Дата последней сборки документа:
6 июля 2020 г. в 00:29

Сборник заметок по СУБД PostgreSQL

Содержание

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:

- найти файл pg_hba.conf; на ОС Windows он располагается по адресу C: ▸ Program Files ▸ PostgreSQL ▸ 11 ▸ data,
- 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 следущая (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. Подзапросы

По связанности подзапросы делятся на:

- *связанные* (или коррелированные) подзапросы, т.е. такие подзапросы, которые ссылаются на элементы внешнего подзапроса или элементы главного запроса,
- *несвязанные* (или некоррелированные) подзапросы.

Фундаментальная концепция состоит в том, что связанные подзапросы выполняются для *каждой* записи (строки) из внешнего подзапроса (или главного запроса), а несвязанные выполняются только один раз.

7. 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 ',';
```

8. 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';
```

9. 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 (...)`.

Ссылки на общие табличные выражения в главном запросе можно трактовать как имена таблиц. PostgreSQL выполняет общее табличное выражение *только один раз*, кеширует результаты, а затем повторно использует, вместо того чтобы выполнять подзапросы всякий раз, как они встречаются в главном запросе [1, стр. 169].

9.1. 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 orders[1]

```
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
)
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 | Comsole         | 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;
```

[1] См. документацию PostgreSQL https://postgrespro.ru/docs/postgrespro/9.5/queries-with

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
)
```

9.2. 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 удается 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 нерекурсивной части

```
id | name   | manager_id | job              | level
=======================================================
7  | Alice  | 1          | VP               | 1
```

Переходим в рекурсивную часть CTE. 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 --step 1
-------------------------------------------------------------
5  | James   | 7          | Manager          | 2 --step 2
6  | Steven  | 7          | DevOps Engineer  | 2 --step 2
-------------------------------------------------------------
2  | Ben     | 5          | Junior Developer | 3 --step 3
3  | Barry   | 5          | Intern           | 3 --step 3
4  | George  | 5          | Developer        | 3 --step 3
11 | Leor    | 6          | Data Scientist   | 3 --step 3
```

9.3. 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`.

10. Оконные функции

Помимо группировки и агрегирования, PostgreSQL предлагает еще один способ вычислений, затрагивающих несколько записей, – *оконные функции*. В случае группировки и агрегирования выводится одна запись для каждой группы входных записей. Оконные функции делают нечто похожее, но выполняются для каждой записи, а количество записей на входе и на выходе одинаковое.

При работе с оконными функциями группировка не обязательна, хотя и возможна. Оконные функции вычисляются *после группировки* и *агрегирования*. Поэтому в запросе `SELECT` они могут находится только в списке выборки и во фразе `ORDER BY`.

10.1. Определение окна

Синтаксис оконных функций выглядит так

```
<function_name> (<function_arguments>)
OVER (
    [PARTITION BY <expression_list>]
```

```
    [ORDER BY <order_by_list>]
    [{ROWS | RANGE} <frame_start> |
     {ROWS | RANGE} BETWEEN <frame_start> AND <frame_end>]
)
```

Конструкция в скобках после слова `OVER` называется *определением окна*. Последняя его часть, начинающаяся словом `ROWS`, называется *определением фрейма*. Ключевое слово `OVER` делает агрегатную функцию оконной.

Набор записей, обрабатываемых оконной функцией, строится следующим образом. Вначале обрабатывается фраза `PARTITION BY`. Отбираются все записи, для которых выражения, перечисленные в списке `expression_list`, имеют такие же значения, как и в текущей записи. Это множество строк называется *разделом* (partition). Текущая строка также включается в раздел. В общем-то, фраза `PARTITION BY` логически и синтаксически идентична фразе `GROUP BY`, только в ней запрещается ссылаться на выходные столбцы по именам или по номерам.

Иными словами, при обработке каждой записи оконная функция просматривает все остальные записи, чтобы понять, какие из них попадают в один раздел с текущей. Если фраза `PARTITION BY` отсутствует, то на данном шаге создается один раздел, содержащий все записи.

Далее раздел сортируется в соответствии с фразой `ORDER BY`, логически и синтаксически идентичной той же фразе в команде `SELECT`. И снова ссылки на выходные столбцы по имени или по номерам не допускаются. Если фраза `ORDER BY` опущена, то считается, что позиция каждой записи множества одинакова.

