base table
        WHERE part = "our_product"
        UNION ALL
        SELECT --recursive part
            p.sub_part,
            p.part,
            p.quantity
        FROM
```

```

        included_parts pr,
        parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) AS total_quantity
FROM included_parts
GROUP BY sub_part

```

На первой итерации временная таблица `included_parts`, вычисленная в некурсивной части, представляет собой результат выборки строк и столбцов из таблицы `parts`. В рекурсивной части можно получить доступ к этой таблице. В завершении выполняем выборку из таблицы `included_parts` по столбцу `sub_part`, группируем по нему и выводим сумму по `quantity`.

Еще один полезный пример. Пусть дана таблица сотрудников

employees				
id	name	salary	job	manager_id
=====				
1	John	10000	CEO	null
2	Ben	1400	Junior Developer	5
3	Barry	500	Intern	5
4	George	1800	Developer	5
5	James	3000	Manager	7
6	Steven	2400	DevOps Engineer	7
7	Alice	4200	VP	1
8	Jerry	3500	Manager	1
9	Adam	2000	Data Analyst	8
10	Grace	2500	Developer	8
11	Leor	5000	Data Scientist	6

Выведем иерархию подчинения сотрудников в компании

```

WITH RECURSIVE managers(id, name, manager_id, job, level) AS (
    SELECT id, name, manager_id, job, 1
    FROM employees --base table
    WHERE id = 7
    UNION ALL
    SELECT e.id, e.name, e.manager_id, e.job, m.level+1
    FROM employees e JOIN managers m ON e.manager_id = m.id
)
SELECT * FROM managers;

```

Сначала в *некурсивной* части WITH RECURSIVE объявляется временная таблица `managers(id, name, ...)`. Она строится на базе таблицы `employees`, к которой слева добавляется столбец, состоящий из одних единиц. Затем выбираются строки, удовлетворяющие условию WHERE; в данном случае это одна строка `e.manager_id=m.id`.

И, таким образом, на данном этапе во *временную* таблицу `managers` попадет только одна строка

managers, вычисленная в некурсивной части				
id	name	manager_id	job	level
=====				
7	Alice	1	VP	1

Переходим в рекурсивную часть СТЕ. Из базовой таблицы `employees` выбираем те строки, которые в столбце `manager_id` имеют те же значения, что и в столбце `id` временной таблицы

managers (на данном этапе таблица состоит из одной строки). Другими словами, выбрать нужно те строки, у которых в столбце **manager_id** таблицы **employees** стоит цифра 7.

В результате временная таблица **managers** на текущем этапе будет иметь вид

managers, вычисленная в рекурсивной части

id	name	manager_id	job	level
5	James	7	Manager	2
6	Steven	7	DevOps Engineer	2

Временные таблицы *рекурсивных общих табличных выражений* всегда ссылаются на результат последних вычислений, т.е. на данном этапе временная таблица **managers** ссылается на таблицу, состоящую из двух строк.

Теперь мы снова выбираем из базовой таблицы **employees** и временной таблицы те строки, у которых в столбцах **e.manager_id** и **m.id** стоят одинаковые числа (в данном случае 5 и 6).

Таким образом

managers, вычисленная в рекурсивной части на 2-ой итерации

id	name	manager_id	job	level
2	Ben	5	Junior Developer	3
3	Barry	5	Intern	3
4	George	5	Developer	3
11	Leor	6	Data Scientist	3

Наконец все временные подтаблицы «склеиваются» и конструкция **SELECT * FROM managers** возвращает таблицу **managers**

id	name	manager_id	job	level
7	Alice	1	VP	1 <i>--step 1</i>
5	James	7	Manager	2 <i>--step 2</i>
6	Steven	7	DevOps Engineer	2 <i>--step 2</i>
2	Ben	5	Junior Developer	3 <i>--step 3</i>
3	Barry	5	Intern	3 <i>--step 3</i>
4	George	5	Developer	3 <i>--step 3</i>
11	Leor	6	Data Scientist	3 <i>--step 3</i>

8.3. Изменение данных в WITH

В предложении **WITH** можно также использовать операторы, изменяющие данные (**INSERT**, **UPDATE** или **DELETE**). Это позволяет выполнять в одном запросе сразу несколько разных операций. Например

