БАЗЫ ДАННЫХ

Лекция 8

ПЛАН НА СЕГОДНЯ

- Немного про группировки
- Оконные функции
- CTE

РАБОЧАЯ ТАБЛИЦА

- Возьмем dataset - качество воздуха и загрязненность воды
- Перетащим из CVS
- Заодно разомнемся
- Сначала заполним временную таблицу, в которой все поля - строки

ПЕРВИЧНОЕ ЗАПОЛНЕНИЕ

```
CREATE TABLE cities (city TEXT, region TEXT, country TEXT, aq TEXT,

COPY cities FROM '/tmp/cities.csv' DELIMITER ',' CSV HEADER ;
 -- нужен абсолютный путь

SELECT * FROM cities;
```

ОБРЕЖЕМ ЛИШНИЕ ПРОБЕЛЫ

```
UPDATE cities SET city = TRIM(city), region = TRIM(region),
                  country = TRIM(country), aq = TRIM(aq),
                  wp = TRIM(wp);

SELECT * FROM cities;
```

ПРОДОЛЖАЕМ

- Нормализуем строковые поля
- Сделаем aq/wp числовыми
- Добавим столбцы через ALTER TABLE
- А потом лишние удалим

ПРОДОЛЖАЕМ

```
ALTER TABLE cities ADD COLUMN aq_n NUMERIC ;
ALTER TABLE cities ADD COLUMN wp_n NUMERIC ;

UPDATE cities SET aq_n = aq::NUMERIC, wp_n = wp::NUMERIC;

ALTER TABLE cities DROP COLUMN aq  ;
ALTER TABLE cities DROP COLUMN wp  ;
```

ЧТО ДАЛЬШЕ

- Создадим таблицу стран
- Заполним ее уникальными странами из нашей таблицы
- И перенумеруем
- Потом заменим страну на номер в таблице

ПРОДОЛЖАЕМ

```
CREATE TABLE contries (id INT, name TEXT);
INSERT INTO countries (name)
        SELECT DISTINCT country FROM cities ORDER BY country ;

SELECT * FROM countries;
```

ПРОДОЛЖАЕМ

```
ALTER TABLE cities ADD COLUMN country_id INT;
UPDATE cities SET country_id = c.id
        FROM countries c WHERE cities.country = c.name;

SELECT * FROM cities;

ALTER TABLE cities DROP COLUMN country;
```

ПРОДОЛЖАЕМ

```
CREATE TABLE cities_data (id INT GENERATED ALWAYS AS IDENTITY,
                          country_id INT, name TEXT);
INSERT INTO cities_data (country_id, name)
        SELECT DISTINCT  country_id, city FROM cities ORDER BY city

SELECT * FROM cities;
SELECT * FROM cities_data;
```

ПРОДОЛЖАЕМ

```
ALTER TABLE cities ADD COLUMN city_id INT;
UPDATE cities SET city_id = c.id
        FROM cities_data c WHERE cities.city = c.name;

SELECT * FROM cities;

ALTER TABLE cities DROP COLUMN country_id;
ALTER TABLE cities DROP COLUMN city;

SELECT * FROM cities;
```

РЕГИОНЫ

- Некоторые регионы отсутствуют
- Здесь отсутствие - это пустая строка
- Создадим отдельную таблицу регионов
- Из нее будет ссылка на таблицу стран

ПРОДОЛЖАЕМ

```
CREATE TABLE regions (id INT GENERATED ALWAYS AS IDENTITY,
                      country_id INT, name TEXT);

INSERT INTO regions (country_id, name)
        SELECT DISTINCT  cities_data.country_id, region
              FROM cities JOIN cities_data ON cities.city_id = citi
              WHERE LENGTH(region) > 0 ORDER BY region ;

SELECT * FROM regions;
```

ПРОДОЛЖАЕМ

```
SELECT c.city_id, r.id region_id
    INTO city_region FROM cities c JOIN regions r
    ON c.region = r.name ORDER BY region_id, city_id  ;

SELECT * FROM city_region;

ALTER TABLE cities DROP COLUMN region ;

SELECT * FROM cities;
```

ГРУППИРОВКА

- Прием объединения записей по общему значению
- В базовом варианте - по совпадающему значению атрибута
- Но не обязательно
- В общем случае - по совпадающему значению функции от строки таблицы