Наконец, обрабатывается определение фрейма. Это означает, что мы берем подмножество раздела, которое следует передать оконной функции. Оно и называется оконным фреймом. У фрейма есть начало и конец.

Начало фрейма может принимать следующие значения:

- `UNBOUNDED PRECEDING`: самая первая запись раздела,
- `<N> PRECEDING`: запись, предшествующая текущей в упорядоченном разделе и отстоящая от нее на `N` записей. Здесь `<N>` – целочисленное выражение, которое не может быть отрицательным и не может содержать агрегатные или другие оконные функции. `0 PRECEDING` указывает на текущую запись,
- `CURRENT ROW`: текущая строка,
- `<N> FOLLOWING`: запись, следующая за текущей в упорядоченном разделе и отстоящая от нее на `N` записей.

Конец фрейма может принимать следующие значения:

- `<N> PRECEDING`,
- `CURRENT ROW`,
- `<N> FOLLOWING`,
- `UNBOUNDED FOLLOWING`: последняя запись раздела.

Оконный фрейм можно определить в режиме `ROWS` или `RANGE`. Режим влияет на семантику `CURRENT ROW`. В режиме `ROWS` конструкция `CURRENT ROW` указывает на саму текущую запись, а в режиме `RANGE` – на первую или последнюю запись с такой же позицией, что у текущей, в смысле сортировки, заданной фразой `ORDER BY`.

В режиме `RANGE` в определении фрейма можно использовать варианты `BOUNDED ...` или `CURRENT ROW`, но не `<N> PRECEDING`.

Если часть `frame_end` опущена, предполагается `CURRENT ROW`. Если опущено все определение фрейма, то предполагается определение `RANGE UNBOUNDED PRECEDING`.

Например, конструкция

```
OVER (PARTITION BY a ORDER BY b ROWS BETWEEN UNBOUNDED PRECEDING AND 5 FOLLOWING)
```

означает, что для каждой строки раздел образуют все записи с одинаковым значением в поле a. Затем раздел сортируется в порядке возрастания поля b, а фрейм содержит все записи от первой до пятой после текущей строки.

10.2. Фраза WINDOWS

Определения окон могут быть довольно длинными, и часто использовать их в списке выборки неудобно. PostgreSQL предлагает способ определять и именовать окна, которые затем используются во фразе OVER оконной функции. Это делается с помощью фразы WINDOW команды SELECT, которая может находится после фразы HAVING, например

```
SELECT count() OVER w, sum(b) OVER w, avg(b) OVER (w ORDER BY c ROWS BETWEEN 1 PRECEDING AND 1
    FOLLOWING)
FROM table1
WINDOW w AS (PARTITION BY a)  -- определение окна
```

Определенное таким образом окно можно использовать как есть. В примере выше функции count и sum так и делают. Но его можно и уточнить, как сделано в функции avg.

Синтаксически разница состоит в следующем: чтобы повторно использовать определение окна, имя этого окна следует указать после ключевого слова OVER без скобок. Если же мы хотим расширить определение окна, то имя окна следует указать внутри скобок.

10.3. Использование оконных функций

Все агрегатные функции, в т.ч. определенные пользователем, можно использовать как оконные, за исключением агрегатов по упорядоченным наборам и по гипотетическим наборам. На то, что функция выступает в роли оконной, указывает наличие фразы OVER.

Если агрегатная функция используется в качестве оконной, то она агрегирует строки, принадлежащие *оконному фрейму текущей строки*.

Пример

```
-- общее табличное выражение выполнятся перед главным запросом
WITH monthly_data AS ( -- CTE
    SELECT date_trunc('month', advertisment_date) AS month,
           count(*) AS cnt -- число строк для каждой группы
    FROM car_portal_app.advertisment
    GROUP BY date_trunc('month', advertisment_date)
    -- чтобы не дублировать записи во фразах SELECT и GROUP BY, во фразе GROUP BY
    -- можно сослаться на первый элемент из списка выборки по номеру
    -- т.е. можно было бы написать GROUP BY 1
)
SELECT
    to_char(month, 'YYYY-MM') AS month,
    cnt,
    sum(cnt) OVER ( -- накопительная сумма
        w ORDER BY month
    ) AS cnt_year,
    round(
        avg(cnt) OVER ( -- скользящее среднее
            ORDER BY month
            ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
```

```
        ),
    1) AS mov_avg,
    -- для того чтобы деление было вещественным нужно, чтобы хотя бы один операнд
    -- был приведен к вещественному типу
    -- round работает только с типом numeric
    round((cnt::numeric / sum(cnt) OVER w) * 100, 2) AS ratio_year -- здесь sum(cnt) вычисляется
     для всего раздела без учета порядка следования записей в таблице
FROM monthly_data -- обращение к результату работы CTE
WINDOW w AS (PARTITION BY date_trunc('year', month)); -- определение окна
```