```
WITH moved_rows AS (
  DELETE FROM products
  WHERE
    dates >= '2010-10-01' AND
    dates < '2010-11-01'
  RETURNING *
)
INSERT INTO products_log (SELECT * FROM moved_rows);
```

Этот запрос фактически перемещает строки из таблицы `products` в таблицу `products_log` (таблица должна уже существовать на момент выполнения запроса). Оператор `DELETE` удаляет указанные строки из `products` и возвращает их содержимое в предложении `RETURNING`, а затем главный запрос читает это содержимое и вставляет в таблицу `products_log`.

9. Работа со строками и регулярные выражения

Больше информации про строковые функции и операторы можно найти на страницах официальной документации PostgreSQL по ссылке <https://postgrespro.ru/docs/postgrespro/9.6/functions-string>.

Пусть дана таблица `employees` вида

id	name	salary	job	manager_id
1	John	10000	CEO	
2	Ben	1400	Junior Developer	5
3	Barry	500	Intern	5
4	George	1800	Developer	5
5	James	3000	Manager	7
6	Steven	2400	DevOps Engineer	7
7	Alice	4200	VP	1
8	Jerry	3500	Manager	1
9	Adam	2000	Data Analyst	8
10	Grace	2500	Developer	8
11	Leor	50000	Data Scientist	6

Выбрать те строки из столбца `job`, в которых содержатся строковые значения, удовлетворяющие шаблону `'_ata %'`, означающий, что первый символ строки может быть любым, а после пробелов может не быть ни одного символа или быть сколько угодно символов. То есть символ `«_»` совпадает с любым символом, а символ `«%»` совпадает с произвольным количеством символов. Здесь используется предложение `LIKE`, которое чувствительно к регистру. В качестве альтернативного варианта можно использовать предложение `ILIKE`², которое не учитывает регистр.

```
SELECT * FROM employees WHERE job LIKE '_ata %';
```

Выведет

id	name	salary	job	manager_id
9	Adam	2000	Data Analyst	8
11	Leor	50000	Data Scientist	6

Подобные задачи можно решать и с помощью предложения `SIMILAR TO`, которое похоже на `LIKE`, но в отличие от последнего при интерпретации шаблонов использует определение регулярного выражения стандарта SQL. Регулярные выражения SQL – это смесь нотации предложения `LIKE` и нотации регулярных выражений.

Шаблоны и `LIKE`, и `SIMILAR TO` должны соответствовать *всей* строке целиком, что, вообще говоря, не согласуется с концепцией обычных регулярных выражений, когда шаблон может соответствовать любой части строки.

`SIMILAR TO`, как и `LIKE` использует символ `«_»` и символ `«%»`, что соответствует `.` и `*` в регулярных выражениях POSIX. Дополнительно поддерживаются символы `|` (указывает альтернативные

²Это расширение PostgreSQL, которое не имеет отношения к стандарту SQL

варианты), * (указывает, что стоящий слева элемент повторяется ноль или более раз), + (указывает, что стоящий слева элемент повторяется один или более раз), (...) (могут использоваться для указания групп), а [...] (определяют символьный класс как в POSIX). Однако ? и {...} не поддерживаются и кроме того «.» не является метасимволом.

Символ «\» экранирует метасимволы, т.е. «снимает» их специальное значение. Другой символ для экранирования можно задать с помощью предложения **ESCAPE**.

Рассмотренную выше задачу можно решить с помощью **SIMILAR TO** следующим образом

```
SELECT * FROM employees WHERE job SIMILAR TO '_ata (A/S)%';
```

Здесь символ «%», означающий произвольную последовательность символов, обязателен, так как шаблоны **SIMILAR TO** должны совпадать со *всей строкой*.

Очень полезна бывает функция **substring()**. Как и **SIMILAR TO** шаблон должен совпадать со всей строкой, например

```
SELECT name, job FROM employees
WHERE substring(job from '%#"Dev#"' for '#')='Dev';
```

Последовательность символов #"..." задают левую и правую скобки группы (можно указать и какой-то другой символ в качестве скобки, например, : "...:", но его нужно указать в ... for ':'). Последовательность, попавшая между этих символов будет возвращена. Как и раньше символы «%» здесь нужны для того чтобы шаблон совпадал со всей строкой.

Обрезать строку можно так

```
SELECT name, job, substring(job,1,4) FROM employees;
```

Здесь первое число – позиция элемента строки (нумерация начинается с единицы), а второе число – число элементов подстроки, которое нужно оставить в выводе.

Эквивалентная конструкция

```
SELECT name, job, substring(job from 1 for 4) FROM employees;
```

Очень гибкое решение дают *регулярные выражения* POSIX (Portable Operating System Interface – переносимый интерфейс операционных систем). Например для того чтобы выбрать, как и в рассмотренном выше примере, всех, кто имеет отношение к работе с данными, можно использовать такую конструкцию