ОСОБЕННОСТИ

- Мы теряем идентичность записей
- В выдаче можем использовать поля, входящие в ключ группировки
- И значения агрегирующих функций
- Кроме стандартного набора Postgres предлагает менее стандартные
- И можно писать свои

ОСОБЕННОСТИ

- Две фильтрации
- WHERE определяет то, что идет на вход группировки
- Если хотим фильтровать результаты группировки, WHERE не подходит в принципе
- Есть отдельная конструкция - HAVING

ПРИМЕРЫ

```
SELECT city_id, COUNT(*)   FROM cities GROUP BY city_id  ;

SELECT city_id, COUNT(*) c  FROM cities GROUP BY city_id
   HAVING COUNT(*) > 1 ;
-- c - нельзя (странность postgres)

-- SELECT city_id, COUNT(*) c   FROM cities
   WHERE COUNT(*) > 1 GROUP BY city_id ;
-- так - нельзя

SELECT city_id, COUNT(*) c  FROM cities WHERE aq > 0
     GROUP BY city_id  ;

SELECT city_id, COUNT(*) c  FROM cities WHERE aq > 0
     GROUP BY city_id HAVING COUNT(*) > 1 ;
```

ПРИМЕРЫ

```
SELECT COUNT(*), AVG(aq), AVG(wp) c
    FROM cities  GROUP BY (RANDOM()*10) :: INT  ;

SELECT (RANDOM()*10) :: INT grp, COUNT(*), AVG(aq), AVG(wp) c
    FROM cities  GROUP BY grp ORDER BY grp  ;

 SELECT (aq / 10)::INT aq_level, COUNT(*), AVG(aq), AVG(wp)
    FROM cities GROUP BY aq_level ;
```

ПРИМЕРЫ

```
-- Страны с количеством городов, представленных в датасете
-- В порядке убывания
-- Те, у которых больше 10 городов
SELECT countries.name, q.count  FROM countries
    JOIN (SELECT cd.country_id cid, COUNT(*) FROM cities_data cd
          GROUP BY cd.country_id )  q
    ON q.cid = countries.id
    WHERE count > 10 ORDER BY count DESC;
```

ОСОБЕННОСТИ

- Группировка не "масштабируется"
- Например: можно определить средней показатель загрязненности по городам
- И можно определить максимальный по таблице
- И то, и другое - одним запросом

ПРИМЕРЫ

```
SELECT MAX(wp), MAX(aq) FROM cities;

SELECT city_id, AVG(wp), AVG(aq) FROM cities GROUP BY city_id;

-- SELECT city_id, MAX(AVG(wp)), MAX(AVG(aq)) FROM cities GROUP BY
-- так нельзя

-- SELECT MAX(AVG(wp)), MAX(AVG(aq)) FROM cities GROUP BY city_id;
-- и даже так
```

ПРИМЕРЫ

```
SELECT MAX(awp), MAX(aaq) FROM (SELECT AVG(wp) awp , AVG(aq) aaq
    FROM cities GROUP BY city_id) q ;

SELECT AVG(awp), AVG(aaq) FROM (SELECT MAX(wp) awp , MAX(aq) aaq
    FROM cities GROUP BY city_id) q ;

WITH q AS (SELECT MAX(wp) awp , MAX(aq) aaq FROM cities GROUP BY ci
     SELECT AVG(awp), AVG(aaq) FROM  q;
```

ОКОННЫЕ ФУНКЦИИ

- Дополнительная возможность Postgres
- Общая идея: в ходе запроса собирается некий агрегирующий контекст
- Этот контекст доступен записи
- Но запись не теряет своей идентичности
- Разберем на примерах

ОКОННЫЕ ФУНКЦИИ

- Пример: хотим посчитать средний показатель по стране
- И вывести его рядом с показателем конкретного замера

```
SELECT c.wp, cd.country_id, AVG (c.wp) OVER (PARTITION BY cd.cou
    FROM cities c JOIN cities_data cd ON c.city_id = cd.id
    ORDER BY country_id;

SELECT c.wp, cd.country_id, AVG (c.wp) OVER (PARTITION BY cd.cou
    FROM cities c JOIN cities_data cd ON c.city_id = cd.id ORDER
```

РАЗБЕРЕМСЯ

- Начинается с агрегирующей функции
- Потом идет ключевое слово OVER
- Оно означает, что мы считаем агрегацию по группе(окну), в которую входит текущая запись
- А критерий разбиения - в скобках