Выведет

```
+---------+-----+----------+------------+
|  month  | cnt | cnt_year | ratio_year |
+---------+-----+----------+------------+
| 2014-01 |  42 |       42 |       5.78 |
| 2014-02 |  49 |       91 |       6.74 |
| 2014-03 |  30 |      121 |       4.13 |
| 2014-04 |  57 |      178 |       7.84 |
| 2014-05 | 106 |      284 |      14.58 |
| 2014-06 | 103 |      387 |      14.17 |
...
```

Здесь общее табличное выражение вычисляется перед главным запросом, так как имя временной таблицы `monthly_data`, связанной с CTE, присутствует в главном запросе. Результат кешируется и используется повторно.

Сначала, как обычно вычисляются объекты, указанные в предложении `FROM`, т.е. в данном случае таблица `advertisment` (дополнительно указана схема базы данных `car_portal_app`). Группировка выполняется по временной метке «усеченной» до месяца (т.е. в формате `'2020-07-01 00:00:00+03'`).

В предложение `SELECT` конструкция, указанная во фразе `GROUP BY`, включается полностью, без изменений и связывается с псевдонимом `month`. Как известно, в *список выборки* фразы `SELECT` в случае группировки могут включаться только атрибуты, указанные в `GROUP BY` и агрегатные функции.

Результат работы общего табличного выражения будет доступен через имя `monthly_data`.

Переходим к главному запросу. Здесь читается временная таблица `monthly_data` и из нее выбираются атрибуты, перечисленные в списке выборки. Функция `to_char` преобразует временную метку к формату `'YYYY-MM'` и связывается с псевдонимом `month`. Атрибут `cnt` выбирается как есть, без изменений.

Далее, первая оконная функция `sum` использует окно `w`, описанное с помощью ключевого слова `WINDOW` и расширенное `ORDER BY`. Определение фрейма в данном случае опущено, поэтому предполагается фрейм `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Следовательно, функция вычисляет сумму значений по всем записям, начиная с начала года и до текущего месяца включительно, т.е. накопительный итог за год.

Вторая функция `avg` для каждой записи вычисляет скользящее среднее по пяти записям – две предшествующие, текущая и две последующие. Ранее определенное окно не используется, потому что при вычислении скользящего среднего год не принимается во внимание, важен только порядок записей.

Третья оконная функция `sum` пользуется тем же определением окна `w`. Вычисленная ею сумма значений за год является знаменателем в выражении, которое дает вклад текущего месяца в сумму. Очень важный момент заключается в том, что в данном случае фраза `ORDER BY` опущена

и это значит, что накопительная сумма вычисляется независимо от порядка следования записей в таблице, и потому возвращается только одно число, равное накопительной сумме по разделу, а не как в первом случае вычисления накопительной суммы (там использовалась фраза `ORDER BY`).

Существует ряд *оконных функций*, не являющихся *агрегатными*. Они служат для получения значений других записей в разделе, для вычисления ранга текущей записи относительно остальных и для генерации номеров строк.

Изменим предыдущий отчет, будет считать, что требуется вычислить две разности: между количеством объявлений в текущем месяце и предыдущем месяце того же года и в том же месяце прошлого года, а также ранг текущего месяца. Вот как выглядит подзапрос

```
WITH monthly_data AS ( -- CTE, выполняется прежде главного запроса
    SELECT date_trunc('month', advertisment_date) AS month,
           count(*) as cnt
    FROM car_portal_app.advertisment
    GROUP BY date_trunc('month', advertisement_date)
)
SELECT to_char(month, 'YYYY-MM') AS month,
       cnt,
       cnt - lag(cnt) OVER (ORDER BY month) AS prev_m,
       cnt - lag(cnt,12) OVER (ORDER BY month) AS prev_y,
       rank() OVER (w ORDER BY cnt DESC) AS rank
FROM monthly_data
WINDOW w AS (PARTITION BY date_trunc('year', month))
ORDER BY month DESC;
```