```
SELECT * FROM employees WHERE job ~* 'data .*';
```

Здесь ~* – оператор соответствию шаблону *без* учета регистра. Если нужно учесть регистр, то используется оператор ~.

Аналогично !~ – оператор несоответствия шаблону с учетом регистра, оператор !~* – оператор несоответствия шаблону *без* учета регистра.

Регулярные выражения могут совпадать с строкой в любом месте (не обязательно со всей строкой).

Еще пример. Нужно выбрать строки из столбца **job**, в которых последовательность символов **dev** стоит в начале строки («^» – якорь начала строки)

```
select * from employees where job ~* '^dev.*';
```

Пример с положительной проверкой вперед

```
select * from employees where job ~* '(?=engineer)';
```

Выведет

id	name	salary	job	manager_id
6	Steven	2400	DevOps Engineer	7
20	Alex	25050	Data Engineer	10

У регулярных выражений есть один тонкий нюанс, связанный с «жадностью» квантификатора. Рассмотрим пример

```
SELECT substring('xy1234z', 'y*(\d{1,3})'); -- вернет '123'
```

Здесь используется «жадный» квантификатор `*`. В общем случае этот квантификатор соответствует элементу, который не повторяется ни разу или повторяется произвольное число раз, однако в данном случае он соответствует строго единственному символу `'y'` (больше в строке символов `'y'` нет). Другими словами подшаблон `y*` жадно забирает символ `'y'` и на этом успокаивается, так как больше ничего от этой последовательности взять не сможет. А затем подшаблон `\d{1,3}` жадно забирает 3 цифры, так как ему никто не мешает это сделать.

Но если использовать «нежадный» вариант квантификатора `*?`, то картина изменится

```
SELECT substring('xy1234z', 'y*?(\d{1,3})'); -- вернет '1'
```

Вариант, когда подстрока не имеет ни одного элемента устраивает подшаблон `y*?` (потому что используется нежадный квантификатор), но не устраивает подшаблон `\d{1,3}`, согласно которому в последовательности должна быть хотя бы одна цифра. Приходится как бы «тянуть» подшаблон `y*?` вправо, потому что нежадный квантификатор стремится схлопнуться в пустое начало строки. Таким образом, на 2-ой итерации курсор сдвигается вправо на один символ и теперь указывает на `'x'`. Нежадный подшаблон `y*?` удовлетворен (ему достаточно и пустой подстроки), но не удовлетворен второй подшаблон. Курсор передвигается еще на один элемент вправо. И теперь указывает на `'y'` (подстрока: `'xy'`). Нежадный подшаблон `y*?` снова удовлетворен, но подшаблон `\d{1,3}` все еще не содержит ни одной цифры, поэтому курсор снова перемещается вправо еще на один элемент. В этот раз подстрока выглядит как `'xy1'`, что удовлетворяет и первый, и второй подшаблоны, но подшаблон `y*?` больше бы устроил вариант, когда подстрока пустая. С другой стороны подшаблон `\d{1,3}` требует, чтобы в подстроке была хотя бы одна цифра, поэтому подстрока вида `'xy1'` (в группу попадает только 1) – это компромисс.

Замечание

Удобно представлять, что *нежадные квантификаторы* типа `*?` представляют собой пружинки сжатия, которые тянут влево и в самом простом случае довольствуются пустой последовательностью, а *жадные квантификаторы* (`*`) можно представлять как пружинки растяжения, которые стремятся занять всю последовательность и с неохотой уступают элементы

10. Приемы работы в pgAdmin 4

11. Приемы работы в psql

`psql` – это терминальный клиент для работы с PostgreSQL. Утилита `psql` предоставляет ряд метакоманд и различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

11.1. Конфигурационный файл

Поведением интерактивного терминала `psql` можно управлять с помощью конфигурационного файла `~/.psqlrc`. На ОС Windows этот файл располагается по адресу `C:\Users\ADM\AppData\Roaming\postgresql` и называется `psqlrc.conf`.

Конфигурационный файл утилиты `psql` может включать следующие настройки

`psqlrc.conf`

```
\set QUIET 1
\timing
\pset border 2
\pset null <NULL>
\set ON_ERROR_STOP on
\setenv PSQL_EDITOR "C:\Program Files\Git\usr\bin\vim.exe"
\set VERBOSITY verbose
\set QUIET 0
```

11.2. Метакоманды `psql`

`\c` или `\connect`: устанавливает новое подключение к серверу PostgreSQL. Параметры подключения можно указывать как позиционно, так и передавая аргумент.

Например

```
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

`\cd`: сменяет текущий рабочий каталог на заданный. Без аргументов устанавливает домашний каталог пользователя.

`\conninfo`: выводит информацию о текущем подключении к базе данных.