РАЗБЕРЕМСЯ

- PARTITION можно пропустить
- Тогда окном будет вся таблица
- Можно определить порядок элементов внутри окна
- Для некоторых оконных функций это важно

RANK

- Пример: функция rank
- Возвращает номер записи в окне
- С точки зрения заданного порядка
- Для равных значений rank одинаков

ПРИМЕР

```
SELECT cd.name, c.wp, RANK() OVER (PARTITION BY cd.country_id ORDER
                c.aq, RANK() OVER (PARTITION BY cd.country_id
                                   ORDER BY c.aq DESC)
    FROM cities c JOIN cities_data cd ON c.city_id = cd.id ;

SELECT cd.name, c.wp, RANK() OVER (cw ORDER BY c.wp),
                c.aq, RANK() OVER (cw ORDER BY c.aq DESC)
    FROM cities c JOIN cities_data cd ON c.city_id = cd.id
    WINDOW cw AS (PARTITION BY cd.country_id) ;
```

WINDOW

- WINDOW задает именованный раздел
- Можно уточнять именованный раздел
- Переопределять элементы нельзя

ПРИМЕР

```
SELECT cd.name, countries.name, c.wp,
       RANK() OVER (ctw_wp), ROW_NUMBER() OVER (ctw_wp),
       c.aq, RANK() OVER (ctw_aq)
   FROM cities c JOIN cities_data cd ON c.city_id = cd.id
                 JOIN countries ON cd.country_id = countries.id
   WINDOW  cw AS (PARTITION BY cd.country_id),
           ctw AS (PARTITION BY cd.id),
           ctw_wp AS (cw ORDER BY c.wp),
           ctw_aq AS (cw ORDER BY c.aq DESC);
```

ДЕТАЛИ

- PARTITION делит таблицу на окна
- Но можно учитывать не все окно
- Если нет порядка, то по умолчанию рассматривается все окно
- Если есть - то от начала до текущей строки

ИДЕЯ

- Рассматривается поддиапазон, у которого задается начало и конец
- Конец можно задать неявно
- Тогда это будет текущая строка
- Можно задавать в трех "единицах": GROUPS, ROWS, RANGE

ПРИМЕРЫ

```
SELECT cd.name, countries.name,
       aq, AVG(c.aq) OVER (cw_aq),
       AVG(c.aq) OVER (cw_aq ROWS 2 PRECEDING),
       AVG(c.aq) OVER (cw)
    FROM cities c JOIN cities_data cd ON c.city_id = cd.id
    JOIN countries ON cd.country_id = countries.id
    WINDOW  cw AS (PARTITION BY cd.country_id),
            ctw AS (PARTITION BY cd.id),
            ctw_wp AS (cw ORDER BY c.wp),
            cw_aq AS (cw ORDER BY c.aq DESC);
```

ПРИМЕРЫ

```
SELECT cd.name, countries.name,
       aq, AVG(c.aq) OVER (cw_aq),
       AVG(c.aq) OVER (cw_aq ROWS 2 PRECEDING),
       AVG(c.aq) OVER (cw)
    FROM cities c JOIN cities_data cd ON c.city_id = cd.id
    JOIN countries ON cd.country_id = countries.id
    WINDOW  cw AS (PARTITION BY cd.country_id),
            ctw AS (PARTITION BY cd.id),
            ctw_wp AS (cw ORDER BY c.wp),
            cw_aq AS (cw ORDER BY c.aq DESC)
      ORDER by countries.name, aq ;
```

ЕДИНИЦЫ СМЕЩЕНИЯ

- ROWS - строки
- GROUPS - группы
- В группу входят строки, равные по критерию упорядоченности
- RANGE - диапазон значений

ПРИМЕР

```
CREATE TABLE simple(v1 INT, v2 INT);
INSERT INTO simple VALUES (2, 10);
INSERT INTO simple VALUES (2, 5);
INSERT INTO simple VALUES (0, 20);
INSERT INTO simple VALUES (10, 20);
INSERT INTO simple VALUES (15, 15);

SELECT * FROM simple;
SELECT v1, SUM(v1) OVER (w1),
       SUM(v1) OVER (w1 ROWS 0 PRECEDING),
       SUM(v1) OVER (w1 ROWS 1 PRECEDING),
       v2, SUM(v2) OVER (w2),
       SUM(v2) OVER (w2 ROWS 0 PRECEDING),
       SUM(v2) OVER (w2 ROWS 1 PRECEDING)
    FROM simple
    WINDOW w1 AS (ORDER BY v1),
           w2 AS (ORDER BY v2);
```