Выведет

```
+---------+-----+--------+--------+------+
|  month  | cnt | prev_m | prev_y | rank |
+---------+-----+--------+--------+------+
| 2015-02 |   9 |    -39 |    -40 |    2 |
| 2015-01 |  48 |     11 |      6 |    1 |
| 2014-12 |  37 |      1 | <NULL> |   10 |
| 2014-11 |  36 |    -10 | <NULL> |   11 |
| 2014-10 |  46 |    -36 | <NULL> |    8 |
| 2014-09 |  82 |     12 | <NULL> |    3 |
...
```

Функция `lag` возвращает значение указанного выражения для записи, отстоящей на заданное число записей (по умолчанию 1) назад от текущей записи. Другими словами, функция `lag` просто смещает значения в столбце на заданное число записей *вниз*, замещая отсутствующие значения в верхней части с помощью `<NULL>`.

Пример

```
-- здесь создается один раздел, упорядоченные по столбцу `n`
SELECT n,
       lag(n) OVER (ORDER BY n) AS shift
FROM generate_series(10,15) AS t(n);
```

Выведет

```
+----+--------+
| n  | shift  |
+----+--------+
| 10 | <NULL> |
| 11 |     10 |
| 12 |     11 |
```

```
...
```

Как видно, в феврале 2015 года опубликовано 9 объявлений – на 39 меньше, чем в январе 2015.

Функция `rank` возвращает ранг текущей строки внутри раздела. Имеется в виду ранг с промежутками, т.е. в случае, когда две записи занимают одинаковую позицию в порядке, заданном фразой `ORDER BY`, обе получают одинаковый ранг, а следующая получает ранг на две единицы больше. То есть у нас будет две первые записи и одна третья.

Перечислим другие оконные функции:

- `lead`: аналогична `lag`, но возвращает значения выражения, вычисленное для записи, отстоящей от текущей на указанное количество записей вперед, т.е. смещает элементы столбца вверх, замещая отсутствующие значения в нижней части с помощью `<NULL>`,
- `first_value`, `last_value`, `nth_value`: возвращает значения выражения, вычисленные соответственно для первой, последней и n-ой записи,
- `row_number`: возвращает номер текущей строки в разделе,
- `dense_rank`: возвращает ранг текущей строки без промежутков (плотный ранг),
- `percent_rank` и `cume_dist`: возвращает относительный ранг текущей строки. Разница между функциями состоит в том, что в первой числителем дроби является ранг, а во второй – номер строки,
- `ntile`: делит раздел на заданное количество равных частей и возвращает номер части, в которую попала текущая строка.

11. Продвинутые методы работы с SQL

11.1. Выборка первых записей

Часто бывает необходимо найти первые записи относительно какого-то критерия. Допустим, к примеру, что в базе данных `car_portal` нужно найти первое объявление для каждого идентификатора `car_id` в таблице `advertisement`.

```
SELECT
    advertisement_id,
    advertisement_date,
    adv.car_id,
    seller_account_id
FROM
    car_portal_app.advertisement AS adv
INNER JOIN
    (SELECT
        car_id,
        min(advertisement_date) AS min_date
     FROM
         car_portal_app.dvertisement
     GROUP BY car_id) AS first
ON adv.car_id = first.car_id
   AND adv.advertisement_date = first.min_date;
```

Но если логика упорядочения настолько сложна, что для ее реализации функции `min` недостаточно, то такой подход работать не будет. Проблему могут решить оконные функции, но они не всегда удобны

```
SELECT DISTINCT
    first_value(advertisement_id) OVER w AS advertisement_id,
    min(advertisement_date) OVER w AS advertisement_date,
    car_id,
    first_value(seller_account_id) OVER w AS seller_account_id
FROM
    car_portal_app.advertisement
WINDOW w AS (PARTITION BY car_id ORDER BY advertisement_date);
```