`\copy`: производит копирование данных с участием клиента. При этом выполнятся SQL-команда `COPY`, но вместо чтения или записи в файл на сервере, `psql` читает или записывает файл и пересылает данные между сервером и локальной файловой системой. Это означает, что для доступа к файлам используются привелегии локального пользователя, а не сервера, и не требуются привелегии суперпользователя SQL. Синтаксис команды похож на синтаксис SQL-команды `COPY`. Все параметры, кроме источника и получателя данных, соответствуют параметрам `COPY`. Альтернативный способ получить тот же результат, что и с `\copy ... to` – использовать SQL-команду `COPY ... TO STDOUT` и завершить ее командой `\g name`.

`\d[S+]`: для каждого отношения (таблицы, представления, материализованного представления, индекса, последовательности, внешней таблицы) или составного типа, соответствующих шаблону, показывает все столбцы, их типы, табличное пространство и любые специальные атрибуты, такие как `NOT NULL` или значения по умолчанию. Также показываются связанные индексы, ограничения, правила и триггеры.

`\da[S]`: выводит список агрегатных функций вместе с типом возвращаемого значения и типами данных, которыми они оперируют.

`\dD[S+]`: выводит список доменов.

`\dL[S+]`: выводит список процедурных языков.

`\dn[S+]`: выводит список схем (пространств имен).

`\encoding`: устанавливает кодировку набора символов на клиенте. Без аргументов команда показывает текущую кодировку.

\!: выполняет команду операционной системы.

11.3. Примеры использования

Вывести список таблиц, присутствующих в базе данных `postgres`, предварительно задав кодовую страницу, соответствующую Windows-кодировке

```
$ psql -U postgres -d postgres -c '! chcp 1251' -c 'd'
```

Вывести первые 5 строк таблицы `aircrafts` из базы данных `demo`, организовав вывод строк в вертикальном формате (\x; эквивалентно ключу `psql -x`)

```
$ psql -U postgres -d demo -c 'x' -c 'TABLE aircrafts LIMIT 5;'
```

Замечание

Лучше для нескольких команд использовать несколько ключей `-c`

Прочитать команды из файла `sql_query.sql`, а не из стандартного ввода. По большому счету ключ `-f` равнозначен метакоманде \i

```
$ psql -U postgres -f sql_query.sql
```

Вывести первые 3 строки таблицы в HTML-формате (-H)

```
$ psql -U postgres -d postgres -H -c '! chcp 1251' -c 'table family limit 3;'
```

Записать вывод результатов всех запросов в файл с именем `psql_output.txt`

```
$ psql -U postgres -d postgres -H -c '! chcp 1251' -c 'table family limit 3;' \
-o psql_output.txt
```

Вывести таблицу в формате L^AT_EX и вывод запроса записать в файл `for_latex_output.txt`

```
$ psql -U postgres -d postgres -P format=latex \
-c '! chcp 1251' -c 'table family limit 3;' \
-o for_latex_output.txt
```

12. Функции, возвращающие множества

12.1. Функции, генерирующие ряды значений

Очень полезна бывает функция `generate_series(start, stop, step)`, которая возвращает последовательность в общем случае вещественных чисел от `start` до `stop` с заданным шагом `step`

```
SELECT generate_series(1.1, 4, 1.3) AS real_num;
```

Выведет

```
+-----+
| real_num |
+-----+
|      1.1 |
|      2.4 |
|      3.7 |
+-----+
(3 строки)
```

Сгенерировать несколько временных меток с шагом в 5 часов

```
SELECT generate_series('2020-05-01 00:00'::timestamp,
                      '2020-05-01 22:00'::timestamp,
                      '5 hours'::interval) AS dates;
```

ВЫВЕДЕТ

```
+-----+
|      dates      |
+-----+
| 2020-05-01 00:00:00 |
| 2020-05-01 05:00:00 |
| 2020-05-01 10:00:00 |
| 2020-05-01 15:00:00 |
| 2020-05-01 20:00:00 |
+-----+
(5 строк)
```

Когда после функции в предложении FROM добавляется WITH ORDINALITY, в выходные данные добавляется столбец типа **bigint**, числа в котором начинаются с 1 и увеличиваются на 1 для каждой строки, выданной функцией. Пример

```
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls, n) LIMIT 5;
```

ВЫВЕДЕТ

```
+-----+---+
|      ls      | n |
+-----+---+
| base         | 1 |
| current_logfiles | 2 |
| global       | 3 |
| log          | 4 |
| pg_commit_ts | 5 |
+-----+---+
```

Список литературы

1. Чакон С., Штрауб Б. Git для профессионального программиста. – СПб.: Питер, 2020. – 496 с.
2. Сوبель М. Linux. Администрирование и системное программирование. 2-е изд. – СПб.: Питер, 2011. – 880 с.