ПРИМЕР

```
SELECT v1, SUM(v1) OVER (w1),
       SUM(v1) OVER (w1 GROUPS 0 PRECEDING),
       SUM(v1) OVER (w1 GROUPS 1 PRECEDING),
       v2, SUM(v2) OVER (w2),
       SUM(v2) OVER (w2 GROUPS 0 PRECEDING),
       SUM(v2) OVER (w2 GROUPS 1 PRECEDING)
    FROM simple
    WINDOW w1 AS (ORDER BY v1),
           w2 AS (ORDER BY v2);
```

ПРИМЕР

```
SELECT v1, SUM(v1) OVER (w1),
       SUM(v1) OVER (w1 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING),
       v2, SUM(v2) OVER (w2),
       SUM(v2) OVER (w2 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    FROM simple
    WINDOW w1 AS (ORDER BY v1),
           w2 AS (ORDER BY v2);
```

ПРИМЕР

```
SELECT v1, SUM(v1) OVER (w1),
       SUM(v1) OVER (w1 RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING),
       v2, SUM(v2) OVER (w2),
       SUM(v2) OVER (w2 RANGE BETWEEN 4 PRECEDING AND 7 FOLLOWING)
    FROM simple
    WINDOW w1 AS (ORDER BY v1),
           w2 AS (ORDER BY v2);
```

ИСКЛЮЧЕНИЕ

- EXCLUDE CURRENT ROW - не считаем текущую строку
- EXCLUDE GROUP - не считаем текущую строку с ее группой
- EXCLUDE TIES - не считаем группу текущей строки, а ее саму - считаем
- EXCLUDE NO OTHERS - никого не исключаем

ПРИМЕР

```
SELECT v1, SUM(v1) OVER (w1),
       SUM(v1) OVER (w1 RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING
                      EXCLUDE CURRENT ROW),
       SUM(v1) OVER (w1 RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING
                      EXCLUDE GROUP),
       SUM(v1) OVER (w1 RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING
                      EXCLUDE TIES),
       v2, SUM(v2) OVER (w2),
       SUM(v2) OVER (w2 RANGE BETWEEN 4 PRECEDING AND 7 FOLLOWING
                      EXCLUDE CURRENT ROW),
       SUM(v2) OVER (w2 RANGE BETWEEN 4 PRECEDING AND 7 FOLLOWING
                      EXCLUDE GROUP),
       SUM(v2) OVER (w2 RANGE BETWEEN 4 PRECEDING AND 7 FOLLOWING
                      EXCLUDE TIES)
    FROM simple
    WINDOW w1 AS (ORDER BY v1),
           w2 AS (ORDER BY v2);
```

ССЫЛКИ

- https://www.postgresql.org/docs/current/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS
- https://www.postgresql.org/docs/current/functions-window.html
- https://www.postgresql.org/docs/current/queries-table-expressions.html#QUERIES-WINDOW

CTE

- Common Table Expression
- Если просто CTE, есть рекурсивные CTE
- Часто говорят CTE, подразумевая рекурсивные CTE
- Просто CTE - сахар над подзапросами
- Были в примерах

ПРИМЕР

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
) SELECT sum(n) FROM t;
```

КАК РАБОТАЕТ

- UNION / UNION ALL - обязательный элемент рекурсивного CTE
- До UNION - нерекурсивный элемент
- Две абстракции - результат и рабочая область
- Сначала вычисляется нерекурсивный элемент
- Кладется в результат и рабочую область

КАК РАБОТАЕТ

- Вычисляется рекурсивная часть
- t - ссылка на рабочую область
- То, что вышло - добавляется в результат
- И пишется в рабочую область

КАК РАБОТАЕТ

- Продолжаем, пока рабочая область не пустая
- И в итоге можем сделать запрос, сославшись на t
- И получить все, что накоплено в результате
- Не обязательно в SELECT, можно в INSERT или в UPDATE

ИСПОЛЬЗОВАНИЕ

- В реальной жизни - для обхода иерархических структур
- Или графовых зависимостей
- В общем случае - JOIN переменной длины
- Определяемой по ходу JOIN-а
- Можно зациклиться, надо ставить барьеры