PostgreSQL предлагает явный способ выбрать первую запись из каждой группы – ключевое слово `DISTINCT ON`. Для каждой уникальной комбинации значений из списка выражений команды `SELECT` возвращает только первую запись. Для определения того, что такое «первая» запись, служит фраза `ORDER BY`

```
SELECT DISTINCT ON (car_id) advertisement_id, advertisement_date, car_id, seller_account_id
FROM car_portal_app.advertisement
ORDER BY car_id, advertisement_date;
```

Другой пример. Пусть дана следующая таблица

```
+---------------------+-----+
|        month        | cnt |
+---------------------+-----+
| 2014-01-10 00:00:00 |  42 | <- превая фраза раздела
| 2014-02-10 00:00:00 |  49 |
| 2014-03-10 00:00:00 |  30 |
| 2014-04-10 00:00:00 |  57 |
| 2014-05-10 00:00:00 | 106 |
| 2014-06-10 00:00:00 | 103 |
| 2014-07-10 00:00:00 |  69 |
| 2014-08-10 00:00:00 |  70 |
| 2014-09-10 00:00:00 |  82 |
| 2014-10-10 00:00:00 |  46 |
| 2014-11-10 00:00:00 |  36 |
| 2014-12-10 00:00:00 |  37 |
| 2015-01-10 00:00:00 |  48 | <- превая фраза раздела
| 2015-02-10 00:00:00 |   9 |
+---------------------+-----+
```

Требуется из каждой группы, сформированной по году, выбрать первую запись. Записи упорядочиваются по атрибуту `month`

```
SELECT DISTINCT ON (date_trunc('year', month)) cnt, to_char(month, 'YYYY-MM') AS month
FROM monthly_data;
```

Выведет

```
+-----+---------+
| cnt |  month  |
+-----+---------+
|  42 | 2014-01 |
|  48 | 2015-01 |
+-----+---------+
```

То есть конструкция `DISTINCT ON` выполняет группировку по указанному атрибуту. И фраза `ORDER BY` не обязательна.

12. 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 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.

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

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 указывате 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

---

13. Приемы работы в pgAdmin 4

14. Приемы работы в psql

`psql` – это терминальный клиент для раоботы 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.

14.1. 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 QUITE 0
```

14.2. 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).

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

`\!`: выполняет команду операционной системы.

14.3. Примеры использования

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

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

Замечание

Лучше для нескольких команд использовать несколько ключей `-c`

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

```
$ psql -U postgtes -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 LaTeX 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
```

15. Специальные функции и операторы

15.1. Предикаты ANY, ALL

Использование `IN` эквивалентно `= ANY`, а использование `NOT IN` эквивалентно `<> ALL`. Пример

```
-- есть ли совпадение хотя бы с одним элементом массива?
SELECT 'test'::text = ANY('{"test","non-test"}'::text[]); -- true
-- есть ли совпадение со всеми элементами массива?
SELECT 'test'::text = ALL('{"test","non-test"}'::text[]); -- false
```

15.2. Оператор конкатенации

Соединить два массива можно с помощью оператора `||`

```
SELECT '{2,3,10}'::int[] || '{5}'::int[]; -- {2,3,10,5}
```

С помощью этого же оператора можно «склеивать» строки

```
SELECT name || '[ ' || job ' ]' AS record FROM employees; -- Ben[ Junior Developer ]
```

К слову, у PostgreSQL нестрогая типизация. То есть PostgreSQL проводит неявное преобразование типов, например, когда используется оператор конкатенации, склеивающий строку и число, то выполняется неявное преобразование числового типа в строковый.

15.3. Диапазонные операторы

Подробнее вопрос обсуждается в документации PostgreSQL 9.19. Диапазонные функции и операторы.

С помощью оператора `@>` удобно проверять покрывает ли диапазон слева от оператора диапазон справа от оператора

```
SELECT '{2,3,5}'::int[] @> '{3}'::int[]  -- true
```

Можно проверить содержит ли диапазон слева от оператора элемент справа от оператора

```
SELECT '[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp;  --true
```

Если требуется выяснить содержится ли элемент в диапазоне, то используется оператор `<@`

```
SELECT 42 <@ int4range(2,10); --false
-- принадлежит ли 3 полуотрезку [2,10)
SELECT 3 || ' in ' || int4range(2,10) AS cond, 3 <@ int4range(2,10) AS bool;  --true
```

Общие элементы (пересечение) удобно выявлять с помощью оператора `&&`

```
SELECT '{3}'::int[] && '{2,3,5}'::int[]; --true
```

15.4. 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 |
+------------------+---+
```

Результаты нескольких табличных функций можно объединить, как если бы они были соединены по позиции строки. Для этого служит конструкция ROWS FROM. Эта конструкция возвращает *отношение*, поэтому ее можно использовать во фразе FROM, как любое другое отношение. Количество строк равно размеру самого большого результата перечисленных функций. Столбцы соответствуют функциям во фразе ROWS FROM. Если какая-то функция возвращает меньше строк, чем другие, то отсутствующие значения будут равны <NULL>

```
SELECT idx, number FROM
    ROWS FROM (
        generate_series(
            '2020-07-06'::timestamp,
            '2020-10-05'::timestamp,
            '14 days'::interval
        ),
        generate_series(100,105)
    ) as t(idx, number);
```

выведет

```
+---------------------+--------+
|         idx         | number |
+---------------------+--------+
| 2020-07-06 00:00:00 |    100 |
| 2020-07-20 00:00:00 |    101 |
| 2020-08-03 00:00:00 |    102 |
| 2020-08-17 00:00:00 |    103 |
| 2020-08-31 00:00:00 |    104 |
| 2020-09-14 00:00:00 |    105 |
| 2020-09-28 00:00:00 | <NULL> |
+---------------------+--------+
```

15.5. Функции для работы с массивами

Пример использования функции unnest, которая преобразует заданный *массив* в набор *строк*, каждая запись которого соответствует элементу массива

```
SELECT unnest('{2,3,4}'::int[]) AS idx, 5 AS number;
```

выведет

```
+-----+--------+
| idx | number |
+-----+--------+
|   2 |      5 |
|   3 |      5 |
|   4 |      5 |
+-----+--------+
```

Работа с несколькими массивами

```
SELECT * FROM unnest('{2,3,4}'::int[], '{10,20,30,40}'::int[]);
```

15.6. Операторы и функции даты/времени

Вычислить возраст, например, пользователя приложения по дате его рождения удобно с помощью функции `age`

```
SELECT age('1986-08-18'::timestamp); -- 33 years 10 mons 16 days
```

В этом же запросе можно извлечь, например, только полное число лет

```
SELECT extract( 'year' from age('1986-08-18'::timestamp)); -- 33
```

Перекрытие временных диапазонов можно определить с помощью SQL-оператора `OVERLAPS`. Пример

```
SELECT
   ('2001-02-16'::date, '2001-12-21'::date)
   OVERLAPS
   ('2001-10-30'::date, '2002-10-30'::date); --t
```

«Усечь» дату до заданной точности можно так

```
SELECT date_trunc('year', '2001-02-16 20:38:40'::timestamp); -- 2001-01-01 00:00:00
```

16. Наиболее полезные команды PostgreSQL

16.1. Команда `VACUUM`

Команда `VACUUM` предназначена для сборки мусора и, возможно, для анализа базы данных. `VACUUM` высвобождает пространство, занятое «мертвыми» кортежами. При обычных операциях кортежи, удаленные или устаревшие в результате обновления, физически не удаляются из таблицы. Они сохраняются в ней, и ждут выполнения команды `VACUUM`. Таким образом, периодически необходимо выполнять `VACUUM`, особенно для часто изменяемых таблиц.

Без параметра команда `VACUUM` обрабатывает *все таблицы* текущей базы данных, которые может очистить текущий пользователь. Если в параметре передается имя таблицы, то `VACUUM` обрабатывает только эту таблицу.

Конструкция `VACUUM ANALYZE` выполняет очистку, а затем анализ всех указанных таблиц. Параметр `ANALYZE` обновляет статистическую информацию по таблицам, которую использует планировщик при выборе наиболее эффективного способа выполнения запроса. Это удобная комбинация для регулярного обслуживания БД.

Пример

```
VACUUM (VERBOSE, ANALYZE) onek;
```

Список литературы

1. *Джуба С.*, *Волков А.* Изучаем PostgreSQL 10. – М.: ДМК Пресс, 2019. – 400 с.
2. *Чакон С.*, *Штрауб Б.* Git для профессионального программиста. – СПб.: Питер, 2020. – 496 с.
3. *Собель М.* Linux. Администрирование и системное программирование. 2-е изд. – СПб.: Питер, 2011. – 880 с.